Advanced Database Systems - Lecture Notes

Attila Dr. Adamkó

Advanced Database Systems - Lecture Notes

Attila Dr. Adamkó

Publication date 2014

Table of Contents

I. Advanced Database Systems - Lecture Notes 2
1. Introduction 4
2. Basics of XML 5
1. Namespaces and reuseability 7
2. Valid XML documents 9
2.1. XML dialects: DTD and XML Schema 10
2.1.1. The most important XML Schema elements 10
3. Information content and processing strategies 12
3.1. Language independent processing strategies: DOM and SAX 14
4. The fundamentals of document design 16
4.1. Descriptive- and data-oriented document structures 17
4.2. Building Blocks: Attributes, Elements and Character Data 19
4.2.1. The Differences Between Elements and Attributes 20
4.2.2. Use attributes to identify elements 22
4.2.3. Avoid using attributes when order is important 23
4.3. Pitfalls 23
3. XML databases 29
1. Native XML databases 30
1.1. Schema-free native XML database collections 30
1.2. Indexing 32
1.3. Classification of XML databases based on their contents 33
1.4. Usage of native XML databases 33
2. Hybrid systems 34
2.1. Problems of the fully XML based storage layers 35
2.2. Relational repositories with XML wrapping layer 36
2.3. Hybrid repositories 36
2.4. Other considerations 38
4. XDM: the data model for XPath and XQuery 41
5. XPath 46
1. Expressions 48
1.1. Steps 49
1.1.1. Axes 49
1.1.2. XPath Node test 52
1.1.3. Predicates 52
1.1.4. Atomization 53
1.1.5. Positional access 53
1.1.6. The context item: 53
1.2. Combining node sequences 54
1.3. Abbreviations 54
1.4. XPath 2.0 functions by categories 54
6. XQuery 57
1. Basics of XQuery 57
2. Dynamic constructors 58
3. Iteration: FLWOR 61
3.1. Ordering 62
3.2. Variables 63
3.3. Quantified Expressions 64
4. Functions 64
5. Modifying XML documents 66
7. Exercises 67
Bibliography 70

List of Figures

2.1. DOM architecture 14
2.2. DOM modules 15
4.1. The XDM type hierarchy 41
5.1. The XPath processing model 47
5.2. XPath axes 51

List of Tables

5.1. XPath Abbreviations 54

Colophon

The curriculum supported by the project Nr. TÁMOP-4.1.2.A/1-11/1-2011-0103.

Part I. Advanced Database Systems - Lecture Notes

Table of Contents

1. Introduction 4
2. Basics of XML 5
    1. Namespaces and reuseability 7
    2. Valid XML documents 9
        2.1. XML dialects: DTD and XML Schema 10
            2.1.1. The most important XML Schema elements 10
    3. Information content and processing strategies 12
        3.1. Language independent processing strategies: DOM and SAX 14
    4. The fundamentals of document design 16
        4.1. Descriptive- and data-oriented document structures 17
        4.2. Building Blocks: Attributes, Elements and Character Data 19
            4.2.1. The Differences Between Elements and Attributes 20
            4.2.2. Use attributes to identify elements 22
            4.2.3. Avoid using attributes when order is important 23
        4.3. Pitfalls 23
3. XML databases 29
    1. Native XML databases 30
        1.1. Schema-free native XML database collections 30
        1.2. Indexing 32
        1.3. Classification of XML databases based on their contents 33
        1.4. Usage of native XML databases 33
    2. Hybrid systems 34
        2.1. Problems of the fully XML based storage layers 35
        2.2. Relational repositories with XML wrapping layer 36
        2.3. Hybrid repositories 36
        2.4. Other considerations 38
4. XDM: the data model for XPath and XQuery 41
5. XPath 46
    1. Expressions 48
        1.1. Steps 49
            1.1.1. Axes 49
            1.1.2. XPath Node test 52
            1.1.3. Predicates 52
            1.1.4. Atomization 53
            1.1.5. Positional access 53
            1.1.6. The context item: 53
        1.2. Combining node sequences 54
        1.3. Abbreviations 54
        1.4. XPath 2.0 functions by categories 54
6. XQuery 57
    1. Basics of XQuery 57
    2. Dynamic constructors 58
    3. Iteration: FLWOR 61
        3.1. Ordering 62
        3.2. Variables 63
        3.3. Quantified Expressions 64
    4. Functions 64
    5. Modifying XML documents 66
7. Exercises 67
Bibliography 70

Chapter 1. Introduction

The history of the databases is closely related to the data models and database systems. Everyone knows the path to their formation, the CODASYL recommendation, the main stages of development, which actually followed through hierarchical network and relational way of mesh data models. Longer carry the '90s and the new millennium marks the achievements. Progress is slowed down, but not stopped. Upon receipt of the object-oriented approach, the next higher step was the inclusion of XML languages and technologies.

In 1999, Tim Berners-Lee – the founder of WWW - has presented his latest idea of the Semantic Web agenda. It brought with it the ubiquity of XML languages and simultaneously published the first databases based on XML documents. The XML database management systems are basically not or do not typically store data in XML structures (to avoid unnecessary waste of space due to the lengthy XML document). The situation here is similar to a typical ANSI database architecture: again, not the physical but the logical database structure has changed.

For querying XML documents, the W3C consortium created a working group and two recommendation: the XPath and the XQuery language were born, these are based on a common data model from version 2.0. The reason of proliferation of XML-based databases, is that today not the users but software's (B2B) are producing and processing automatically and semi-automatically messages / queries. In this picture, the XML is well suited as a tool for data exchange. XQuery and XPath-based solutions to meet such requirements is much better than the SQL-based systems, in which data must be transferred to the transformation process is costly.

The purpose of this book means description of this world and demonstration of the use of XML in databases. The book does not aim to solve the problems of the storage of XML documents, just concentrate on the application approach. Complex methods exist for storing XML documents, a number of experimental systems exist which are beyond the scope of this book.

However, review of the structural design of the different approaches and the advantages and disadvantages of a new XML document in the book must part, because this is the point where you should learn about the various opportunities offered by these techniques in order to be able to work effectively document structure. In addition, we present validation, because a database is not simply a collection of data, but also help formulate complex rules to monitor the constraints on the stored data. To do this, we will also use standard technologies.

Chapter 2. Basics of XML

The basics of the XML language is contained in many notes and books, so we will not cover in details here, we just turn the key points. Necessary to know the basic structure and the field of meta-modeling used to process XML documents. We need to know which center on XML technologies and feel to know that a given context is created, or rather with a combination of technologies built around XML into the language. These skills are now only a schematic overview will be described in what you start with , what are the objectives of the XML :

- be easy to use in (web)applications,
- support a wide range of applications,
- be compatible with SGML,
- be easy to write programs for processing XML documents,
- human -readable , structured documents are clear ,
- be easy to create XML documents .

Of course, these goals are not only blind to the world appeared , but many - are still existing - problems were brought to life. These include identified by the W3C (World Wide Web Consortium) problem areas , such as the rapidly growing content , and versus a content access methods , which are still more focused on the appearance rather than the content itself . This created the need for a new Web, which is capable of understanding the content , supporting efficient access to information. The same has been passed more than a decade, the Semantic Web idea to conception, unfortunately, still has not been a widely accepted technology. The opportunity lies hidden in it, but have to wait for the accomplishment. However, the rise of computers power of increasing growth, can open the way for a brute-force search length of survival.

Equally problematic content, the structure structural handling of documents. XML is here to provide substantial progress since the 80’s SGML (ISO) standard also provides a free grammar, but the implementation is complicated and expensive. Conversely, XML is considered for simplification of SGML and therefore 'less expensive' item.

Which in turn should not draw parallels to HTML. Even if it is so similar to an HTML and XML documents, there are huge differences between them. The HTML:

- concentrate on display,
- can not be expanded notation
- loose (non-standard) syntax treatment.

In contrast, XML:

- strict syntax verification,
- expandable,
- content-oriented notation.

After that, if you are a little broader attempt to interpret what is XML, the format is simple text document but include such meta-language family of standards and technology as well. Attention for a number of document formats to XML:

- de facto: DOCX, PDF, RDF, ...
- de jure: HTML, SGML, XML, ODF, ...

We can see that either text or structured documents can be created. While the former focuses on the presentation (word processing and presentation), while the structural focuses on content and access to information (for the transmission, and storage). Here is an example of a text format:

```
<section>
 <head>A major section</head>
 <subsection>
  <head>A cute little section</head>
  <paragraph>
   <para>
    <person>R. Buckminster Fuller</person>
                        once said,
    <quote>When people learned to do <emphasis>more</emphasis> with
     <emphasis>less</emphasis>, it was their lever to industrial success.
    </quote>
   </para>
  </paragraph>
 </subsection>
</section>
```

We can see in the example that the document has some kind of structuring, but where the content itself appears it shows no structuring anymore. In contrast, documents that are for data interchange always have a strict structure. This way they can be processed with a computer. (The example is a very stripped edition. In reality many extra pieces of information appear in each item's description.

XML, as a meta language ,can be used to describe languages which is determined by tags and structures (vocabulary and grammar). The most common examples are DTD (Document Type Definition) and XML Schema. These are discussed later.

XML, as a family of standards , is made up of basic standards by several standardization organizations. Let's see a few examples of these standards without attempting to be comprehensive:

- **ISO/IEC JTC 1** standards:
  - Standard Generalized Markup Language (SGML)
  - Document Schema Definition Languages (DSDL) - defining the DTD and XML Schema-t
  - HyperText Markup Language (HTML)
  - Open Document Format for Office Applications (ODT) v1.0
- Organization for the Advancement of Structured Information Standards (**OASIS**)
  - Business Centric-Methodology (BCM), base for SOA
  - Universal Description Discovery and Integration (UDDI)
  - Web Services Business Process Execution Language (WS-BPEL)
  - Security Assertion Markup Language (SAML) - to replace LDAP
  - DocBook - a format used to create this note
- World Wide Web Consortium (**W3C**)
  - Extensible Markup Language (XML) (1998, 2004, 2008)
  - Resource Description Framework (RDF) (2004)
  - XML Namespaces (1999, 2004, 2009)
  - XML Schema (2001, 2004, 2012)
  - DOM (1998, 2000, 2004, 2008)

- XML Path Language (XPath) (1999, 2007, 2010)
- XSL Transformations (XSLT) (1999, 2007, 2010)
- XHTML (2000, 2001, 2007, 2010)
- XQuery: An XML Query Language (2007, 2010)

XML, as a technology , gives tools and methods for solving tasks in a variety of areas. Technology, because it contains standards, products for making, processing and displaying structured documents. The land of uses can be:

- web presence (server and client side transformation)
- interchange (format, transformation) – e-Business
- text representation and processing
- document formats in Office Applications (OpenOffice, MS Office)
- Web 2.0
- technical document's language
- configuring software (ant, maven)
- defining user interfaces (XUL)
- Curriculum Vitae in the EU (Europass)
- …

Based on this list we can see that it points back to the previously seen properties of the XML, as a document format and as a metalanguage. Moreover, there are technologies to display and transform it, language bindings to make it available for programming languages (like JAXB for Java) and technologies to store and query it.

1. Namespaces and reuseability

Before we discuss the important factors for XML storage, we need to see what are the possibilities for reuse. XML solves this by introducing namespaces. This makes possible to utilize foreign tools and to avoid name conflicts which could became reality if we use multiple sources and some of them use the same naming convention for different terms. It can only be resolved by clearly stating which element belongs to which namespace.

Namespaces are **syntactic mechanism's**. It can be used to distinguish a name in different environments. It allows us to use elements of several markup languages in one file. Every namespace is **an infinite set of qualified names**, where the qualified name is a `<namespace, local_name>` pair, where the namespace name could be empty.

The XML standard uses the `xmlns` (XML Namespace) attribute to the introduce namespaces. This attribute associates a given namespaces to a specified name - as a prefix. So the defined elements in the namespace can be used after declaration with the help of the prefix.

```
<?xml version="1.0" encoding="utf-8"?>
<account:persons xmlns:account="http://example.org/schema">
 <account:person name="John Doe" age="23">
  <account:favourite number="4" dish="jacket potato" />
 </account:person>
 <account:person name="Gipsz Jakab" age="54">
  <account:favourite number="4" movie="Matrix" />
 </account:person>
 <account:person name="Hommer Tekla" age="41" />
</account:persons>
```

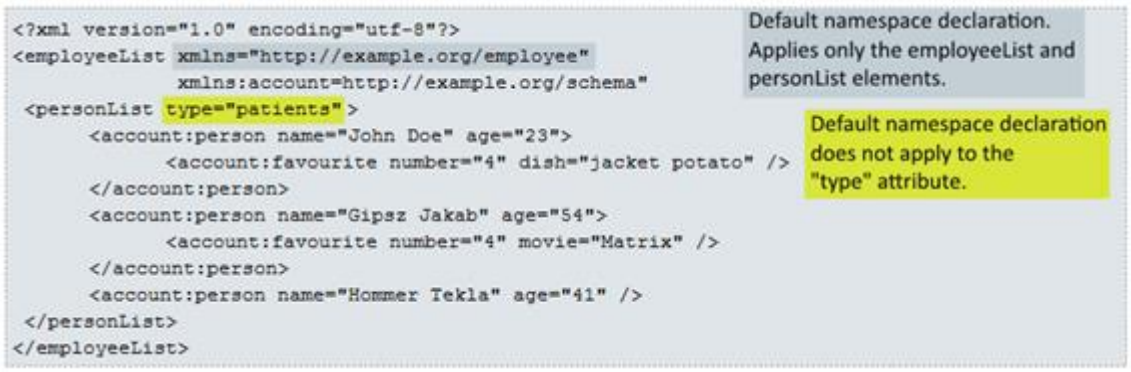
When using namespaces the readability is degraded and could be very difficult for humans to read it. One solution could be the usage of default namespaces (lack of the prefix) if the utilized parser (processor) allows it. In that case we do not have to use prefixes because the element and all children are placed into that namespace by default:

```
<account xmlns="http://example.org/schema">
```

Note

Adding only a default namespace declaration to an XML document in question eliminates the need to write a namespace prefix for each and every element, so it saves a lot of time.

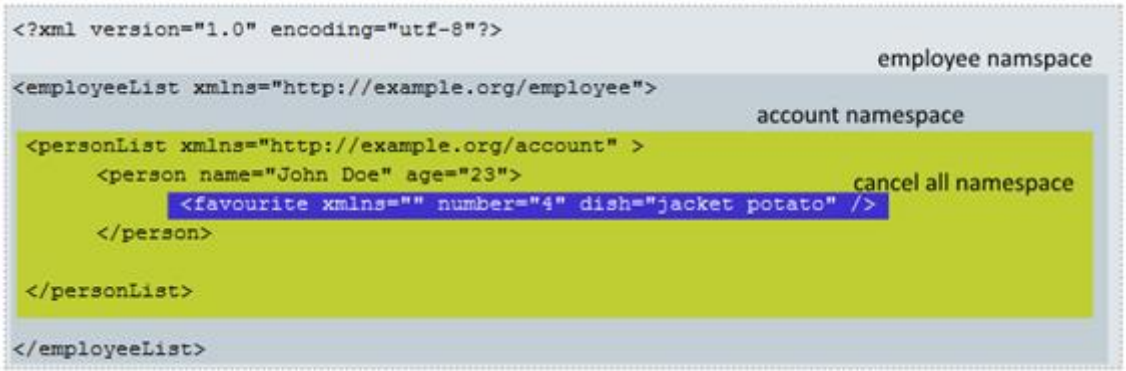
On the other hand, there are **drawbacks**. First, omitting the namespace prefix **makes it more difficult to understand which element belongs to which namespace**, and which namespace is applicable. In addition, programmers should remember that **when a default namespace is declared, the namespace is applied only to the element, and not to any attributes**!



Notation to cancel a default namespace

The `xmlns` attribute must specify an URI (Uniform Resource Identifier). Every element in the hierarchy must follow the specified structure introduced by the namespace. However, we can also delete a prefix like this:

```
<person xmlns:account="">
```



We can use several namespaces with different prefixes to avoid name collisions:

```
<?xml version="1.0" encoding="utf-8"?>
<account:persons xmlns:information="http://example.org/information"
xmlns:account="http://example.org/schema/person">
 <account:person name="John Doe" age="23">
  <account:favourite number="4" dish="jacket potato" />
  <account:favouriteWebPage>
   <account:address>http://www.w3.org</account:address>
  </account:favouriteWebPage>
  <information:address>
   <information:city>Debrecen</information:city>
   <information:street>Vasvari Pal</information:street>
```

```
  </information:address>
 </account:person>
</account:persons>
```

Both of them contains an `address` type, which is a simple type in the `account` namespace, but in the other case it is a complex type. If we use them without namespaces, the XML processor would not be able to decide whether that given type can occur at the given place or not, moreover whether it contains valid data or not.

2. Valid XML documents

We call a well-formatted XML document to a valid XML document if its logical structure and content is fit for the rules defined in the XML document (or in an external file attached to the XML document). These rules can be formulated with the help of:

- DTD
- XML Schema
- Relax NG.

The goal of schema languages is validation, they has to describe the structure of a given class of XML documents. The validation is the job of the XML parser, it checks whether the document suites to the description of the schema. The inputs are the document and the schema, the output is a validation report and an *optional***Post-Schema Validated Infoset (PSVI)** - which will be presented in the next chapter.

Schema languages give a toolkit to

- define names for the identification of the document elements
- control where the elements can appear in the document structure (forming the document model)
- define which elements are optional and which are recurrent
- assign default values to the attributes
- ...

Schema languages are similar to a firewall which protects the applications from the unexpected/uninterpretable formats and informations. An open firewall allows everything that is not forbidden (for example Schematron), or a closed firewall which forbid everything that is not permitted (like XML Schema).

A kind of classification of the schema languages

- Rule-based languages – for example Schematron
- Grammar-based – DTD, RELAX NG
- Object-oriented languages – for example XML Schema

But these schema languages are not created in a such simple way and without any limitations. It could be seen at the list of XML technology families that several ISO standards can be listed for them. These include the Document Schema Definition Languages (DSDL, ISO-19757) too. The object of the standard is to create an extensible frame for validation-related tasks. During the validation various aspects can be checked:

- structure: the structure of the elements and the attributes
- content: the content of the attributes and text nodes
- integrity: uniqueness test, links integrity
- business rules: for example the relationship between the net price, gross price and VAT or even as complicated things as spell check.

This could hardly be solved with the help of only one language, therefore a combined use of different schema languages could be required. Typical example is the embedding of the Schematron rules into an XML Schema document. An XML schema language is the formalization of the constraints, the description of the rules or the description of the structure's model. In many ways the schemas can be considered as a design tool.

2.1. XML dialects: DTD and XML Schema

Using schemas during the validation we can ensure that the content of the document complies with the expected rules and it is also easier to process it. Using different schemes we can validate differently. The XML 1.0 primer contains a tool to validate XML document's structure, called DTD. The DTD (Document Type Definition) is a toolkit that helps us to define which element and attribute structures are valid inside the document. Additionally we can give a default value to the attributes, define reusable content and metadata as well.

DTD use a solid, formal syntax which shows us exactly which elements can occur in the given type of document and which content and property can an element have. In DTD we can declare entities, that can be used in the instances of the document. A simple example of DTD is shown below:

```
<!ELEMENT <!ELEMENT book (author, title, subtitle, publisher, (price|sold )?, ISBN?, zip
code?, description?, keyword*  ) >
                            <!ELEMENT keyword ( #PCDATA ) >
                            <!ELEMENT description ( #PCDATA|introductory|body ) * >
```

Potentials provided by the DTD don't meet the requirements of today's XML processing applications. It is criticized mostly because of the followings:

- non-XML syntax which does not provide the usage of the general XML processing tools, such as the XML parsers or XML stylesheets,
- XML namespaces are not supported, which are unavoidable nowadays,
- data types, that are typically available in SQL and programming languages are not supported,
- there is no way of contextual declaration of the elements. (All elements can be declared only once.)

Many of the XML schema language have been created to correct these. The nowadays "living" and widely used schema languages are the W3C XML Schema, RELAX NG and Schematron. While the first is a W3C recommendation, the last two are ISO standards. The most commonly used is the XML Schema but it also has some disadvantages:

- The specification is very big which makes the implementation and understanding difficult.
- The XML-based syntax leads to talkativeness in the schema definition that makes the XSD reading and writing harder.

2.1.1. The most important XML Schema elements

The XML Schema definition has to be created in a separate file, to which the same well-formatted rules apply as to an XML document which are followed by some additions:

- the schema can contain only one root element, called `schema`,
- all elements and attributes used in the schema have to belong to the "`http://www.w3.org/2001/XMLSchema`" namespace, thus indicating that this is an XML Schema, most commonly used as `xsd` or `xs` prefix.

The most important attribute of the `schema` element is the `elementFormDefault` which specifies that the element must be qualified or it could be omitted. The same is true for the attributes (`attributeFormDefault`) as well. Based on the experience the best thing we can do, that we work through with qualified names, so we won't be surprised by any processors.

The building blocks of the XML Schema documents are the `elements`. The elements store the data and define the structure of the document. Elements can be defined in the XML Schema in the following way:

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

example - Defining a simple item

The `name` attribute is required, it must appear in the document. The `type` attribute determines the type of value that an element can be contain. There are predefined types which are almost similar to the ones met in the Java language (for example xs:integer, xs:string, xs:boolean), but we can define own types as well. We can further refine the available value of the element with the `default` and the `fixed` property. If we do not specify any value in the document, the application will use the default value. If the fixed is set, you can use only this value for the element.

You can specify cardinality, which tells the maximum number of times the element may occur in a given position. The `minOccurs` and `maxOccurs` attributes specify the minimum and the maximum occurrences. The default values for both are 1.

```
<xs:element name="address" type="xs:string" minOccurs="1" maxOccurs="unbounded">
```

Example – Enter the cardinality of an element.

It is possible to create your own type, so that your own type is derived from the subtypes.

```
<xsd:element name="EV">
 <xsd:simpleType>
  <xsd:restriction base="xsd:gYear">
   <xsd:minInclusive value="1954"/>
   <xsd:maxInclusive value="2003"/>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
```

The derivation is performed by the `restriction` keyword. So that we give the type to the base feature of the restriction element which is used for the derivation in order to get your own type. You can specify the constraints of your own type through the children of the `restriction` element. There are no limit for the constraints, it offers option for anything which is superseded by the possibility of regular expressions.

You can define the structure with **complex element types**. There are two groups of these: *simple content* and *complex content* based. Both of them may have attributes but only the complex one may obtain child element. You can see below a definition of a simple content based type:

```
<xs:complexType name="address">
 <xs:simpleContent>
  <xs:attribute name="city" type="xs:string" />
 </xs:simpleContent>
</xs:complexType>
```

Example – The definition of a simple content based type

Let's see the definition of a complex content based type:

```
<xs:complexType name="address">
 <xs:complexContent>
  <xs:sequence>
   <xs:element name="street" type="xs:string"/>
   <xs:element name="streetnumber" type="xs:integer"/>
  </xs:sequence>
 </xs:complexContent>
</xs:complexType>
```

Example – The definition of a complex content based type element

At the complex element type you have to define a compositor. This will specify that how to manage the child element. There are three of them:

- sequence
- choice

- all

The `sequence` method will recommend that the document has to show the children elements as they appear in the scheme. In case of `all` the order of the elements have no significance. If this option is `choice`, then only one child element can be shown from the listed ones.

The complex types are recyclable, if they are independent from any elements, they are like global definitions. If you define a new type, you have to give a name for it.

Finally we discuss the references, which make the management of redundant data easier because it lowers their numbers. The reference method can be performed by the “ID” and “IDREF” types in the scheme. The “ID” type is used to identify an element or attribute and you can refer to them with the “IDREF” type. These identifiers must be unique in the whole document. If you try to use an already existing identifier or refer to a non-existing one, the document will be invalid. The identifier is a “NCName” type, whose first character element must be a letter or underline, the rest is up to the programmer.

The following two types are part of a hospital system scheme, which represent a link between a sick-bed and the patient admissions.

```
<xsd:complexType name="Bed">
 <xsd:sequence />
 <xsd:attribute name="id" type="xsd:ID" use="required" />
 <xsd:attribute name="number" type="xsd:string" use="required" />
</xsd:complexType>
```

Example - Sick-bed type

```
<xsd:complexType name="AdmissionInformation">
 <xsd:sequence>
  <xsd:element name="admissionDate" type="xsd:dateTime" />
  <xsd:element name="endDate" type="xsd:dateTime" minOccurs="0" />
 </xsd:sequence>
 <xsd:attribute name="bedRef" type="xsd:IDREF" use="optional"/>
 <xsd:attribute name="medicalAttendantRef" type="xsd:IDREF" use="required" />
 <xsd:attribute name="end" type="xsd:boolean" use="optional" />
 <xsd:attribute name="recovered" type="xsd:boolean" use="optional" />
</xsd:complexType>
```

Example - Patient admissions type

The “Bed” type has two attributes: an identifier, “id”, and a bed number, “number”. The “AdmissionInformaton” is more complex and the most important that it has two “IDREF” attributes: one that refers to the patient’s bed, “bedRef”, and one, which refers to the doctor, “medicalAttendantRef”.

3. Information content and processing strategies

A well-formatted XML document seems to be only a set of characters, but if you try to learn the obtainable information, it is more valuable than at first look. To this end, the W3C created a recommendation, named `InfoSet`, which defined the abstract dataset to describe the well- formatted XML documents’ abstract information content.

An XML document has InfoSet, if:

- Well-formatted
- Corresponds to the namespace specification
- But the validity is not a requirement!

Basically, this is built up from information elements, which is the abstract description of the XML documents’ parts, which has named properties. It always contains at least one element, “`document`”, which is the root element. There are eleven different information elements; you can see the most important ones below:

- Document information elements (you can reach the other elements from this, directly or indirectly)

  - [children], [document element], [notations], [unparsed entities], [base URI], …
- Element
  - [namespace name], [local name], [prefix], [children], [attributes], …, [parent]
- Attribute
  - [namespace name], [local name], [prefix], [normalized value], [attribute type], [references], [owner element]
- Character
  - [character code], [element content whitespace], [parent]
- Namespace
  - [prefix], [namespace name]

From the example of the W3C's website we can easily understood how to imagine that abstract information set.

```
<?xml version="1.0"?>
<msg:message doc:date="19990421"
     xmlns:doc="http://doc.example.org/namespaces/doc"
     xmlns:msg="http://message.example.org/">Phone home!</msg:message>
```

This XML document has the following InfoSet:

- one `document` element
- one `element` element with a "http://message.example.org/" namespace with a local part `message` and a `msg` prefix.
- one `attribute` element with a "http://doc.example.org/namespaces/doc" namespace with a local part `date` and a `doc` prefix, containing the normalized value of 19990421
- three namespace element: http://www.w3.org/XML/1998/namespace, http://doc.example.org/namespaces/doc, http://message.example.org/namespaces
- two `attribute` element for the namespace attributes
- and 11 `character` element for the character data.

After all it is important to note what are NOT in the Infoset:

- the DTD content model
- the DTD name
- and everything included in the DTD
- the formatting characters outer from the document element
- trailing letter after the processing instructions
- the specification of the letters (reference or real)
- form of the empty elements
- spacing letter inside the opening tag
- the type of end-of-line characters (CR, CR-LF or LF)
- the order of the attributes

- the type of the apostrophes
- the CDATA section boundaries

This overview is an important piece of information about the Infoset because this is the base of the different parsing strategies. The most important implementations are inside the DOM and XPath data model. However, before reaching them we need to note that there are an extended version of the Infoset which could be obtained during the validation. The process itself is called InfoSet Augmentation, and the produced output (the augmented InfoSet) document is called **Post-Schema-Validated InfoSet** (PSVI). It is important to highlight again that it produced during the validation and ONLY XML Schema could be used for it.

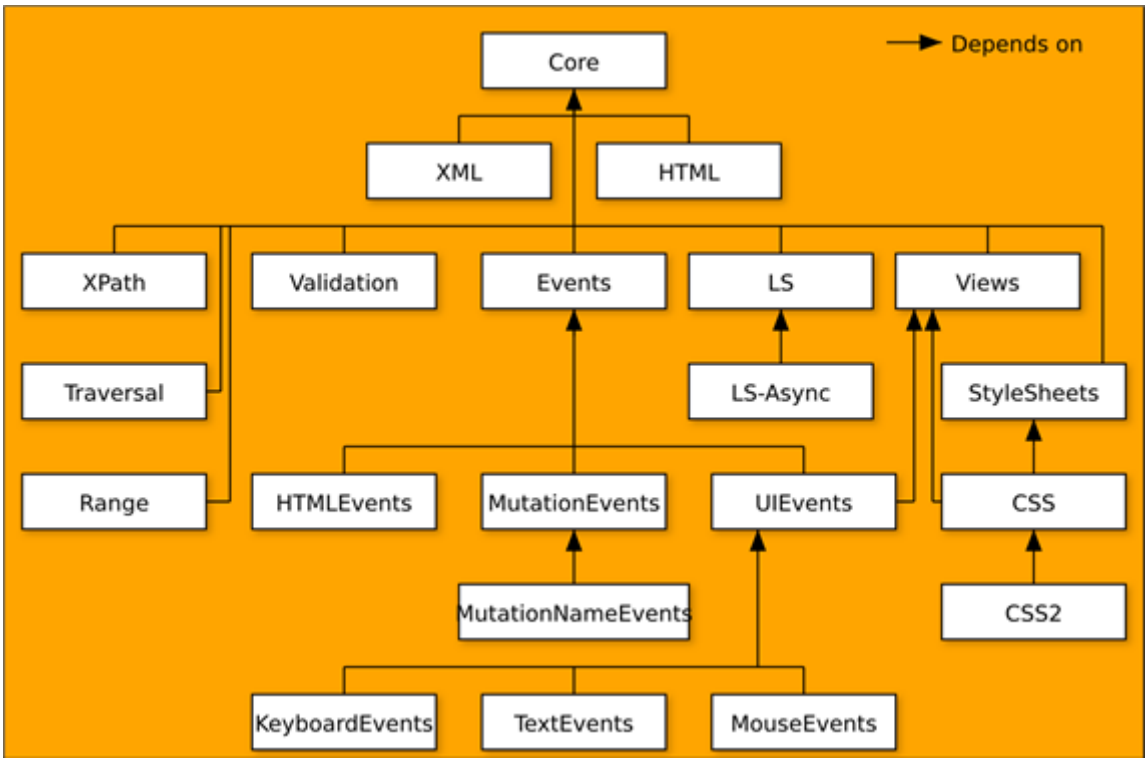
3.1. Language independent processing strategies: DOM and SAX

The Document Object Model (DOM) is a platform- and language independent standard application programming interface which models the HTML, XHTML, XML and other, closely related formats' structure and their interaction with their objects. The DOM is a system of objects that are related through a parent-child relationship. It contains the document's content, as well as every other part of the document, and changes here will also change the appearance of the page in the browser.

It's creation was motivated by the browser war that happened during the years of 1990, which eventually get concluded by the born of the W3C standardization processes. After the first draft of DOM appeared in 1998, two more appeared in 2000 and 2004. Currently the 2004 recommendation is used, since 2012, version 4 is being created. It's also worth mentioning that server-side interface events was still not a recommendation in 2012, but it is still very close to it.

DOM architecture can be visualized according to this chart:

Figure 2.1. DOM architecture

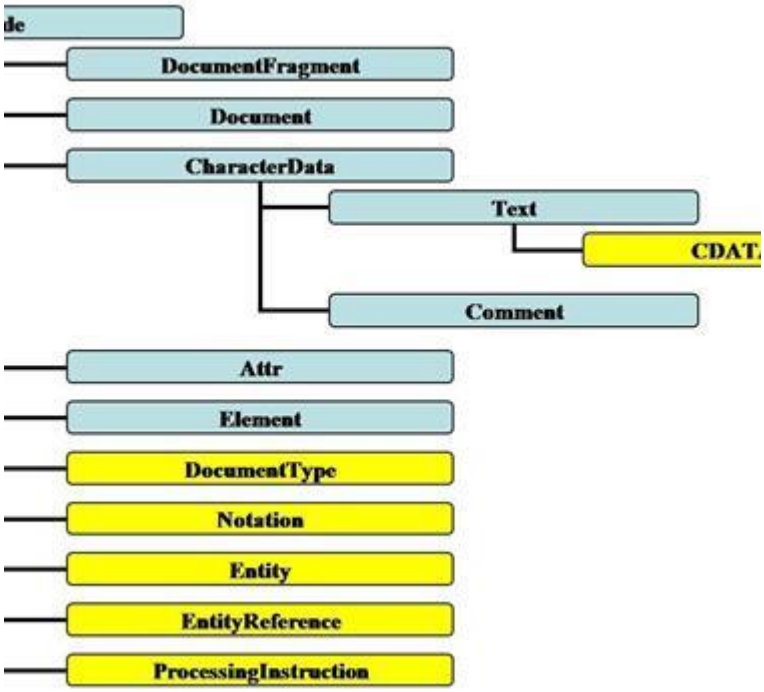


The DOM most important specifics:

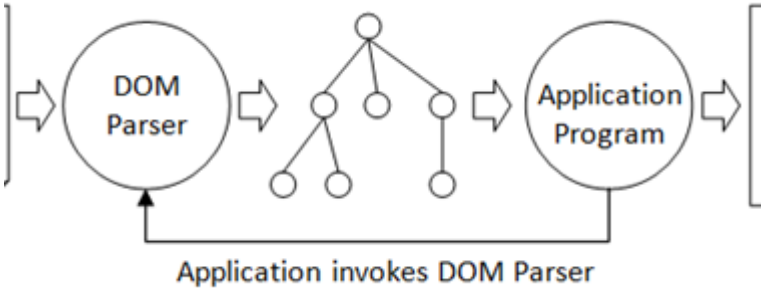
- -the document is logically managed as a tree (Node object hierarchy)
- -object model in the sense of the classical OO
  - documents (and it's other parts) are objects with identity, structure, behavior, and relations

- the DOM API provides two possibilities:
  - inheritance-based OO approach
  - a simple (flattened) view („everything is a Node”)
- Suitable for:
  - creating, building documents
  - structurally go through documents
  - element and content adding, modifying, deleting in documents.
- It’s built up from modules (DOM Core is blue, XML DOM is yellow).

Figure 2.2. DOM modules

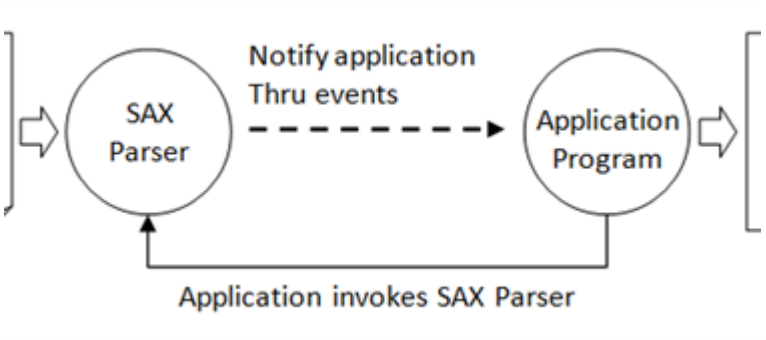


The usage of DOM looks like the following figure:



The Simple API for XML (SAX) is an event-driven interpreter, which means that it doesn’t create a representational model, like the DOM, that can be traversed in any way, rather the document processing is linear, similar to reading a text. The document is treated as a stream, during the interpreting, reaching certain points activates the corresponding event, to which the programmer can answer through implementing the API functions. SAX is not managed by the W3C, rather the Java language version is always the most up-to-date.

The usage of SAX looks like the following figure:



Compared to the DOM, an XML document does not have classes used for representation, instead the interpreter uses an interface, through which programs can access the data in the processed document with function calls. It consists of 4 basic types:

- string nodes
- element nodes
- processing instructions
- comments.

This provides a very fast and memory-efficient method for the SAX interpreter, because it requires significantly less memory compared to the earlier mentioned DOM interpreter. The biggest disadvantage is that its usage requires complex design and implementation, because we don't have already defined tools, through which we could easily represent and store the document's necessary and required parts. These must be designed and implemented by the developer.

The usage of the SAX API is recommended when we want to process large XML documents. In this case, we need less space during the interpreting itself, also local file usage is much more likely to be faster compared to the DOM API. If we want to use inner references or we need to access the document elements randomly, as such we're using almost the whole document, it's better not to use the SAX API because of it's complexity.

DOM and SAX quick review:

- DOM
  - tree based model (data is in nodes)
  - quick access
  - possibility for node adding/deleting
- SAX
  - methods called when reaching markup
  - better performance
  - less memory usage
  - suited for reading through the document, not modifying it.

4. The fundamentals of document design

The topic of designing documents is not the one that you can summarize in a short paragraph. As such, we will concentrate on three fundamental characteristics:

- **Descriptive- and data-oriented document structures** :

These two styles of documents give the vast majority of XML documents. In this chapter we will examine the characteristics of these two styles, and when to use either.

- **Building blocks: attributes, elements and character data** . Since there are many other features of an XML document, there are only a few that actually effects the design, legibility and interpretability, but the above three can do this.

- **Pitfalls** : Data modeling is such a huge topic, a separate book could be written about it. So, instead of covering everything, we will try to focus on the pitfalls that can be the cause of serious problems while using your document.

4.1. Descriptive- and data-oriented document structures

The XML documents are usually used for two types of data modeling. The descriptive document structure uses the XML content to supplement existing text-based data, similar to the way HTML uses tags for web pages. As for data-oriented document structures, the XML content itself is the important data. In this chapter we will examine these two styles and check out a few examples of when we should use each of them.

Use data-oriented document structures for modeling well-structured data

As previously mentioned, the data-oriented structure of the document is the one, wherein the XML content describes the data directly, in other words, the XML markup is the important data in the document. Recall the previously mentioned XML example with the hospital beds.

```
<?xml version="1.0" encoding="utf-8"?>
<account:persons xmlns:information="http://example.org/information"
xmlns:account="http://example.org/schema/person">
 <account:person>
  <name>John Doe</name>
  <age>23</age>
  <account:favourite>
   <number>4</number>
   <dish>jacked potato</dish>
  </account:favourite>
  <account:favouriteWebPage>
   <account:address>http://www.w3.org</account:address>
  </account:favouriteWebPage>
 </account:person>
</account:persons>
```

This document uses XML to specify the characteristics of a person. In this case, the text content (aka the interpreted character data or just character data, we use these terms interchangeably) in the document is meaningless without the XML tags. You could also write:

```
<?xml version="1.0" encoding="utf-8"?>
<account:persons xmlns:information="http://example.org/information"
xmlns:account="http://example.org/schema/person">
 <account:person name="John Doe" age="23">
  <account:favourite number="4" dish="jacked potato"/>
  <account:favouriteWebPage address="http://www.w3.org" />
 </account:person>
</account:persons>
```

This document records the same data set as the first, but instead of using character data to give the values, it uses attributes. In this case, there is absolutely no character data and it is clear that the XML content itself is the data in this document. If we were to remove the XML content of the document, apart from the whitespaces, it would be empty. This document style is not what we are recommending for use in practice, in this case it only serves the purpose of comparison.

A further thought is that we could of course continue to convert all the content to attributes of the element person. In this case it is obvious that the XML markup is the only useful data in the document. We can see that there are many simple ways of encoding the same data.

All three documents are clear examples of data-oriented documents, also they all use XML markups to describe the well-structured data. With the help of the data inside the documents, you can imagine that the data is one-to-

one mapped with the characteristics of our application, which makes the XML document the serialized version of our objects. This kind of serialization is very common in XML. We can also exploit the hierarchical nature of XML while describing more complex structures.

For this process, standard solutions have been integrated into many programming languages. Microsoft .NET Framework Common Library contains functionality that is suitable for serializing any class automatically, and deserializing any XML with the help of the class XmlSerializer in the package System.Xml.Serialization. The standard Java library does not contain similar classes, however there is an API which bears similar functionality. This API is part of the Java XML package, and it is named Java API for XML Binding (JAXB). It offers the possibility to map XML documents to Java classes back and forth, and makes it easy to validate Java objects stored in memory. The latter functionality of course requires the developer to provide DTD or XML Schemes.

Obviously, maintaining serialized data structure is not the only use of data-oriented documents. For Java Beans or any other data structures the mapping can be done easily as well. In addition, XML can be used to display other data that are not directly linked to a data structure. For example, the Apache Ant builder system also uses an XML file to describe processes, in this case, processes for creating software. Thus, a well-described process may be thought of as a well-structured data.

Overall, we can say that the data-oriented documents are great choices for any kind of well-structured data representation. Now, let us proceed and examine the descriptive documents that are less suitable for structured data.

Descriptive documents

The main difference between the descriptive document structures and data-oriented structures is that while the descriptive ones have been designed for user consumption, the data-oriented structures are usually made for applications to use. In contrast to data-oriented structures, the descriptive documents are usually texts that can be simply read by human beings, and those texts are just extended with XML tags.

The two essential characteristics that distinguishes a descriptive text from a data-oriented document are the following:

- *The content is not determined by the markup*:

  XML is generally not an integral part of the information provided by the document, but it extends the plain text in various ways.

- *The markups are less structured*:

  while the data-oriented documents are meant to describe a data set, and are strictly structured, the descriptive documents contain free-flowing text, which is meaningful content, such as any book or article. The data can be seen in these types of documents unstructured in the aspect that the markup in the document does not follow any strict or repetitive rule.

  For example, the number and order of tags in an HTML document, inside the <body> tag are infinitely flexible, and different from document to document.

Probably the most obvious examples of descriptive documents (although not strictly XML) are HTML web pages that use HTML markup tags in order to make their pages' more colorful. Although this information is important and definitely enhances the reading experience (for example, it specifies the location, size and of color images and texts on the page), it should be noted that the relevant information is independent of the HTML markup. Just as you use HTML tags in HTML documents to specify appearance, XML markup is used for things, such as to provide explanations for definitions in the text.

Take an example of how you can use descriptive document structure in practice. If you imagine that a hospital would use XML to represent minutes spent during surgeries, it would look something like this:

```
<operation>
    <preamble>
     The operation started at <time type="begin">09:30</time> on <date>2013-05-
30</date>.
     Involved persons: <surgeon>Dr. Al Bino</surgeon> and <surgeon>Dr. Bud
Light</surgeon>, assistent <assistent>Ella Vader</assistent>
```

```
   </preamble>
   <case>
    Dr Al Bino using a <tool>lancet</tool> made an incision on the <incisionPont>left
arm</incisionPont>.
   </case>
   ...
</operation>
```

This example document clearly shows that the important part is only the description of the surgery by a simple and clear way. The XML markups gives some useful information to the text (to the content) like the date of the surgery, the involved doctors and similar infos, but without these tags the content remains understandable (not like the previously mentioned data-oriented documents without the tags).

Descriptive documents could be useful in the following scenarios:

- **Displaying**

  the XHTML is a perfect example to indicate how could we use XML markups for descriptive documents to influence the appearance.

- **Indexing**

  applications could be achieved an effective highlight on descriptive documents by using XML markups to identify key elements inside. After all, it could be used for indexing - which could be done by a relational database or a specific software.

  The document describing surgeries are a good example for it. All the key elements are marked, so indexing could be done based on it.

- **Annotations**

  applications could use XML to append annotations to an existing document. It makes possible to the users to mark the text without modifying it.

4.2. Building Blocks: Attributes, Elements and Character Data

An XML document has many traits, but the three most fundamental that influence document design are elements, attributes and character data. They can be seen as the basic building blocks of XML and they are the keys for the way of good document design. (The secret of good quality hide in the knowledge of the proper use of them.)

Developers often face the decision whether to use attributes or elements for encoding data. There is no standard method or one best way to do this, often it's just a question of style. In some cases, however, (especially when we are handling large documents) this choice can make a huge difference in terms of the performance by the way of data handling by the program.

It is impossible to think of every data type and data structure that might be stored in XML format. That is why it is impossible to make a list of rules that can always tell the designer when to use attributes or elements. But if we understand the difference between these two, we will be able to make the right decisions based on the operation requirements of our application. That is why the following comparison focuses on the difference between elements and attributes, but we must not forget that character data also play a crucial role.

The "Rule" of Descriptive Documents

Unlike in data-oriented document structures, here we have a very easy rule that tells us when to use attributes or elements: **a descriptive text should be represented as character data**, every text should be part of an element, and every other information about the text should be stored in attributes.

We can easily comprehend this if we think about the purpose of the document markers: they provide extra meaning to the main text. Everything that adds new content (like the `<time>` in the previous hospital example) should be represented as an element, and everything that just describes the text without giving any new content should be represented as an attribute of that element. And finally, the content itself should be stored as character data within the element.

```
            Descriptive text became the content of an element, while the information
about it became attributes.
```

4.2.1. The Differences Between Elements and Attributes

Sometimes it is not a crucial decision whether to represent data as an element or as an attribute. However these two have very different behaviors, which can, in some cases, degrade the performance of our application. These are discussed in the following sections.

4.2.1.1. Elements Require More Time and Space than Attributes

Processing elements requires more time and storage space than attributes, if we are to represent the same data in both formats. This difference is not much if we process only one element or one attribute. However, on a larger scale or when document size is a major factor (e.g. if we send it through on a network with low bandwidth) our XML documents have to be as small as possible and using a lot of elements may prove to be a problem.

Question about Space

Elements always have to contain XML markers, therefore they will always take more space than attributes. Let's look at our patients' address as an example:

```
<information:address>
 <information:city>Debrecen</information:city>
 <information:street>Kis utca 15.</information:street>
</information:address>
```

This is a very simple XML encoding. If we want to make a C# or Java class according to this specification, we could use strings to store the data, and even the previously mentioned XML serialization would give us this result. This encoding requires 108 characters, not counting the spaces. We could, however, use attributes (along with the elements), and then we would be able to dispose of the end tags.

```
<information:address>
 <information:city v="Debrecen" />
 <information:street v="Kis utca 15." />
</information:address>
```

This takes only 94 characters, even if it makes reading the document more complicated (we know that v means value, but someone might not). In addition to the less space, it saves us some processing time too, because we don't have to deal with any free text contexts (character data). However, we should not use this method.

Let's look at the example once more, this time using only attributes:

```
<information:address city v="Debrecen" street v="Kis utca 15." />
```

This is the shortest possible encoding, and it takes only 69 characters, not counting the spaces (which saved us an extra 36%). This was just a simple example; this technique can be much more useful in case of larger documents where records could contain multiple packets of data. Both the saved storage space and the less time our application takes to read and process the document can be crucial.

Utilizing this method can make significant differences even on a smaller dataset. The documents using attributes are almost 40% smaller than the ones using elements and 35% smaller than the mixed ones. The mixing technique only saved about 6%.

This clearly shows that attributes are worth using instead of elements, if size is an issue. We also shouldn't forget that the size-problem not only appears when we are dealing with large documents, but also when our application uses and/or generates many smaller documents.

The storage space the elements need may also affect the memory, depending on the processing mode we are using. For example, DOM creates a new node for every element it finds in the data tree. This means that it has to build all parts of an element, which require additional memory space (and also processing time). Creating and storing attributes in a DOM tree requires much less, and some DOM parsers are able to optimize the application by not processing attributes until they are referenced. This leads to shorter processing time and reduced memory

usage because attributes are using much less space as the same data represented as an element object in the DOM processing.

Question about Processing Time

Elements not only require much more storage space, but also require much more processing time than attributes (based on how the basic DOM and SAX parsers work). Processing attributes using DOM produces significantly less load as they are not processed until they are referenced. Creating objects instead would mean unnecessary additional work (allocation and cration).

As we can later see, this can add up really fast and make DOM unusable, because having lots of elements can make the document a thousand times greater as its optimal size. If our favorite DOM implement is well documented, we may be able to check its source code and see how fast it handles attributes and elements, if not, we have only one thing to do - simply testing it.

Using SAX, the elements show a more obvious hindrance. All loaded elements in a document call two different methods: startElement() and endElement(). Also, if (like in our previous example) we use character data for storing values, it means calling an additional method too. That means, if our document has many unnecessary elements, calling two or more methods when we only need one makes the processing much slower. On the other hand, attributes do not require calling any methods: they are sorted into a structure by SAX (namely: Attributes) and then they become arguments of the startElement() method. This kind of initializing saves the computer much work, and saves us much processing time.

Based on our empirical experience, we can state that attribute style encoding in SAX (based on 10.000 elements) results only a minimal forehead like its counter part, the element only style - which is underlines our initial thoughts. However, the mixed style performs as the worst one, increasing processing time with the added elements and attributes.

By using DOM, we can estimate that the attribute only style could result big savings. The practice show nearly 50% forehead in the processing time. The mixed style is the worst one in this scenario too.

Using large amount of elements in SAX the results are nearly the same in all variations but the mixed style is still the worst one - approximately double the time requires for it to reach the same result as an attribute based version - highlighting that the mixed style never the best if performance is a key factor.

4.2.1.2. The elements are more flexible than the attributes

Attributes are quite limited in regards to what data they can display. An attribute can only contain character value. They are absolutely unsuitable of receiving any structured data, unequivocally intended to short strings. In contrast, elements specifically fit to receive structured data because they may contain embedded elements or character data as well.

However we can store structured data in attributes but at this point, we have to write all the code to interpret the string. In some cases, it may be acceptable, for example it is life-like to store a date as an attribute. The parser code is likely to analyze the string that contains the date. This is actually a pretty clever solution because we can enjoy the benefits of the attributes in contrast with the elements, as well as the time to evaluate this is minimal, so it saves us XML processing time.

(It should be noted that this solution works very well with dates, because their format is general enough by using them as strings, so they do not require any special knowledge in the analysis. However, if you overuse this technique it will ruin the XML representation, so use it with caution.)

In general terms, it is not advisable to be too creative when encoding structured data into a string. This is not the best use of XML, and is not a recommended practice. However, clearly shows the lack of flexibility that attributes represent. If we use the attributes in this way, then we are probably not making the most of the potential of XML. Attributes perform very well in storing relatively short and unstructured strings – and this is why you should use them for most of the cases. If we find ourselves in a situation that is requires to keep a relatively complicated structure as a text in your document, you might want to store it as a character data which has a very simple reason: performance.

In addition to that very long strings as an attribute value are stylistically undesirable, could cause problems in the performance as well. Because attribute values are only string instances, the processor must hold the whole

value in the memory all the time. However, the SAX way of it has only a minor problem with large character data because it wrapped into smaller chunks, which are then processed one at a time by the ContentHandler's characters() method. So you do not have to keep the whole string in the memory at the same time.

4.2.1.3. Character Data vs. Attributes

It is mostly a stylistic thing but there are some guidelines that will help you making the right choice. You should consider **using character data** when:

- *the data is very long*

  Attributes are not suitable to store very long values because the whole value should be kept in the memory at the same time.

- *large number of escaped XML characters are present*

  If we work with character data, we can use a CDATA section in order to avoid parsing. For example, using an attribute for a Boolean expression, we encounter something like this:

```
"a &lt; b &amp; b &gt; c "
```

  However, if we have use a CDATA section, we could have encoded the same string into a much more readable form:

```
 <![CDATA[ a < b && b > c ]]>
```

- *data is short, but the performance is important and we use SAX*

  As we have seen in the mixed-style documents, character data are requires considerably less time to be digested during processing than attributes in the case of SAX.

You should consider **using attributes** when:

- *data is short and unstructured*

  That's what the attributes have been created for but be careful because it can degrade performance, especially in the case of large documents (>100.000) using with SAX.

- *SAX is used and we want to see a simple processor code*

  It is much easier to process attribute data than character data. The attributes are available when we begin to parse the context of an element. Processing character data is not too complicated either but it requires some extra logic which can be a source of error - especially for novice SAX users.

4.2.2. Use attributes to identify elements

In many cases, the data has an attribute (or a set of attributes) which serves to clearly distinguish from other similar types of data. For example, an ISBN number, or a combination of an author and the title of book can be used to identify book objects clearly.

Using attributes instead of elements simplifies the process of parsing in certain circumstances. For example, if there is no default constructor of the class that we reconstruct then you can use the keys as an attribute to simplify the creation of the object. Simply, because the data needed for object creation are located in the opening tag of the element and promptly available (thinking about a SAX processors). The code remain clean as well, because the method will not over helmed with different types of elements, so we can always know where we are and what we are processing currently.

In contrast, if keys are stored as elements, then it will be difficult to track its processing code. It became complex because continuously requires an extra examination that is just for determining the type of the element and tracking the information needed to proceed. This could result in a more complex and confusing code.

Similar situation arrives when we would like to validate the document's content. Sometimes, we want to perform a quick check on the document before going into deeper processing. This is especially useful for stratified,

distributed systems: a small amount of validation performed at the beginning to save unnecessary traffic in the lower layers or in private networks (like a procurer system checks the format and number of your credit card before your order is forwarded to the executor).

In these situations it is a huge help, if the identifier or key information available as an attribute and not as an element. The processing code can greatly simplify the process and the performance will also be much better because most of the content can be completely suppressed in the document, and the process can be completed quickly.

Ultimately, using attributes against elements is a stylistic issue because the same result can be achieved using either approach. However, there are situations when attributes has a clear advantages in identifying data. Shortly, when specifying document structure make sure you know which category it belongs to.

4.2.3. Avoid using attributes when order is important

In contrast with the elements, there is no fixed order on attributes. It doesn't matter how we specify the attributes in a schema or in a DTD because there is no constraint or rule that specifies the order that attributes should follow. Because XML does not require any attribute precedence, avoid them in such situations where the order of the values is important. It is better to use elements than attributes because this is the only way how we can enforce that order.

4.3. Pitfalls

The way, in which we modeling our data in XML, will affect virtually every aspect of how people and applications come into contact with these documents. It is therefore critical that the data, which is represented, make the interactions more efficient and hassle-free. So far talked about a very low level of modeling, dispute the fact which XML primitives (elements and attributes) are more appropriate for different situations. For the rest, we'll examine a bit higher level principals.

We should avoid planning for special platforms and/or processors

The best features of the XML are that it is portable and platform-independent. This makes it a great tool to share data in a heterogeneous environment. When documents are used to share data or communicate with an external application, the planning of the document becomes extremely important.

As a designer we have to take a lot of factors into consideration. For example: the price of changing the structure of the data. After released and written codes were made to this structure, changing it is almost forbidden. In fact, once we presented the document structure for the outside world, basically we lose every control over it. It’s also recommended to think about how the documents, which are appropriate for our structure, will be processed.

It’s never a good idea to prepare a document structure for a version of some processor. I don’t mean that we should design document, which adapts to a processing technology (like DOM, SAX or pull-processors), rather I mean we have made plans for a certain version, like if we would adjust our JAVA code to a specific VM. The processors are evolving, so there is a little point to optimize a certain implementation.

As a result, the best thing we can do as a document designer is to plan documents to meet the needs of those what it will serve. **The structure of the document is a contract and nothing more, not an interface or an implementation.** Therefore, it is paramount importance to understand the usage of those documents, which follow this structure.

Following the database approach, consider the case when, let’s say, we use massive or just line-oriented database interface. If our API will affect hundreds or thousands of rows in the database, then probably we would design it to use collections or arrays as parameters, because one database query which affects thousands of rows is considerably faster than loads of query which run on only one row. It’s not necessary to know SQL Server or Oracle to make a good decision in this situation.

Also when we are planning a document structure, it much more deserves to focus on the extensive use of technology rather than worry about the special characteristics of a certain implementation. It’s much more important to understand the general difference between the elements and attributes than to know whether a process implementation is optimized to work with a large number of attributes or not. To continue the example

above, if we design a document structure, what we know that it will contain thousands of entries, it's important to know whether the usage of the element would cause significant performance penalty in terms of disk usage or time, during which the document will be processed.

If the documents aren't made only for our application, then we should definitely make sure we plan the structure of the document that it does not depend on any platform specific or processor specific implementations characteristic.

The underlying data model isn't usually the best choice for XML

XML is often used in situations where it isn't the permanent presentation form of the described data. Think about for example the XML based RPC, which is bound to web services. In the case of XML RPC, the parameters of the remote procedure, as well as the return value (if it exists) are described in XML in a SOAP compliant document. The fact that we used XML to remote execution it can be clearly seen that this is not the primary representation of the data – this is just a secondary representation, which is only for conversation.

In situations like these, where the XML only does the intermediate coding of the data, it's often a good idea to rethink the data structure, instead of simply recycle the existing structure, like in the XML-based structure. Often, the base data model isn't the most optimal for the XML.

We can refer again to the world of databases, where the most persistent storage, like the relational databases, simple files, object databases or something similar wasn't designed with XML and its structure in mind. Each of them was created for special purposes; they have got their strengths and weaknesses in this point of view.

One of the most common uses of the XML is to describe data, which has permanent representation as a relational data in some kind of a RDBMS. Relational databases have a simple structure, which contains separate tables, what has connection points with each other that are only logical. These relationships are defined by the foreign keys. Using relational databases onto hierarchical data usually means that there is a data table, which represents the parent in a hierarchical relationship, and there is a separate table (or tables) representing the "children" in the relationship.

If we have to represents this table structure in an XML document, the result would be a useless document structure, which would provide needless complication and plus work for the processing task because it isn't use one of the greatest features of XML: the ability to represent hierarchical data. Flat document structure can be beautifully achieved which hasn't got anything to do with the true face of XML, like the following example:

```
<?xml version="1.0"?>
<planet name="earth">
   <record region="Africa" country="Mauritius" language="Bhojpuri" year="1983"
females="99467" males="97609"></record>
   <record region="Africa" country="Mauritius" language="French" year="1983"
females="19330" males="16888"></record>
   <record region="Europe" country="Finland" language="Czech" year="1985" females="42"
males="36"></record>
   <record region="Europe" country="Finland" language="Estonian" year="1985"
females="330" males="102"></record>
   <record region="Far East" country="Nepal" language="Limbu" year="1981"
females="65318" males="63916"></record>
   <record region="Far East" country="Nepal" language="Magar" year="1981"
females="107247" males="105434"></record>
   <record region="Middle East" country="Israel" language="Russian" year="1983"
females="56565" males="44500"></record>
   <record region="Middle East" country="Israel" language="Yiddish" year="1983"
females="101445" males="87775"></record>
   <record region="North America" country="Canada" language="English" year="1981"
females="7521960" males="7396495"></record>
   <record region="North America" country="Canada" language="French" year="1981"
females="3178190" males="3070905"></record>
   <record region="South America" country="Paraguay" language="Castellano" year="1982"
females="91431" males="75010"></record>
</planet>
```

If we were to take this structure and place it directly into an XML, we would get a document structure similar to the following::

```
<?xml version="1.0"?>
<planet name="earth">
 <region name='Africa">
  <countries>
   <country name="Mauritius">
    <languages>
     <language name="Bhojpuri" year="1983" females="99467" males="97609" />
     <language name="French" year="1983" females="19330" males="16888" />
    </languages>
   </country>
  </countries>
 </region>
 <region name="Europe">
  <countries>
   <country name="Finland">
    <languages>
     <language name="Czech" year="1985" females="42" males="36" />
     <language name="Estonian" year="1985" females="330" males="102" />
    </languages>
   </country>
  </countries>
 </region>
...
</planet>
```

Its a lot more natural XML-structure where even the relations of the original database can be observed. This document is much easier to read than the previous one because it utilises the internal hierarchical structure of XML in order to represent data in a more natural way. Structuring documents in this way can also make the processing code easier and more efficient for documents.

The most important thing to bear in mind when designing the structure of the document is that it should be based on the abstract model that best describes what we would like to code. Not in any way should it rely on the implementation of the model in some other technology (Java, C++, etc.) There might be cases where the software implementation of the model is very similar to its XML implementation, but this is very rarely the case. This type of analysis usually helps to create a more XML-friendly data model, which results in larger documents with simpler processing codes as well as better performance during the processing.

Avoid large documents

One of the side-effects of using XML is the lot of extra work that goes hand in hand with coding. To be honest, XML is not the most efficient descriptive mechanism. Each data of the document is associated with the appearance of some marker. As a general rule, it is advisable to keep the size of the document as small as possible for several reasons:

- although disk space is not costly, it is not advisable to waste it,
- each and every bit of the document is going through an XML processor, consequently, the longer the document, the longer this process takes,
- if the document is used as a means of communication such as a SOAP document for an XML-based web service, then the whole document needs to be sent out to the network.

Although the long names of elements and attributes can greatly affect the size of the document, this is not where we typically try to gain space from, when attempting to fit the size of the document into a reasonable range. It is recommended that one should make sure that the document remained easily readable. (It also must not be forgotten that these are the elements that will shrink to the smallest size in case of compression).

It is often effective to analyse the data getting into the document to avoid increasing the number of unnecessary bits. It is a common trend in XML coding to just send the whole for coding without actually reviewing it. We can think of the issue of *derived data* as an example. The same holds true in the case of *replication* as well. It is needless to transfer the whole database, it is enough to transfer the changes only. Mostly, *temporary data* also do not need coding. In Java, what is marked as transient will not automatically be serialized. Fields that store temporary states such as the cache or the interim states during a long-running computation are generally transient variables.

Of course, it is also not practical to shorten our document too much; it is always a balance to be kept between the current performance and future flexibility. We have to make our own decisions which observe the requirements of our own applications. With a little more attention to the data that are in our document compared to what is actually needed, we can make the size of our document smaller, which will make it easier to process and handle.

The last general rule is that we should avoid redundancy when building the structure of the document. This leads to the next issue.

Avoid the use of document structures overloaded with references

One of the most inconvenient points about writing an XML processing code occurs when the document structure to be processed contains a lot of references similar to the following:

```
<document>
 <anElement name="szulo" />
 <anElement name="gyerek" parent="szulo" />
</document>
```

This example illustrates the use of references in a very simple way. We must not forget it though that references are merely text-based and their use is only supported by very few language elements, and it is usually the task of the application code to process and handle them.

References can of course be very useful under certain conditions in XML. For example, the Ant build system uses XML to describe the processes, and we use references in an Ant build file to specify the dependencies of the target. Here is a quick example:

```
<project name="MyProject" default="dist" basedir=".">
 <description>
 simple example build file
 </description>
 <!-- set global properties for this build -->
 <property name="src" location="src"/>
 <property name="build" location="build"/>
 <property name="dist" location="dist"/>
 <target name="init">
 <!-- Create the time stamp -->
 <tstamp/>
 <!-- Create the build directory structure used by compile -->
 <mkdir dir="${build}"/>
 </target>
 <target name="compile" depends="init" description="compile the source " >
 <!-- Compile the java code from ${src} into ${build} -->
 <javac srcdir="${src}" destdir="${build}"/>
 </target>
 <target name="dist" depends="compile"
 description="generate the distribution" >
 <!-- Create the distribution directory -->
 <mkdir dir="${dist}/lib"/>
 <!-- Put everything in ${build} into the MyProject-${DSTAMP}.jar file -->
 <jar jarfile="${dist}/lib/MyProject-${DSTAMP}.jar" basedir="${build}"/>
 </target>
 <target name="clean" description="clean up" >
 <!-- Delete the ${build} and ${dist} directory trees -->
 <delete dir="${build}"/>
 <delete dir="${dist}"/>
 </target>
</project>
```

The processing of documents overloaded with references can be quite complicated, and depending on the size and structure of our document as well as the effort put into the references, the performance of our processing code may be damaged, and the code itself may become difficult to understand for others. The problem with the use of documents overloaded with references is that we have to release the references ourselves. (This holds true for DOM and SAX at least. XSLT, which is more advanced, has more robust mechanisms to release references).

However, the resolution of references is not an unbearably complicated technical problem. Usually the code writing is very simple, especially if the relationship is one-to-one or one-to-many. The real problem is that we

have to keep a considerably large amount of „states” in the memory for the resolution during the whole process. It’s not a big deal if the document is small, but as the document size increases, so does the state size and it can be a problem after a certain point. (Just think about that if we use DOM for document processing, than it will be a serious problem due to DOM requires to keep the whole document in the memory.)

Let’s look at an example. Let's say you are working on a document that uses a lot of references in a simple, non-recursive parent-child relationship. In the document the parents will be listed before than the children.

```
<document>
 <parent name="parent1" />
 <parent name="parent2" />
 ...
 <gyerek name="child1_1" parent="parent1" />
 <gyerek name="child2_1" parent="parent1" />
 ...
 <gyerek name="child1_2" parent="parent2" />
 <gyerek name="child2_2" parent="parent2" />
 ...
</document>
```

We need some information about the parents for processing children in order to successfully process the document. For instance, we have to check that every children have a reference for a valid parent (this is called as reference integrity). It is also possible that we need information about the parent’s attribute to process a child.

If the number of parents is relatively small, the extra place reserved for the parents in the memory will not crash our system. But it will complicate the code of a SAX parser due to we have to write a code which handles the data. Although the presence of a relatively large number of parents can significantly affect the processing performance of your code because the memory usage increases linearly with the number of elements in the parent document. Even, any collection used for the handling in the parsing code should be sized properly, or should be increased (vectors, list arrays) constantly which also consumes a lot of memory and processing time.

The problem of references can worsen if the referring elements come earlier than the referred ones. For instance, if we rewrite our document in such a way that children would come sooner than parents, the amount of data in the memory would be much higher. There will probably be at least as many children as parent, if not more. To successfully process the document in this case, we should keep the data of all children in the memory until we process their own parents. When we find a parent we want to process all of its children which also complicates the code for the processing.

Most of these reference problems could be avoided if our document uses one-to-one or one-to- many relations, taking advantage of the hierarchical nature of XML. The processing of hierarchical data against references may lead to a much better code and memory usage.

To continue our previous example, if we rewrite our parent-child document using hierarchical structure it would not only simplify but would led to a cleaner and more readable code:

```
<document>
 <parent name="parent1">
     <gyerek name="child1_1" />
     <gyerek name="child2_1" />
 </parent>
 <parent name="parent2">
     <gyerek name="child1_2" />
     <gyerek name="child2_2" />
 </parent>
 ...
</document>
```

If the document is hierarchical then the current piece of the document contains the parent of a specific child node. Therefore, in order to get the kids processed, the only parent we need to keep in mind can be found in that text. When a parent gets out of context, it is guaranteed that there will not be more children related to this parent, so it is not needed to retain information about this parent. Another great advantage of this structure that it is much more readable by humans.

In the case of Ant, this cannot be exploited because it models such data which are in many-to-many relation and a specific compilation target can be dependent on any number of other targets and any target can be a

dependency of any other target. In such situations, certainly not worth the hierarchically encoded data, as there is no general pattern to the relationship of targets.

When documents are being planned, strive to use hierarchy instead of reference. If we have to model such a data which requires references, try to imagine the parsing code in our head so at least the memory usage will be predictable during the processing time.

To sum it all, we need to find answers to the following questions:

- Element or attribute?
  - frequent data change
  - small, simple data with rare modification
- Hierarchy or reference?
  - (sub)structures
  - containing more lines
  - multiple occurrence
- Stylistic choices
  - readability
  - constructing/parsing issues

During the creation of our own XML applications follow these steps:

1. Define elements
2. Look for key elements
3. Find relations between elements

Chapter 3. XML databases

Let’s start with an apparently simple question:

```
Is an XML document a database by itself?
```

Just like from the previous chapters, it has become clear that an XML document consists of data as well as metadata. The collection of data is the most basic definition for databases. This collection contains information, more specifically data, like the bookkeeping records of a firm, or even worldwide demographic data. Beyond this, the basic database structure describes what kind of data can be found in it.

What is the structure descriptive data in a database? The answer is, of course, the database's metadata. That metadata of the database is the data from that part of the database which describes the information held in the database. These amount of metadata makes possible the retrieval of data, just like when we know during the reading of the customer or demographic data, that we are reading just the names of the customers or the data about all citizens who live on a particular part of the World. Without metadata the database's data become a holy shit.

The semantics of the data can obviously be implemented into an application. These apps can be programmed to read specific items from a disk, like the name of a client. Contrarily, it is much more simpler if we also hold this information in the database itself. The structure in result is significantly simpler for the programmers and for the applications as well, and also more effective.

All in all, it can be seen from the previous explanations that **XML documents also forming databases by themselves**!

The question rises:

```
"Is this an information that held in the XML document but in the XSD (XML Schema
Definition)?"
The answer is NOT of course.
```

An XML document by itself is also a database – exactly what it is. An XSD or a DTD only needed when we want to check the structure of the XML document whether fulfills or not the appropriate implementation created by the developer.

The XML language is a self-describing language because it contains the data and the necessary metadata for describing its structure in one place. This metadata can be found in the name of the attributes and elements, continued in the hierarchy of the XML document itself. The metadata is descriptive in both scalar and structural meaning. XML metadata are double-natured:

- The elements and attributes of the XML document describe the property of the data. They are equivalent to the tables and fields of the relational database.
- An accordingly structured hierarchical XML document describes the relations among different types of data in the database. This is more or less equivalent to the relation between the tables in a relational database, but it is much more like the structure built among the classes in the object oriented world.

If it has became confusing again by mixing up the XSD and DTD way with the relational approach, it is understandable but it is important to point out that the metadata of an XML document are the elements (tables), attributes (fields) and the hierarchy (relations between tables) that come with an XML document – these are the metadata. An XML document does not need a relational table structure to define itself – because it is already a database by itself.

All in all, the usage of XML has a huge benefit which means that an XML document already has a built-in database structure. XML is self-describing in that interpretation that contains both data and metadata. The metadata are also a descriptive in both scalar (as the fields and values) and structural way (like hierarchy). XML is portable because it is clearly interpretable.

On the other hand, the usage of XML as a database has a lot of disadvantages as well. There can be overlaps in XML documents and because of that it can become wordy. In reality, an XML document contains a huge amount of information in one place and often in one file. This property can lead to enormous processing times

for huge amount of data. Essentially, the connection to XML documents – in its basic form – requires full reading, interpretation and of course in cases of potentially big text files and huge resources.

Of course, it would be pointless to compare this size with the power of commercial (typically an Oracle or SQL Server) databases where the amount of data can be measured in terabytes. Right now you can't find such system on the market which could achieve a size more than a few gigabytes, but even if it could, its processing time would be out of sanity.

1. Native XML databases

By definition, an NXD (Native XML Database) can be an XML document and also an XML datatype as well. An XML datatype is a special data type of relational databases. This means, an NXD can be practically anything that can store XML data - like an XML document. So an XML document represented in the browser is also an NXD. Besides this, the usage of XML datatypes in relational databases like Oracle or SQL Server can enable to reach the properties of the NXD. Essentially, all we need to describe a database to be an NXD or at least NXD capable is to store an XML document like and XML document. This way we can use the relational, object (-relational) or hierarchical databases as well.

The native XML expression actually means that the database is part of the XML. In other words, as we have put down earlier and repeated in the text, the XML document contains both data and metadata. The data is data and the metadata can be found in the structure, or at least bounds some meaning to the data. The enhancement of the metadata can be seen in the example below:

```
<?xml version="1.0" encoding="utf-8"?>
<account:persons xmlns:information="http://example.org/information"
xmlns:account="http://example.org/schema/person">
 <account:person name="Teszt Elek" age="23">
  <account:favourite number="4" dish="Sultkrumpli"/>
  <account:favouriteWebPage>
   <account:address>http://www.w3.org</account:address>
  </account:favouriteWebPage>
  <information:address>
   <information:city>Debrecen</information:city>
   <information:street>Kis utca</information:street>
  </information:address>
 </account:person>
</account:persons>
```

1.1. Schema-free native XML database collections

An NXD often contains and stores more than one XML document like collections of XML fragments. An NXD can contain XML documents from different types where the documents are about different topics containing independent data. For example, one collection consists of the data of a firm's customers while the other collection contains demographic data of the countries from the Earth. In addition, the XML fragments' structure in each collection can differ from the others. Every simple fragment in a collection can have different structure from the other fragment in the given collection. This result structural and scheme independent XML data.

The collections, if validation required for the XML data, can have poses a relation to schemes. Scheme independent XML makes development spectacularly flexible, faster and easier. However, the flexibility comes with low data integrity and with the risk of faults among the data inside the database. In reality, there are uncountable reasons why we should transform our collection of XML documents into an NXD for achieving better performance and storing methods.

Nevertheless, the misunderstanding and misinterpretation of this flexibility can cause problems. The repetition of too many data, too much or not enough structural complexity can lead to surprisingly many faults. The list of potential pitfalls is as long as the possible different variations of a topic made by an XML-like tool's flexibility. We can represent this through a simple example where we would like to describe a region:

```
<?xml version="1.0"?>
<database name="Africa">
 <collections>
  <collection name="Countries">
  </collection>
```

```
 </collections>
</database>
```

The example above did not contains any data but it consists more collections where every collection stands for a state. In this example the states belong to the African region. We can add new regions to achieve a multilevel database.

```
<?xml version="1.0"?>
<databases>
 <database name="Africa">
  <collections>
   <collection name="Countries">
   </collection>
  </collections>
 </database>
 <database name="Europe">
  <collections>
   <collection name="Countries">
   </collection>
  </collections>
 </database>
</databases>
```

In spite of this, we can go back to the way of getting more collection inside the database:

```
<?xml version="1.0"?>
<database>
 <collections>
  <collection name="Regions">
   <region name="Africa">
    <population>789548670</population>
   </region>
   <region name="Europe">
    <population>488674441</population>
   </region>
  </collection>
  <collection name="Countries">
   <country name="Finland" region="Europe">
    <population>5231372</population>
   </country>
   <country name="Germany" region="Europe">
    <population>82422299</population>
   </country>
  </collection>
 </collections>
</database>
```

Collections inside of a collection can also be arranged into real hierarchies just like the countries into the appropriate regions:

```
<?xml version="1.0"?>
<database>
 <collections>
  <collection name="Regions">
   <region name="Africa">
    <population>789548670</population>
   </region>
   <region name="Europe">
    <population>488674441</population>
    <country name="Finland" region="Europe">
     <population>5231372</population>
    </country>
    <country name="Germany" region="Europe">
     <population>82422299</population>
    </country>
   </region>
  </collection>
 </collections>
</database>
```

When continuing this example we can see how limitless the options for further evolution in these schema-free XML databases. However, they are carrying all the aforementioned risks which can easily turn into problems and faults. Of course, if we assign a scheme to an XML document we loose this flexibility but the other hand, the database system gain the power to enforce structural constraints.

1.2. Indexing

We can already be familiar with the techniques of indexing related to relational database systems, where one of our aims is to get the data with the least I/O processes. In case of relational tables the most commonly used indexing methods are binary trees, hash algorithms and bitmaps. When we talk about indexing, there is one important thing that we have to emphasize: there is a sequence in it. That is, when we are reading an index, we are reading the data in the sequence as the index has created them, independent from the order that can be found in the table or in the XML document. However XML document already contain an inner sequence, which is based on the XML structure, so the arranged index is not necessarily an advantage in case of XML data bases. XML data bases have different indexing methods. Probably every native XML data base applies different ways of indexing in their actual implementation.

The most widely used indexing method in case of XML documents in relational databases, even in case of XML data type, is separating those elements that need indexing. This way the index will contain a simple field about every record in the table. This single field contains a name about every region. Furthermore, the index will contain another pointer/indicator, which makes the linking of index and the table (or XML document) possible; it a physical address on I/O level. In other words, the database has to have a physical address about every (indexed) element in the XML document. This way the index contains the address of this pointer. The result is the following: when it finds a region in the index, the indicator related to the index will be given to a function, which finds the record in the table, or in an XML document on the basis of the hard disk address of the entrance record. The hard disk addresses get assigned to the table or the elements of the XML document when the whole pile of data or the index are created. The process and the steps of the process fully depend on the software used to reach the XML database, or which makes a simple XML document or a set of XML documents as one collection (These collections are obviously stored as XML).

Some questions may rise at this point, such as:

```
Why don’t we store XML documents in XML data type, if indexing the XML data type is
problematic, and it may be subordinated to the indexing of the relational data base?
```

or:

```
Why don’t we simple store data in relational tables and convert them to XML, if needed?
```

At first sight, we should examine the second question at first. Converting is not necessary, if the storage is in XML format. What’s more, we will have access to such useful XML tools like XPath and XQuery. Answering the first question: index is index. Some databases have more friendly indexing techniques than others. There is no reason, why XML indexing would be less effective than any other indexing, no matter what kind of relational database we talk about. Naturally, XML data can be stored in relational table. It’s the matter of choice. Both ways have their own advantages and disadvantages. In general we can say, that the bigger the database, the more sensible it is to store data in relational table – and not the XML stored in an XML data type. Unless we use XML data type collections. Keeping these in mind, we know 4 ways of indexing:

- **Structural index**

  Index of elements and attributes, and their relative location to the other elements and attributes in a simple XML document. It helps searching for elements.

- **Value index**

  If text and attributes are often searched items in an XML document, the best solution is if we index these, or the combination of text and attribute values.

- **Full text index**

  This one relates to the quest of a specific value in the collection of XML documents, and returns some portion of this collection. The value of the index is quite big, it contains several XML documents or some parts of a collection.

- **Context index**

  This one is a more general type of indexing, maybe a little bit old-fashioned, where many documents are indexed in a way, that the index contains some kind of a value, which identifies the XML document unambiguously. Indexes are stored in a so-called “side” table and this table gets an index. The solution is a quick indexed access in the collection of XML documents. This method is quite tiring. It’s better if the context of the XML document gets an index. It’s like we were using big and disordered XML files instead of smaller and better categorized fragments. All kind of manual categorization is very time and resource consuming in modern databases, simply based on the physical size of the information.

1.3. Classification of XML databases based on their contents

XML documents contain data, metadata and some kind of semantics in their inner hierarchical structure. Document centred documents are better for human application. However, they can be data-centred as well. These are more generic, and mainly used for processing by script languages.

Document-centred XML

A document-centred XML, which is also good for human consumption, is hardly interpretable for a computer, if it is possible at all. The document-centred XML is a document, which is normally handwriting – as a Word document, PDF file, or something similar to these notes. These kind of XML documents are usually stored as wholes and they are normally not understandable by programming methods, or by an XML element's content. Sometimes these documents are indexed for index searching, like the library of technical papers. Contrarily, there are some databases that contain technical data, which mixes data and document centrality. For example in the database of a library, which stores technical data of several years of the past, it is worth to categorize data according to authors, subject, time of publishing, or other kind of descriptive information. The content of these documents could be indexed, so that it would give general information about the subject of the given text.

A special type of document-centred native XML is called Content Management System (CMS). Content management systems permit some kind of control and management through XML data written by human beings, which are stored as XML types in a native XML database.

Data-centred XML

A data-centred XML document uses XML data in their simplest way for data transmission among computers. In reality, XML often contains the mixture of data-centric and document-centric features. The part, which has been made by humans and which can be interpreted for human beings belongs to the document-centred features. Something, that is generic, and available for a programme as it is reproducible, belongs to the data-centric part.

Maybe the best examples for data-centric documents can be found on websites, like Amazon or Ebay. These pages contain a mass of information, with the widest range of diversity. For example a book on the Amazon website, or better say every feature of it, such as ISBN number, author, or other kind of data are data-centric. On the other hand, many books on Amazon are available in PDF format. PDF files are document-centric, specific to a given book, and can be programmed correlated to the PDF document, in contrast with the content of the PDF.

1.4. Usage of native XML databases

It is important to go through the different aspect of the usage of native XML databases

- **Storage in native XML documents:**: XML documents can be stored as character data in databases, as binary object, or some kind of XML data type. Some relational database make it possible to store more than 4000 characters as a string. As the length of XML documents cannot be predicted, this is usually not enough. Binary objects can be simple elements stored binary (BLOB), or special binary elements, big text documents (CLOBS). The size of CLOBS is usually physically limited (in most cases 4 GB) in contrast with the strings. What is more, in contrast with long strings, or BLOBs, CLOBS usually make some kind of textual search or pattern-matching possible. Although we have to say that no CLOB based pattern-matching can be compared to the real XML handling technologies developed for XML documents. Some relational database makes it possible to store XML documents in their own format. As XML data type, with full XML capabilities. Native XML databases that have been created for this reason obviously have to have the functionality of XML handling besides just storing it.

- **Concurrency handling, locking, transaction handling**: Native XML databases in relational database as in form of XML data types only support locking on the level of the XML document. That is, the bigger the XML file, the worse the concurrency handling will be and the multi-user aspects as well. The node-level is also an option in XML documents, but imagine the implementation of that. Locking on the node-level definitely requires validation and force a schema too – this way we have thrown XML flexibility away. We can only enhance it knowing that XML documents have hierarchical structure by nature, so any node-locking infers the locking of every parent-node as well. Locking on the node level is not a good solution with today's technology. The way we structure an XML file highly depends on the requirements of the application. The bigger XML documents we store, the less suitable the database will be in a multi-user environment. A simple XML document, as native XML database is a single user environment in most of the cases

- **Reading native XML databases**: reading data of native XML databases, or XML data types requires special tools. These tools are the XPath and XQuery.

- **Changing the content of a native XML database**: several tools and functions are available to change the content of an XML document. As for the standards, XQuery makes updates unambiguously possible (despite it is called query, similarly to SQL).

2. Hybrid systems

As we have seen XML is perfect for representing structured data, it can be used to store and archive data, similarly to a relational database. The inner structure of the XML makes it possible to represent the sophisticated relations of various data in one unit. XML documents can be categorized into two groups: traditional SGML based descriptive documents, and the more and more popular data-centred documents.

Descriptive documents are text documents, in which markers help in the formatting, indexing, etc. (such as the collection of technical handbooks, web pages, or online newspapers and magazines). These kind of documents are typically split into articles, sections, or some similar separate and meaningful units. Data-centred XML documents rarely have free formed textual data, and they usually contain similar data structures, such as units that consist of fields (or columns, rows).

Let's see an example of descriptive document structure. We suppose, that surgeons make copies of cards in a handheld device. We have to create a system, where these cards can be stored for future reference, and they can also be shared with other systems (like the billing system of a hospital). At this example, we can use XML to enter the cards on the devices, but we would surely use it to transfer them to the storage place (where storage might be in form of XML, relational database, or a combination of these). It does not matter how the cards are actually stored, we would probably use XML to export data into other subsystems (billing, medical report, HR, etc…) This is what an example card looks like:

```
<transcript>
 <physican id="MD-123456" />
 <patient id="035-419-876" />
 <procedure>Making a 2 cm long cut on the <location>left upper arm</location> skin in
order to remove the nuclation with a
   <tool>lance</tool>.
 </procedure>
</transcript>
```

We have several options during storage:

- Does it make sense to store the XML document as a simple file in the system?

- Is it worth to divide it into separate sections or values, and store them in a relational database instead?

- And what if stored the data in a native XML database?

Let's change our example a little bit. We will use the same base structure with one difference, now we create invoice documents within a trading system. The transaction data (invoices) have to be stored for future viewing and in such a form, that it could be shared with other systems. The base structure will essentially stay the same, but this one results in a much more data-centric solution, because it does not contain textual parts.

The architecture of the base system stays the same in terms of information flow, so the question remains the same: how can we optimize the subsystems and the data storage architecture at these two documents?

2.1. Problems of the fully XML based storage layers

If an organization decides, that they integrate XML into their architecture on venture-level, XML developers often make the mistake of creating a storage layer, which is purely XML based (and it contains only single XML files).

In many cases it seem to be a sensible decision. After all, data enter and leave the system in XML form, and we use XML to represent data to the end users (using XML style sheets to show data in various forms of media). They why should we take this document apart and store the data in some other form, such as a relational database? Why should not we store everything in XML?

As it has already turned out, the scope of handling data as XML is quite limited – in most cases an XML representation will not be suitable to handle the processing procedures of data, what we would expect for XML documents.

Searching on the purely XML based storage layer is slow

Except the simplest cases, such a purely XML based storage layer will not be enough. These are only a few files in a folder, all of which corresponds to an XML document. **XML is not an indexed medium** - there is no way to clearly identify, whether an XML document contains relevant information in terms of a programming task. The only way to find, e.g. a specific invoice is to parse the documents and look for an element that suites the searching criteria.

For example in the mentioned invoice system, those customers who sign in can review their invoice history. To make it happen, the system has to be able to identify which XML document suits the given customer, and use these documents (such as creating an XSLT layer, which shows the customer the information that can be found in these documents). What helps the system to decide which document belongs to which customer? The system has to parse every document on the storage layer looking for the <customer> element with the "id" attribute value, which is the same as the currently signed in customer's id. So every invoice and customer information has to be checked, which is, in a huge system, where there are many customers, and many documents have to be checked which are irrelevant in terms of our current task, leads to a massive loss in processing time – even with a "streaming" parser like SAX.

Document aggregation is complicated

Similarly, there is no easy way of processing many XML documents at one time to obtain data – all of them has to be parsed separately, the most important information has to be gained and saved, and after that, if it is needed, the information has to be aggregated.

Going back to our former example, let's say, we have an invoicing system, where we would like to create a cumulative invoice, which contains all the invoices that are related to one single customer. To achieve this, our billing system has to develop the problem mentioned at our previous example and solve it one step further. Not only does it have to identify which invoice belongs to the given customer ID (and not paid), but it has to retrieve the information from these invoices that is needed to the given task (this case to create a cumulative invoice). After that, the whole of that has to be merged into one format (to the whole amount that has to be paid), this step is often referred to as aggregation processing, or document aggregation.

Manipulation of a document is in vain

Another worrying factor with plain XML storage is the way we process XML documents and gain certain data from them. Let's suppose, we would like to find out in our invoice example, how many of item YCK001-123456 has been sold last month. After we found out, which documents contain reference to YCK001-123456, we have to call the amount attribute in this document. It is not easy to do, if we parse XML documents directly, because the whole document has to be run through to gain the information. It is also true, if we run the data through the memory with the help of a SAX parser, or if we process the whole with DOM.

Solving the problem of indexing

The general solution of the problem is quite obvious – we have to give some kind of indexing to our XML storage mechanism, to enable us to quickly search for and aggregate the information found in the document. Several solutions exist to index XML data, such as: the previously shown XML databases, the relational solution, or object-oriented databases, or the hybrid combination of these.

As we have seen, native XML databases give index information to the XML storage. These might prove to be a good choice, if we want to keep the original XML appearance of the information, with the condition, that we only want to make a limited processing of the data, because currently these are too resource demanding – however, it will reduce with time and they can become a meaningful alternative. Let's have a look at the hybrid solutions to the problem.

2.2. Relational repositories with XML wrapping layer

Another approach of the topic of indexing is storing some part of the information, or the whole of it in a relational database, and using wrapping layers to transfer data between the relational and XML representation. This solution can easily be used in case of repositories, where can be easily broken down. At this approach, XML documents are not stored directly. Instead, the information is parsed, then stored in relational database, or deserialised from XML, depending on the circumstances (for example to operate transmitter or representational levels).

This approach is especially effective, if the information that enters the system is not already in form of XML, or it does not have to be ordered into XML form as part of the enterprise guidelines. Searching methods using this architecture could not be simpler. As every piece of information is stored in relational database, we have access to every mechanism which we would normally use for searching in a relational database: table indexing, key relations, etc…. In order to use the online invoice checking, we only have to write an SQL query that searches for invoices that have the same customer ID. The retrieved invoices can be transformed into XML for representation, whether using an inner XML output producer, or a general procedural code.

In contrast with the native XML databases, this approach makes data aggregation and handling easier. As is breaks down the information into pieces, before they are stored in the database, we can use the working SQL aggregation commands to retrieve data directly. In order to create a uniform invoice for example, we should only write one stored procedure that would download all the unpaid invoices from the customer's account. Another benefit of this is that this information can be serialized into a proper form – we are not bound to the original XML structure in any way. This makes our presentation layer much simpler as well. Of course, there are some, who live this as a disadvantage, but XML is not a wonder substance, and for many databases using relational storage model might be a better solution, than the XML tree structure.

A disadvantage of this relational-wrapper approach that the serialization and deserialization steps of the XML take time. If a given system highly depends on XML data format (for example a specific XML standard is used to transfer data), than the purely relational approach will take much processing time by compiling and decompiling XML documents. Another problem might be the knowledge of programming – instead of XML and XPath the developer has to be familiar with RDBMS and SQL as well. What is more, *in case of XML documents with descriptive style it is very hard, or even impossible to store data in relational database in any usable way*.

The purely relational storage method (around an XML wrapper relational layer) might be a good solution, if the system is not too XML-centric, does not contain descriptive information, or it does not have to execute huge aggregation changes on the information.

2.3. Hybrid repositories

According to the third approach we assign index to a stored XML documents, that we store the index in a relational database. In reality, there are two different methods, both of them have their advantage.

Storing full XML documents and relational data

In case of the first type of hybrid repositories XML documents are saved exactly as they enter the system, in other words, any other structure that we create on the data will stay as in the XML representation stored in the repository. Besides storing the XML document, the system processes it, and creates a copy of the index information in a relational database. Depending on the requirements of the system, the indexing might be minimal (it indexes only one key, such as the customer ID at our invoice example), or extensive (every single

element, or attribute is a separately indexed part). The retrieved indexed information is stored in a relational database index, by the help of which we get access to the proper XML documents.

The deep analysis of the system decides the balance between minimal indexing and the much costly "index everything" approach. The requirements, which drive the system analysis might change in time, as the procedure of the process regulates which parts of the data have to be indexed. Thus, when requirements change, and further indexing is needed, the database can easily be rebuilt by changing an XML document and re-indexing data. This one is another example that XML might be useful at maintaining complex systems – it is usually much easier to change a textual configuration document, than the process code.

As at the relational model, searching at a hybrid repository is also easy – providing, that we index the information that we would like to search for. It is similar to the case, when we search for indexes at a relational database – we have to index the information that the system will probably need for the search. The indexed database gets a command to find the identifier of those XML documents, which contain our customer ID (as in our invoice example). Then it should use these data to reach the XML documents. It is important to point out, that this method makes avoiding the serialisation step possible, which would be unescapable at a simple relational method., so increasing the performance.

Aggregation and document manipulation also become much less complicated thanks to this approach. At a good design, the information that is often used for aggregation and document handling gets indexed – that is, the necessary information can be gained without opening the XML documents themselves. This kind of system analysis is very similar to what is used in designing the traditional RDBMS indexing. As at the purely relational approach, we can take these data and build our XML document, in order to match with our output requirements.

One disadvantage of this method is the extra storage place, which is taken over by the repository. Basically we store the information twice: once the original (maybe lengthy) XML format, and in the relational index. One drawback related to this is the subsidiary complexity of the database maintenance, any change done on the data, will have to be changed in the XML document and its relational index copy as well. This way making synchronization more difficult. Here we have to make the traditional compromise: accepting more complex update codes and leaving more storage place at the documents lead to faster processing and easier code retrieval. Whether it is a good choice at a given programming task, depends on the nature of the application, and the software/hardware boundaries of the system.

Storing XML document fragments and relational data

At the other form of hybrid repository every indexed content is in a relational database, as at our first, purely relational example. However, those parts of the document, which does not have to be indexed (because we rarely search or aggregate them) are stored by the system as XML documents instead of breaking them down into single elements or attributes.

Let's take our invoices again, which appear in an online invoice tracking system. According to the requirements of the system, if it does not have to search each invoice item, we might store the invoices in a structure that looks like this:

```
Customer table   Invoice table        Items table
--------------   -------------        -------------
custIdinvoiceIditemId
Name             date                  item
Address          custId                invoiceId
City
Zip
```

If we need certain information of an item (for example for presentational purposes), we simply ask for the XML fragment, which is stored in the Items table in correlating with a given invoice.

This method, as the first hybrid method also makes searching in index fields easier. We can use traditional relational database mechanisms to get to the indexed fields, and to identify the documents that meet the requirements. Similarly, this solution also makes aggregation and documents handling easier – we shall simply design the indexing of data often used at aggregation and document handling, this way we will not need an aggregation code.

Comparing hybrid approaches

So what is the difference between the two hybrid approaches? The first one needs more storage place (as the data are repeated from the original XML document in the relational indexing), but it does not need further processing time or code to recover the information in its original form.

In case of the second hybrid solution less storage place is needed, as only the non-indexed XML of the original document is stored, and there is no duplication because of the index saving. The huge disadvantage shows however, when we have to restore the original XML structure - it takes code and processing time to restore the XML document from the indexed parts and fragments.

> We shall use the hybrid solution to store XML, if we have to make indexed procedures, such as searching and aggregating information, but we also have to keep the XML format, in which the information first entered the system. Store the whole document as in the first solution, if keeping the storage place is less important, than the quick restore of the original XML documents. In other cases apply the (second) method, which is based on the XML fragments.

2.4. Other considerations

Please note that the industry standards are becoming more and more prevalent. If we choose an arbitrary division, there is a consortium that defines the XML structures, which are used for data exchange between systems and their description. In many cases, engineers design document repositories without the appropriate experience, which simply store XML as they receive them. This is especially true if a system was built on a principle which transmits the information in the same XML format between itself and other partners. There are several drawbacks of this respect.

Problem of original XML's storage

As we mentioned earlier, the problem with XML standards is that they fail to meet as a one to one mapping with our unique personal representation of business information, or any display requirements. As standards evolve, they may meet individual business requirements and this section will be void, but today it is true.

It is important to emphasize that most XML documents that works well for one purpose, are completely useless for another. Relying on a single XML structure will definitely lead to increased code complexity and processing time, as it can cause major problems when it comes to maintain or update the software, which is creating and searching in it.

Consider again the billing system that we talked about earlier. We would like to store individual accounts, as the clients hand them in (as at a typical online transaction processing, also known as OLTP system), and let clients have the opportunity to access this information directly later. What is more, we would like to be able to do some analysis of the administered accounts – such as "how many invoices a client receives a year ", or "how many pieces were ordered of a given product in 2012," and so on... These systems are the most well-known as "online analysis processes, "or **OLAP** systems. We have already defined an XML format for our personal accounts, and we want to save them in a separate simple XML format. While this is effective in supporting the OLTP system side (since the display is the same as the original input), it is more than ideal from the OLAP perspective. If analysis is needed, the document storage must be adapted to be able to establish the analytic data, which will lead to slower and slower analytic performance as the database grows.

One solution to this problem is to create a database that can perform both OLTP and OLAP functions. However, it is less satisfactory for processing or analytical purposes – as one or the other almost always will be damaged. There is a different kind of solution you can choose, we can design a divided XML storage layer to directly support both the transactional and analytical requirements of the system.

Changing storage according to the requirements of the task

In this solution, the OLTP documents are processed as they enter the system. The information is read from these OLTP documents and is used to update OLAP documents. For example, if we would like to show the total amount of every single product, which have been ordered in our system every month. To do this, we could design a document with its own structure, then, as the invoices enter the system, they are able to be processed and the important information can be used to update the appropriate OLAP documents. Once we have the OLAP document, we can simply rely on a naming convention of files, or create a relational index to the documents.

The disadvantages are clear – more storage space is consumed, the information is repeated or extended processing time will be required to recover the appropriate OLAP information from each document when it arrives. The advantage of the OLAP processing is that it is much simpler and we have direct access to the XML documents without the need for the step of XML serialization.

Note

Consider using customized XML structures that directly support the XML data input and display requirements. This solution depends on three criteria (e.g., storage space, OLTP processing time or OLAP process time), which is the most critical in our personal architecture and business environment.

Difficulties in managing unstructured data

Another problem with the split of the XML documents to relational databases, is the question of narrative elements. Take our medical record example. It will be very difficult, if not impossible, to define a relational database structure that describes the number of possible variations of an inherently descriptive document. Although there are structured islands in the text, they can appear in so many different combinations and sequences so that we would not like to force them into a certain relational structure. Of course, we cannot just leave texts out just because they do not fit into a pre-defined structure, since this data is likely to be needed later during processing. It is clear that a different approach must be selected.

Dismantling a complete XML document into a relational database can only be meaningful if the document is data–centric without any unstructured text and the document meets those business functions that we want to perform on the data.

The best solution that is expected is to apply the first version of the hybrid modelling: the content of the XML document is always available, and indexing the information makes quick data capture easy, which we currently need at a given task.

In case of descriptive documents use hybrid storage mechanism in order to have access to the indexed items without splitting up the reference section, while retaining the original document. Otherwise (for example in case of data-centred documents) use hybrid storage with relational index and stored XML fragments. Simply leave out the unused parts of the XML that will not be needed in the processing later.

The problem of archiving

About archiving we mean the long-term storage of data where we no longer need direct access (as opposed to, say the online processing), they cannot be completely removed (due to business and / or legal considerations). A properly designed single XML document is able to give complete picture of a particular data set. As long as you have enough meta-information (in the form of good attributes and element names), the document can be self-contained and yet be interpretable outside the context for any processing software, data dictionary or schema. Moreover, since XML documents are in text format and can be compressed easily, making it possible to store a lot of documents and related meta-information in a single volume for future recovery. However, forcing data archiving in XML – especially if the source information is not XML (like data of a relational data base) – can result in difficulties when you want to access archived data.

The most obvious method by designing an archiving strategy is to ensure that the information is in retrievable in a usable format. In the past, it usually consisted of transhipment the relational data into a file, whose structure is the same as the original database so that the data can easily be loaded (parts thereof or bundles) into their original structure for future work. However, the advent of XML has created an opportunity to make archive documents that are self-sufficient, also known as self- descriptive. This is particularly relevant in today's rapidly changing technological world. No one knows that the tailor-made system that generated all the archived entries will exist in even twenty, ten, or even five years from now. With this in mind, the archiving strategy should be directed to design self-describing documents, which are able to provide all the information that you have access to.

Recovery of archived information

Documents archived in XML can fall into the same error as individual documents stored in XML have to face. These documents are stored in many files and still they should remain readily identifiable to a specific document or documents in certain restore operations.

Fortunately, we already know a way to store information, namely XML. You can use XML for information indexing, transformation and summarization as part of the archiving process. This way we reduce the number of documents to be processed when we access the archived material but improve the archive search and retrieval performance. When different ways to search XML documents were discussed all of them revolved around the theme of indexing– extracting specific information from the documents and using them to return to their original XML document. We can apply a similar strategy to the archives, but we will create our index in XML format.

Several solutions can be found compared to former ones. First, let's look at a way of simply selecting a few important data elements and attach a copy of them next to the correct file reference in the archive, for example, customer or patient's name, date of registration. When a request is received, information obtained under this index that helps to identify to which actual file has to be checked searching for the detailed data.

The alternative to this is a more advanced version of it so you can create documents that provide aggregate information. A typical case where a business processes often receives queries to retrieve aggregated information, and not just for the daily data. In this case we might create aggregated documents as well. These summary documents could take over the role of the archive documents, or (more probably) collaborate with them in an archive. This is similar to the indexing technique which has been previously analysed in this section – the aggregated XML documents efficiently form an upper level index for the sub–documents, thereby allowing the summarized information query without the need to delve into individual archives.

> When creating archives, design additional XML documents, which will support the extensive data in the archive. Both the aggregated documents (which are groups the data) and the index documents (which provide quick access to certain documents in the repository) prove to be a good choice in almost all cases of XML archive.

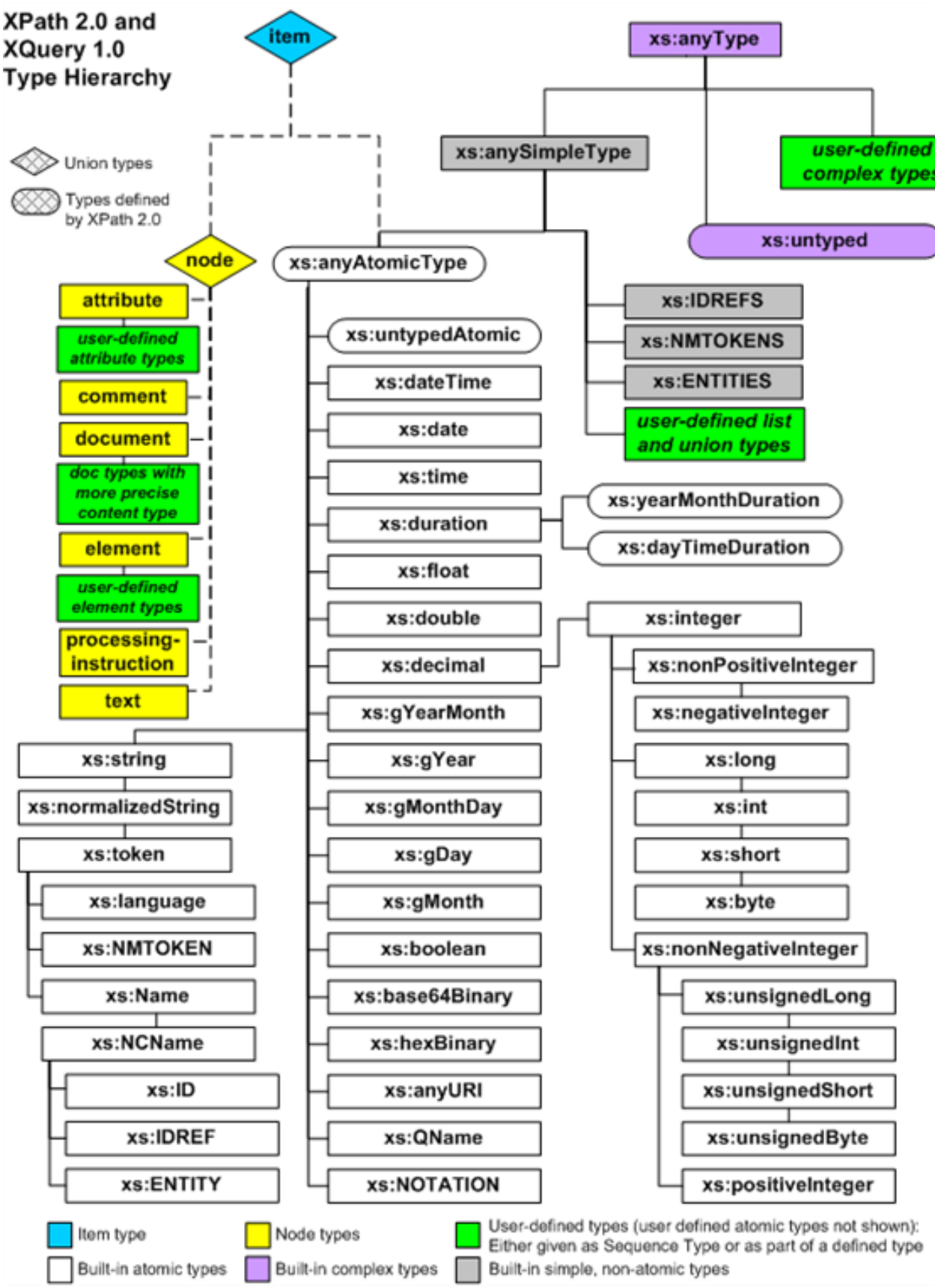
As a closure we have to talk about something that people rarely think about. The displaying code or style sheet for a given XML document. In some cases, there may be a business or legal restriction that the documents should be maintained in a particular format for years but it is not the bigger part. If we take advantage of the desirable decomposition of the semantic meaning and the representation at the XML database design layer, we can face a situation where we may no longer have the sufficient information to restore it to the original form without some code or style sheet. For this reason, it is strongly recommended to archive the style sheets and codes related to all XML documents.

Chapter 4. XDM: the data model for XPath and XQuery

The XQuery Data Model (XDM) is more precisely a common data model for XQuery 1.0 and XPath 2.0 (extended with XSLT 2.0). It became a W3C recommendation in December of 2010. thus replacing its previous version appeared in 2007. The primary goal of the XDM is to provide the required information set from an input XML for an XSLT or XQuery engine. Moreover, it provides the acceptable expressions for the XSLT, XQuery and XPath languages.

By definition, a language is closed for its data model if any of the language's expressions value remains inside the data model. The XSLT 2.0, XQuery 1.0 and the XPath 2.0 are all closed for the forthcoming data model which is based on the previously mentioned Infoset recommendation from 2004 extended with the type hierarchy used by XML Schema.

Figure 4.1. The XDM type hierarchy



Like the Infoset, the XDM defines all the information that could be collected from an XML document but does not impose any language binding or interface for reaching the data. The early 1.0 version was extended by the typed atomic values and the term of the sequences (replacing the set term).

A ***sequence*** is a collection of elements. In this new notation, an ***element*** could be a *node* or an *atomic value*.

```
Sequence type t (abstract)
t ::= empty-sequence() | item occ
occ ::= + | * | ? | ε
item ::= atomic | node | item()
node ::= element(name) | text() | node() | ...
name ::= * | QName
atomic ::= integer | string | double | ...
```

Important to know that during the parsing every sequence has a type always. In the following example the most general type comes first:

```
<x>foo</x> => element(x), item()
()         =>  empty-sequence(),integer*
("foo", "bar")   => string+, item()*
(<x/>, <y/>)     => element(*)+, node()*
```

Sequences cannot hold sequences inside. When we merge them the result always a flat sequence will be. Nested sequences cannot exists.

```
(0, (), (1, 2), (3)) ≡ (0, 1, 2, 3)
(0, 1) ≡ (1, 0)
```

Moreover, there is no difference between an element and a one sized sequence. An element is also a sequence! An other important difference form the 1.0 version - which originates for the set theory where one element could exist only one times in a set, that in 2.0 the sequences are allowing to include one element more than one times in the sequence and its identity is reserved.

An **atomic value** is derived from its domain where the atomic type could be a primitive type or its derivatives - created by restrictions. The root of the type hierarchy is the untypedAtomic, which is utilized when we wok with an unvalidated XML document. In that case, the runtime engine tries to cast it to the most specific type automatically (which could be an advantage but imposes risks too):

```
"42" + 1 ⇒ type error (compile time)
<x>42</x> + 1 ⇒ 43.0E0 (: double :)
<x>fortytwo</x> + 1 ⇒ conversion error (runtime)
```

Along with the sequences, the ***node identity*** play an important role in XDM. In every instance of the data model every node has its own identity. (Contrary with the atomic values whose identity is not definable.) The character '5' will mean the same five as a number everywhere in the document.

```
<x>foo</x> is <x>foo</x> ⇒ false()

<x>foo</x> = <x>foo</x> ⇒ true()
```

During the processing the most important term is the**document order**: its interpreted inside the nodes affected by a query or transformation and defines the order of the elements in the serialized version of the document. It is corresponding to the order in which the first character of the XML representation of each node occurs in the XML representation of the document after expansion of general entities. Further characteristics:

- the root node will be the first node.
- Element nodes occur before their children.
- Siblings are in order of the occurrence of their start-tag in the XML /not alphabetical or any other/
- Children and descendants are always before as siblings.
- The attribute nodes and namespace nodes of an element occur before the children of the element.
- The namespace nodes are defined to occur before the attribute nodes.
- The relative order of namespace nodes and attribute nodes are implementation-dependent.

The instances of the data model could be created basically from two types of XML documents:

- Well-formed XML documents, fulfilling the namespace definitions,
- Validated XML documents, using only DTD or XML Schema.

In the first case, the data model is based on the InfoSet where general entities are resolved. In the second case, the above mentioned PSVI is used fo it. In the XDM the XML document is modeled by tress with the following node types only:

- Root Node:

  The root node is the root of the tree. A root node does not occur except as the root of the tree. The element node for the document element is a child of the root node. The root node also has as children processing instruction and comment nodes for processing instructions and comments that occur in the prolog and after the end of the document element. The string-value of the root node is the concatenation of the string-values of all text node descendants of the root node in document order. The root node does not have an expanded-name.

- Element Nodes:

  There is an element node for every element in the document. An element node has an expanded-name computed by expanding the QName of the element specified in the tag in accordance with the XML Namespaces Recommendation. Its string value is the same as the Root node.

  The namespace URI of the element's expanded-name will be null if the QName has no prefix and there is no applicable default namespace.

- Text Nodes:

  Character data is grouped into text nodes. As much character data as possible is grouped into each text node: a text node never has an immediately following or preceding sibling that is a text node. The string-value of a text node is the character data. A text node always has at least one character of data. A text node does not have an expanded-name.

- Attribute Nodes:

  Each element node has an associated set of attribute nodes; the element is the parent of each of these attribute nodes; however, *an attribute node is not a child of its parent element*. An attribute node has a string-value. The string-value is the normalized value. An attribute whose normalized value is a zero-length string is not treated specially: it results in an attribute node whose string-value is a zero-length string.

- Namespace Nodes:

  Each element has an associated set of namespace nodes, one for each distinct namespace prefix that is in scope for the element (including the XML prefix and one for the default namespace if one is in scope for the element. The element is the parent of each of these namespace nodes; however, a namespace node is not a child of its parent element.

  Elements never share namespace nodes: if one element node is not the same node as another element node, then none of the namespace nodes of the one element node will be the same node as the namespace nodes of another element node.

  A namespace node has an expanded-name: the local part is the namespace prefix (this is empty if the namespace node is for the default namespace); the namespace URI is always null. The string-value of a namespace node is the namespace URI that is being bound to the namespace prefix; if it is relative, it must be resolved just like a namespace URI in an expanded-name.

- Processing Instruction Nodes
- Comment Nodes

Its good to know what are the available properties for these nodes:

- node-name tag: name of the element,

- parent: parent element, could be empty,
- children: child element, could be empty,
- attributes: all the attributes,
- string-value: the aggregated value,
- typed-value: the value of the element (after validation ONLY)
- typed-name: the assigned type's name during the validation

The following example show how data types are changing:

```
<x>
  04<!--  unexpected  comment -->2
</x>
```

The attributes of the element before validation:

- node-name: x,
- parent: ()
- children: (t1; c; t2),
- attributes: -
- string-value: <LF> 042<LF>
- typed-value: <LF> 042<LF>
- typed-name: untypedAtomic

The attributes of the element after validation:

- node-name: x,
- parent: ()
- children: (t1; c; t2),
- attributes: -
- string-value: <LF> 042<LF>
- typed-value: 42
- typed-name: integer

Differences from the DOM

It is prohibited to follow text nodes each other and attribute nodes have not got a parent. CDATA sections are appearing as a text node. Finally, all the entity references are resolved, so they cannot appear as nodes.

Namespaces are always present at each node, not like in DOM where only explicit namespaces are in use.

Chapter 5. XPath

The XML Path Language (XPath) is a declarative, expression oriented query language which is used for selecting nodes from an XML document. XPath 1.0 became a Recommendation in 1999 and is widely implemented and used, either on its own (called via an API from languages), or embedded in languages such as XSLT, XProc, XML Schema or by the discontinued XForms. The primary goal of XPath is to provide a toolset for accessing different parts of an XML document. In practice it means one can use it for addressing, identifying, referencing and locating the (element, attribute, text and all the defined) nodes from an XDM instance. It is based on the logical structure of the document and used for selecting nodes. We can filter all the elements and attributes by different expressions, we can navigate between them using the parent-child relation and we can modify all the values from any supported data types. Finally but not last, it supports namespaces.

Over the node selection functionality it contains built-in functions to deal with data types from text to logical ones, and supports node and node sequences manipulation. XPath did not define anything for the result, its the underlying XPath implementation's task. XPath 2.0 is the current version of the language; it became a Recommendation in 2007. A number of implementations exist but are not as widely used as XPath 1.0. The XPath 2.0 language specification is much larger than XPath 1.0 and changes some of the fundamental concepts of the language such as the type system and the default set oriented behaviour is replaced by sequences. Every expression returns a sequence and several new functions were introduced to support this aspect. Its latest (and most current) version was published in 2010. However, a newer edition, version 3.0 is on its way and currently it is a proposed recommendation dated on 22 October 2013. It has some new aspects: functions became first-class values, so they can be used inline, or as an argument or as a return value which overcomes the limitations of the XDM type system. Nevertheless, in the following sections we will use the XPath 2.0 version.

Context if the expressions

Before we see the typical usage of XPath, we need to understand one of the most important fact: the context. For every expression its context determines the behavior. The processing takes two steps:

1. static analysis:

   an operation tree is built based on the expression and its static context; it need to be normalized and after that a static type is assigned to the expression

2. dynamic evaluation:

   the operation tree, the input data and the dynamic context used to determine the return value of the expression and a dynamic type is assigned to the expression

Context of an expression: contains the information which influences the result of the expression:

- static:

  information which could be gathered during the static analysis, like namespace and schema definitions, function signature

- dynamic:

  information that are available during evaluation; the static context +

  - **context item**: the currently processed element (atomic value or node)
  - context **position** : the position of the context item in the sequence of elements being under processed
  - context**size** : the number of elements in the sequence being under processed
  - values, function implementations, actual dateTime value, ...

  During the evaluation of an expression the context node, size and position could be different! When we think about an `E1/E2` expression or an `E1[e2]` predicate, the evaluation of the steps are always resulting a new context. E.g., in the first case, when we made the step, from E1 to E2, the context for E2 is determined by E1. E2 should be interpreted instead the context of E1.

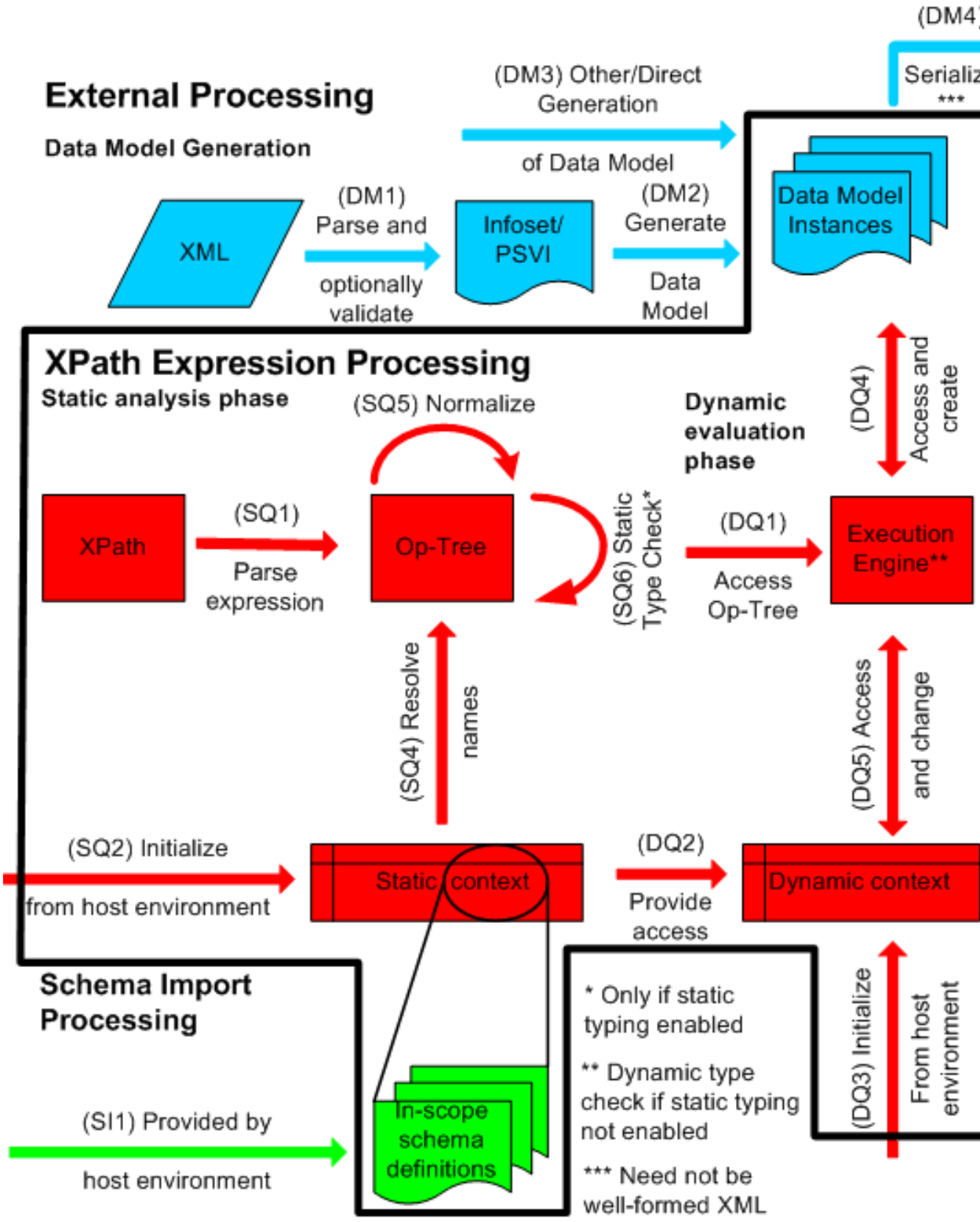
The XPath processing model

The XPath processing model basically based on the data model and the context of the expression. The model contains the following steps:

1. Data model generation:
   - The XML parser generate an InfoSet - and using an optional validation, we can get an PSVI
   - this Infoset or PSVI is transformed to an XDM instance
2. Scheme import's
3. Expression evaluation:
   - static analysis
   - dynamic evaluation
4. Serialization

This processing could be seen in the following figure:

Figure 5.1. The XPath processing model



1. Expressions

Expressions are form the base of XPath. With expressions we can identify different parts of an XML document. Its important to know that an expression could be processed in a given context only. The result of the expression could be in one of the four different types: node sequence, logical, textual or numerical.

Expressions could be divided into three big category: Path expressions, Value expressions and Sequence expressions. Path expressions are discussed in more details in the following section. The Value expressions are

covering the primary expressions (literals, variable references, function calls) and arithmetic, logical and comparison expressions. The third group is discussed in the next part of the book because conditional, iteration and quantified expressions are more better placed in the XQuery section.

For the second group the following remarks are useful: String literals are defined by apostrophes or double quotes. Numbers are interpreted as double precision float pointing numbers. Variables are starting with the special dollar ($) sign which followed by a qualified name (QName). Variables could be any type from the supported XPath types. A function call is performed by the name with parenthesised arguments and those function names should be qualified names where each argument is an expression and the return value could be any supported value.

1.1. Steps

The XPath notation is very similar to the notation for file access by operating systems. Like the UNIX shell environment, the slash (/) is used for separating the steps in an expression. But this notation is used by the URL also, where the first part shows the primary resource and the following steps are identifying the further elements inside this resource.

In XPath the navigation inside the tree starts with the context node (the sequence of them). The navigational syntax is the following:

```
cs₀/step
```

where cs_0 (context sequence 0) denotes the context node sequence, from which a navigation in direction step is taken. It is a common error in XQuery expressions to try and start an XPath traversal without the context node sequence being actually defined. An XPath navigation may consist of multiple steps. Like the following example, step $step_1$ starts off from the context node sequence cs_0 and arrives at a sequence of new nodes cs_1. After that cs_1 is used as the new context node sequence for $step_2$, and so on.

```
cs₀/step₁/step₂/...
   □
((cs₀/step₁)/step₂)/...
  '-------'
     cs₁
```

Most of these expressions are locating different parts of an XML document. These parts are selected with one or more steps. Steps could be start as a *Relative path* starting from the context node or could be an *absolute path* which start with the slash (/) letter. One XPath location step is contains three parts:

```
ax :: nt [p₁] ... [pₙ]
```

1. axis (ax):

   the direction of navigation taken from the context nodes. It defines the relation of the context node and the selected nodes in the tree.

2. node test (nt)

   which can be used to navigate to nodes of certain kind (e.g., only attribute nodes) or name.

3. optional predicates (p_i)

   which further filter the sequence of nodes we navigated to. The given node is only selected if all the predicates are evaluate to true.

The result of one step is always a set (exactly a sequence) of elements. During processing, it creates a set of elements based on the axis and the node test which further filtered by the predicates. The axis and the predicates could be omitted as well. The node test should contain names or meta characters - like the "*" which selects all the elements in the given context.

1.1.1. Axes

Inside the XPath expressions we can use several axes to select nodes. These axes are defining the selection directions relative to the context node. Axes are predicting the ways inside the tree. Practically, any node could be reach form any point form the tree with the help of these axes. The notation for them is the `"::"` operator after the name of the axe.

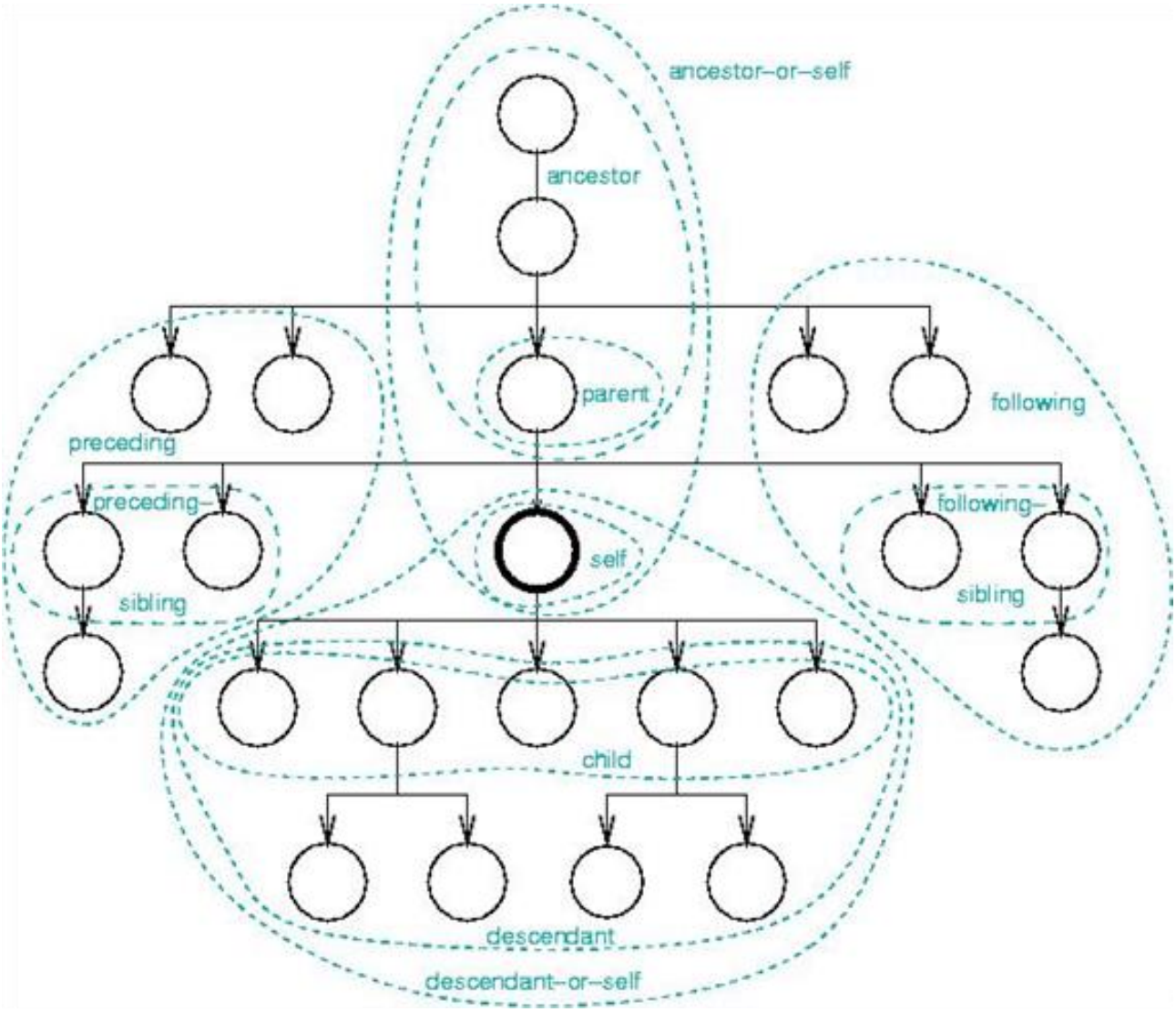
A 12 axes are the following:

Axe	**Description**	**Direction**	**Available nodes**
child::	Contains the children of the context node.	Forward	The children of a document node or element node may be element, processing instruction, comment, or text nodes. Attribute, namespace and document nodes can never appear as children.
parent::	Returns the parent of the context node, or an empty sequence if the context node has no parent	Reverse	Element nodes. **Note:** An attribute node may have an element node as its parent, even though the attribute node is not a child of the element node.
descendant::	Defined as the transitive closure of the child axis; it contains the descendants of the context node.	Forward	Element, processing instruction, comment, or text nodes.
ancestor::	Defined as the transitive closure of the parent axis; it contains the ancestors of the context node	Reverse	Element nodes.
descendant-or-self::	Contains the context node and the descendants of the context node	Forward	Attribute, namespace and document nodes can never appear.
ancestor-or-self	Contains the context node and the ancestors of the context node; thus, the ancestor-or-self axis will always include the root node.	Reverse	Attribute, namespace and document nodes can never appear.
following::	Contains all nodes that are descendants of the root of the tree in which the context node is found, are not descendants of the context node, and occur after the context node in document order	Forward	Attribute, namespace and document root nodes can never appear.
preceding::	Contains all nodes that are descendants of the root of the tree in which the context node is found, are not ancestors of the context node, and occur before the context node in document order	Reverse	Attribute, namespace and document root nodes can never appear.
following-sibling::	Contains the context node's following siblings, those children of the context node's parent that occur after the context node in document order; if the context node is an attribute or namespace node, the following-sibling axis is empty.	Forward	Attribute, namespace and document root nodes can never appear.
preceding-sibling::	contains the context node's preceding siblings, those children of the context node's parent that occur before the	Reverse	Attribute, namespace and document root nodes can never appear.

	context node in document order; if the context node is an attribute or namespace node, the preceding-sibling axis is empty.		
attribute::	Contains the attributes of the context node.	Forward	Just attributes.
namespace::	Contains the namespace nodes of the context node.	Forward	Just namespaces.
self::	Contains just the context node itself.		Could be any node.

These axes could be illustrated with the following figure:

Figure 5.2. XPath axes



XPath semantics:

- The result node sequence of any XPath navigation is returned in document order with no duplicate nodes (recall node identity).

```
(<a b="0">
 <c d="1"><e>f</e></c>
 <g><h/></g>
</a>)/child::node()/parent::node()              => ( <a ..> ... </a> )
```

```
     /child::node()/following-sibling::node()  => ( <c d="1"><e>f</e></c> ,<g><h/></g>
)
```

- XPath semantic follows document order:

```
(<a><b/><c/></a>,
<d><e/><f/></d>)/child::node()          => (<b/>,<c/>,<e/>,<f/>)
```

The XPath document order semantics require <b/> to occur before <c/> and <e/> to occur before <f/>. (Naturally, the result (<e/>,<f/>,<b/>,<c/>) would have been OK as well.)

1.1.2. XPath Node test

Once an XPath step arrives at a sequence of nodes, we may apply a node test to filter nodes based on kind and name.

Kind Test	**Semantics**
Node()	Let any node pass.
Text()	Preserve text nodes only.
Comment()	Preserve comment nodes only.
Processing-instruction()	Preserve processing instructions.
Processing-instruction(p)	Preserve processing instructions of the form `<?p…?>`.
Document-node()	Preserve the (invisible) document root node.

XPath Name test

A node test may also be a name test, preserving only those element or attribute nodes with matching names.

Name test	**Semantics**
name	Preserve element nodes with tag *name***only** (for attribute axis: preserve attributes).
*	Preserve element nodes with arbitrary tag names (for attribute axis: preserve attributes).

Note

Note: In general we will have `cs/ax::*` as a subset of `cs/ax::Node()`.

1.1.3. Predicates

The optional third component of a step formulates a list of predicates [p_1] … [p_N] against the nodes selected by an axis. These predicates are used to give further conditions to fulfill by the nodes.

Its important to underline that **predicates have higher precedence than the XPath step operator** ('/ 'sing):

```
cs/step[ p₁ ][ p₂ ] □ cs/((step[ p₁ ])[ p₂ ])

// The p₁ are evaluated left-to-right for each node in turn.
// In p₁, the current context node 24 is available as '.' .
// Context item, actually: predicates may be applied to sequences of arbitrary items.
```

When using more than one predicate we can apply logical (or, and, not) and comparator (<, >, =, !=) operators as well.

```
/persons/person[@id or number]

// if we use a name inside a predicate without any operator or function, its existence
is checked
```

Moreover, predicates could be nested into each other (unlimited deeply).

```
/shop/items/item[price<2000 and stock[@available=true()]]

// Show all the items whose proce is lower than 2000 Ft and available in stock
```

1.1.4. Atomization

Atomization turns a sequence (x_1, …, x_N) of items into a sequence of atomic values (v_1,.., v_N):

1. If x_i is an atomic value, $v_i \equiv x_i$,
2. if x_i is a node, v_i is the *typed value* of x_i

> Note: the typed value is equal to the string value if x_i has not been validated. In this case, v_i has type `untypedAtomic`.

Atomization could be implicit:

```
(<a>                                          (<b>42</b>,
    <b>42</b>                                  <c><d>42</d></c>,
    <c><d>42</d></c>                    =>     <d>42</d>
    <e>43</e>                                  )
</a>)/descendant-or-self::*[. eq 42]
```

or explicit:

```
(<a>
<b>42</b>
<c><d>42</d></c>
<e>43</e>
</a>)/descendant-or-self::*[data(.) cast as double eq 42 cast as double]
```

1.1.5. Positional access

Inside a predicate [p] the current context item is '.':

- An expression may also access the position of '.' in the context sequence via position(). The first item is located at position 1.
- Furthermore, the position of the last context item is available via last().

```
(x1, x2,...,xn )[position() eq i] => xi
(x1, x2,...,xn )[position() eq last()] => xn
```

A predicate of the form `[position() eq i]` with i being any XQuery expression of numeric type, may be abbreviated by `[i]`.

Furthermore, it is important to remember back to precedence rule because the following example could result surprises:

```
// predicate [.] is stronger than a step (/)
// however, it is evaluated only after them
(cs/descendant-or-self::node()/child::x)[2]
vs.
cs/descendant-or-self::node()/child::x[2]
```

1.1.6. The context item: .

As a useful generalization, XPath makes the current context item '.' available in each step and not only in predicates. It means, in the expression `cs/e` the expression 'e' will be evaluated with '.' set to each item in the context sequence cs (in order). The resulting sequence is returned.

Note: Remember: if e returns nodes (e has type node*), the resulting sequence is sorted in document order with duplicates removed.

```
(<a>1</a>,<b>2</b>,<c>3</c>)/(. + 42) => (43.0,44.0,45.0)
(<a>1</a>,<b>2</b>,<c>3</c>)/name(.)  => ("a","b","c")
```

```
(<a>1</a>,<b>2</b>,<c>3</c>)/position() => (1,2,3)
(<a><b/></a>)/(./child::b, .)          => (<a><b/></a>,<b/>)
```

1.2. Combining node sequences

XPath provides three operators to combine sequences: union (abbreviated as |), intersect and except. These operators remove duplicate nodes based on identity and return their result in document order. Note: We examine these expressions here because several useful query could be built on the top of them.

```
Selecting all x children and attributes of context node
cs/(./child::x | ./attribute::x)

Select all siblings of context node
cs/(./preceding-sibling::node() | ./following-sibling::node())
or
cs/(./parent::node()/child::node() except .)

Select context node and all its siblings
cs/(./parent::node()/child::node() | . )

First common ancestor
(cs0/ancestor::* intersect cs1/ancestor::*)[last()]
```

1.3. Abbreviations

Table 5.1. XPath Abbreviations

Abbreviations	Expansion
nt	child::nt
@	attribute::
..	parent::node()
//	/descendant-or-self::node()/
/	root(.)
step	./step

Examples:

```
a/b/c => ./child::a/child::b/child::c
a//@id => ./child::a/descendant-or-self::node()/attribute::id
//a => root(.)/descendant-or-self::node()/child::a
a/text() => ./child::a/child::text()
```

1.4. XPath 2.0 functions by categories

- Accessors:
  - fn:node-name(node) – Returns an expanded-QName for node kinds that can have names.
  - fn:string(arg) – Returns the value of $arg represented as a xs:string. If no argument is supplied, the context item (.) is used as the default argument.
  - fn:data(item [, item,…]) – Takes a sequence of items and returns a sequence of atomic values.
- Error Function: The fn:error function raises an error. While this function never returns a value, an error is returned to the external processing environment as an xs:anyURI or an xs:QName. The error xs:anyURI is derived from the error xs:QName.
  - fn:error(error, description, object) – Raising an exception.
- Constructor Functions: Constructs object and atomic type instances.
  - xs:date(), xs:string(), xs:Name(), xs:token()

- Numerics: As most of the programming languages, numeric functions made calculations and conversions on numbers. There are several built-in functuions and severek has abbreviated versions, like the „numeric-add” whichis backed up with the + symbol.
  - fn:round(num) - Rounding.
  - fn:abs(num) – Absolute value.
  - fn:number(arg) – Converts a string liter into a number.
- Functions on Strings: Almost operates on strings and returns a string value but not all of the cases.
  - fn:concat(string [, string..]) – Concatenates two or more xs:anyAtomicType arguments cast to xs:string.
  - fn:string-length([| string]) – Returns an xs:integer equal to the length in characters of the value of $arg or if empty, it returns the length of the context item.
  - fn:starts-with(string1, string2)/ fn:ends-with(string1, string2)
  - fn:contains(string1, string2)
  - fn:replace(string, pattern, replace)
- URI functions: Operates with URI. The only one function in this category:
  - fn:resolve-uri(relative, base) – resolve a relative URI into an absolute.
- Boolean functions
  - fn:boolean(arg) – String, numeric or nodes effective boolean value. Note: (), 0, NaN, "" and false() evaluates to false()!
  - fn:not(arg) – Inverts the xs:boolean value of the argument.
- Functions and Operators on Durations, Dates and Times
  - fn:dateTime(date, time) – returns a TimeStamp object.
  - fn:year-from-date(date) – The duration from the given year. It exists in the form of months and dates as well.
  - fn:hours-from-time(time) - Same as above just measured in hours.
  - fn:adjust-dateTime-to-timezone(datetime, timezone) – Converts a timestamp object to the given time zone.
    - There are functions to add, extract and compare dates too.
- Functions Related to QNames (Qualified Name): QName represents XML qualified names. The value space of QName is the set of tuples {namespace name, local part}, where namespace name is an anyURI and local part is an NCName.
  - fn:QName() – Returns an xs:QName with the namespace URI given in the first argument and the local name and prefix in the second argument.
  - fn:local-name-from-QName() – Returns an xs:NCName representing the local name of the xs:QName argument.
  - fn:namespace-uri-from-QName() – Returns the namespace URI for the xs:QName argument. If the xs:QName is in no namespace, the zero-length string is returned.
- Nodeset functions:
  - fn:name([nodeset]) – The name of the actual node or the first element"s in a sequence.

  - fn:root([node]) – Returns the root node.
  - op:is-same-node – True if the two nodes are the same.
- Sequence functions: A sequence is an ordered collection of zero or more items. An item is either a node or an atomic value.
  - fn:count(collection) - Returns the number of items in a sequence.
  - fn:max(collection) - Returns the maximum value from a sequence of comparable values.
  - fn:avg(collection) – Returns the average of a sequence of values.
  - fn:empty(collection) – True if the sequence is empty.
  - fn:exists(collection) – True if the sequence is not empty.
- Context functions: The following functions are defined to obtain information from the dynamic context.
  - fn:last() –Returns the number of items in the sequence of items currently being processed.
  - fn:current-date()/fn:current-time() – returns the current xs:dateTime or xs:time.
  - fn:implicit-timezone() – Returns the value of the implicit timezone property from the dynamic context.
- Other functions: Operators on base64Binary and hexBinary, Operators on NOTATION.

Chapter 6. XQuery

The XQuery for XML is like the SQL for the relational databases, could be used to query and modify data. It became a W3C recommendation in 2006. It is intended to query anything which could be represented as an XML data. Essentially, it contains a data model and the query operators working on it.

XQuery is a truly declarative (and not a procedural) language specifically designed for the purpose of querying XML data. While in procedural languages we need to describe the algorithm to solve a given problem, in declarative languages we need to define conditions and restrictions while the execution engine need to solve the problem. While declarative languages are higher in abstraction as procedural languages, in the case of XQuery we could seen a hybrid approach.

It has programming features like explicit iteration,variable binding, recursion and functions, and also has dynamic features like filtering, grouping and join. In XQuery, the query examines the document (or a part of it) using the restrictions (as filtering, grouping, predicates) and returns all the nodes which are fulfilling the specified rules. That result is a subset of the original XML document in XML format. Technically, XQuery could be used everywhere XML could be. While XML tries to became an universally interpretable data source which is independent from any language or system, therefore XQuery tries to became an universal query language for it.

Both XPath and XQuery are using the same functions and operators on the same data model. Technically, an XQuery query may use XPath expressions but it can be extended with conditions, iterations , variables and functions.

Preliminaries

The most important fact that XQuery is not written in XML but could be used in XSL stylesheets and could be included in HTML files. It has seven different node types: element, attribute, namespace, processing instruction, comment and document.

- Value: always a sequence.
- Sequence: an ordered collection of zero or more items.
- Item: An item is either an atomic value or a node.
- Atomic value: An atomic value is a value in the value space of an atomic type.
- Node: an instance of one of the node kinds defined in XDM. Each of them have identity and string value.

1. Basics of XQuery

Some basic rules::

- Like XML, XQuery is a case-sensitive language. Keywords in XQuery use lower-case characters and are not reserved—that is, names in XQuery expressions are allowed to be the same as language keywords with some exceptions.
- The name of a node is a value of type xs:QName.
- Variables can be defined by the `let` keyword and could be referenced by the `$` sign, e.g.: `let $variable`.
- Conditional expression has the following syntax:

```
if(…)thenelse ...
```

- General comparison is made by standard arithmetic operators, like `=, !=, >, <=`

> **Comparisons:** any XQuery expression evaluates to a sequence of items. Consequently, many XQuery concepts are prepared to accept sequences (as opposed to single items).

```
e₁ θ e₂ where
θ ∈ {=,!=,<,<=,>=,>}
yields true ( true() ),
if any of the items in the sequences e₁,e₂ compare true
(existential semantics)

(1,2,3) > (2,4,5)  => true()
() = 0             => false()
(1,2,3) != 3       => true()
(1,2) != (1,2)     => true()
not((1,2) = (1,2)) => false()
```

- Value comparison operators: `eq, ne, gt, lt`

```
2 gt 1.0               => true()
<x>42</x> eq <y>42</y> => true()
(0,1) eq 0             => (type error)
```

- Node comparison: `is - identity, document order - <<, >>`

```
<x>42</x> eq <x>42</x> => true()
<x>42</x> is <x>42</x> => false()
let $a := <x><y/></x>
return $a << $a/y      => true()
```

- Arithmetic: normal operators as natural: `+, -, *, div, idiv`

  Note: the operator first normalizing the values of the operands and after that converts to the proper type. If any of the operands is an empty sequence, the result will be the same: ().

```
<x>1</x> + 41 => 42.0
() * 42       => ()
(1,2) - (2,3) => (type error)
x-42          => ./child::x-42 (use x - 42)
x/y           => ./child::x/child::y (use x div y)
```

- Data types could be checked with built-in operators: the `instance of` returns true if the element has the given type:

```
patient[id=2]/year instance of xs:integer
```

  The `typeswitch`/case / default language tool could be used to make decisions based on types:

```
let $str := "fertőző"
let $patient := element {$str} {}
// only works on element nodes, so the upper two statement creates it
typeswitch($patient)
   case element(intenzív) return "sürgős"
   case element(fertőző) return "elkülönítendő"
   default return "paciens"
```

2. Dynamic constructors

In the previous sections we could seen several ways to reach the nodes. However, the XQuery language has the tools to create nodes dynamically (during run-time) as well. Its important to highlight the node identity again because the nodes created by the constructors always have a new identity.

While XQuery always flatten the sequences the nested application of node constructors is the only way to hierarchically structure values in XQuery. (It makes it possible to optionally replace an XSLT transformation with XQuery.) XQuery expressions may construct nodes with new identity of all seven node kinds known in XML. XQuery node constructors come in two flavors:

- direct constructors: The syntax of direct constructors exactly matches the XML syntax: any well-formed XML fragment f also is a correct XQuery expression. Note: Text content and CDATA sections are both mapped into text nodes by the XQuery data model (“CDATA sections aren’t remembered.”)

- computed constructors: The syntax of computed constructors are always introduced by a keyword and both its name and content could be computed dynamically of the new node.

The most important constructors:

- document: creates a whole new XML document(`document {}`)

```
document {
<book year="1977">
<title>Harold and the Purple Crayon</title>
<author><last>Johnson</last><first>Crockett</first></author>
<publisher>HarperCollins Juvenile Books</publisher>
<price>14.95</price>
</book>
}
```

Note

Pay attention that existing documents always have the invisible document root node. If we are working with XML fragments this role is played by the most outer element, like in a fragment with the `<x><y/></x>/*` expression results only a `<y/>` element.

- element:
  - creates a new element in a **direct way**:

```
<x><![CDATA[foo & bar]]></x> ≡ <x>foo &amp; bar</x>
// a CDATA nem őrződik meg
```

The tag name of a direct constructor is constant, its content, however, may be computed by any XQuery expression enclosed in curly braces {⋯}. The dynamic construction is evaluated during execution!

```
<tag>3*3</tag> results the same what we see: <tag>3*3</tag>
in contrast, using a dynamic construct:
<tag>{3*3}</tag> the result: <tag>9</tag>
```

However, it can be nested but only the outer dynamic construct is evaluated, inside the construct new evaluation could not occurred:

```
<tag>{<int>3*3</int>}</tag> result only a <int>3*3</int> part.

If we would like to continue this:
<tag>{
    <int> {
        let $x:=3
        return $x*$x
        }
    </int>}
</tag>

Using our example from the surgery system:
for $p in /persons/person
 return <newPerson>
        {
         let $info := $p/information/personalInformation
         return
         {
            <name>{ data($info/name) }</name>,
            <birthDate>{ data($info/birthDate) }</birthDate>
          }
        }
       </newPerson>
```

In the second dynamic construct we list all the new nodes by separating with a coma - resulting a sequence. Inside these nodes we do not have to create new elements, so we have not used the element itself, rather than we used the contained data. This value could be requested from the node by the `data` function.

  - creates a new element in a **dynamic way**:

```
element {expression_for_the_name} {expression_for_the_value }
```

```
element { string-join(("foo","bar"),"-") } { 40+2 } => <foo-bar>42</foo-bar>
```

This construction makes it possible to replace XSLT with XQuery in some situations. Take the example, replacing word in a file using a dictionary:

```
// the dictionary
<entry word="address">
    <variant lang="de">Adresse</variant>
    <variant lang="it">indirizzo</variant>
</entry>
// relacng the element (+ attributes):
element
{ $dict/entry[@word=name($e)]/variant[lang="it"] }
{ $e/@*, $e/node() }
```

- attribute:
  - direct mode:

```
<x a="{(4,2)}"/> => <x a="4 2"/>
```

  - dynamic way:

```
attribute {expression_for_the_name} {expression_for_the_value }
```

```
element book {
attribute year { 1977 },
element author {
    element first { "Crockett" },
    element last { "Johnson" }
},
element publisher {"HarperCollins Juvenile Books"},
element price { 14.95 }
}
```

- text
  - direct mode: characters in the element content with CDATA section. Content sequence e is atomized to yield a sequence of type anyAtomicType*. The atomic values are converted to type string and then concatenated with an intervening " ".
  - dynamic:

```
text {expression}
```

```
if (empty(e))
then ()
else text { string-join(for $i in data(e)
                        return string($i)," ") }
```

The XQuery element constructor is the most flexible because the content sequence is not restricted and may have type item*. The construction consists the following steps:

1. Consecutive literal characters yield a single text node containing these characters.
2. Expression enclosed in {···} are evaluated.
3. Adjacent atomic values are cast to type string and collected in a single text node with intervening " ".
4. A node is copied into the content together with its content. All copied nodes receive a new identity.
5. Then, adjacent text nodes are merged by concatenating their content. Text nodes with content "" are dropped.

```
// Evaluate the expression below:
count(
<x>Fortytwo{40 + 2}{ "pi",3.1415,<y><z/></y>,("","!")[1] }</x>/node()
)
```

The constructed node is:

```
                x            (the value of the text node is: "Negyvenkettő42pi
3.1415")
              /   \
            text   y
                   |
                   z
```

We need to take care on one thing during the construction of the content: well-formed element content is needed. (I.e.: no two attributes of the same element may share a name and attribute nodes precede any other element content.) Finally, construction establishes document order!

3. Iteration: FLWOR

This is one form of the „for" loop in XQuery. Remember back that **XPath performs this in an implicit way** by actualizing the '.' context item in a `cs/e` expression. XQuery makes the same but in a more programmatic way, using a for loop for all the elements in a sequence - resulting the same. The FLWOR word originates the „For, Let, Where, Order by, Return" words.

Let's see what are the role of these blocks:

- **for**: iterating through a sequence.
- **let**: declaring variables.
- **where**: Like in SQL, the passing nodes are selected only.
- **order by**: ordering the selected elements
- **return**: returning with an atomic value or a node

It means we can use the „for" loop to iterate through a sequence by elements. The „where" block helps to work with only the relevant nodes. The „order by" assures the proper order on the result set which is returned after the „return" keyword.

The versatile FLWOR is used to express:

- nested iteration,
- joins between sequences (of nodes),
- groupings,
- orderings beyond document order.

Explicit iteration is expressed using the for ⋯ in construct:

```
for $v [at $p] in e1
return e2
```

This syntax makes it possible to catch the actual position of the currently processed element (this is marked in syntax with p). If e_1 evaluates to the sequence (x 1 ,...,x n), the loop body e_2 is evaluated n times with variable $v bound to each x_i [and $p bound to i] in order. The results of these evaluations are concatenated to form a single sequence.

```
// returning element pairs
for $x in (3,2,1)
return ($x,"*")    => (3,"*",2,"*",1,"*")

// concatenating only one value to the result
for $x in (3,2,1)
return $x,"*"      => (3,2,1,"*")

// Descartes multiplication
for $x in (3,2,1)
```

```
return for $y in ("a","b")
        return ($x,$y) => (3,"a",3,"b",
                           2,"a",2,"b",
                           1,"a",1,"b")
```

Abbreviations:

```
for $v1 in e1          for $v1 in e1          for $v1 in e1
return                   for $v2 in e2          $v2 in e2
    for $v2 in e2          return e3              return e3
        return e3
```

However, it could be used to solve more complicated queries as well:

```
// How deep is a tree:
// (!) Pay attention how the leaf nodes are collected (!)
max( for $i in cs/descendant-or-self::*[not(*)]
     return count($i/ancestor::*) )

// get every odd element
// ( remember back of the nesting of sequences and the ebv of 0 )
for $i at $p in e            expressed better:     for $i at $p in e
return if ($p mod 2)                                where ($p mod 2)
        then e[$p]                                  return e[$p]
        else ()
```

Returning back to the differences between the *explicit and implicit iteration* its important to note the difference between the *context item* as well:

```
XPath:
a[@b = "foo"]/c[2]/d[@e = 42]

XQuery:
for $a in a
where $a/@b = "foo"
return for $c at $p in $a/c
        where $p = 2
        return for $d in $c/d
                where $d/@e = 42
                return $d
```

At first sight, both does the same. the difference is in the handling of the context item. *While XPath replaces it in every step(/), XQuery using the same item during the whole processing!*

3.1. Ordering

A FLWOR expression basically using the original order of the elements inside the sequence to produce the result. However, we can overwrite this behaviour with the well-known "order by" clause. We can use the modifiers like `[ascending|descending]`, applied more than one item (in this case use a coma separated list). In XQuery there is a modifier to specify the position if the empty sequence: it could be placed into the first or to the last position using the `[empty greatest|least]` clause.

Examples:

```
for $i in doc("companys.xml")/persons/person[id lt 10]
 where $i/@sex = "female"
 order by $i/name
 return $i/name

let $a := <a>
<b id="0">42</b>
<b id="1">5</b>
<b id="2"/>
<b id="3">3</b>
<b id="4">1</b>
</a>
for $b in $a/descendant::b
order by $b/text() empty greatest
return $b/@id
```

We need to take into account the following behaviour during ordering - take the following example:

```
let $authors := for $a in doc("books.xml")//author
                order by $a/last, $a/first
                return $a
return $authors/last
```

We need to achieve a list of authors using an ordered list based on last name. The problem with this expression that the result still remain in document order because the '/' and the '//' operators are working in this manner. One need to take into account that these *operators ('/' and '//') are always resulting document order overwriting the order by clause*!

```
The proper solution:
for $a in doc("books.xml")//author
order by $a/last, $a/first
return $a/last
```

During ordering one more thing could come into the picture: handling the same values. Basically, it is irrelevant for ordering when nodes are repeated with the same content, they are following the document order and are placed into the result in the same way. However, if we want to remove these repeating sub-trees than we need to use the `distinct-values()` function which will compare the value's of the nodes and only one will be released from the same values.

```
for $l in distinct-values(doc("books.xml")//author/last)
return <last>{ $l }</last>

// We can do the same with a more complex way, comparing full names:
let $a := doc("books.xml")//author
for $l in distinct-values($a/last),
    $f in distinct-values($a[last=$l]/first)
return
<author>
<last>{ $l }</last>
<first>{ $f }</first>
</author>
```

3.2. Variables

Every FLWOR expression works on *tuple*s what we need to define. XQuery provides two possibilities for it. The for and the let keyword is used for it, so at least one of them need to be used at least one times in an XQuery expression, and there is no restriction which one should it be. Moreover, there is also no restriction of the order of them, so XQuery will accept any variation of these keywords.

The most important issue is to know how the for and let keywords are produce these tuples The case of 'for' is much more clearer because id I specifiy a sequence all of the included items will be processed:

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

result:
<tuple><i>1</i></tuple>
<tuple><i>2</i></tuple>
<tuple><i>2</i></tuple>
```

In contrast, the 'let' is treating it as only one value:

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

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

Combine both of them:

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

```
result:
<tuple><i>1</i><j>1 2 3</j></tuple>
<tuple><i>2</i><j>1 2 3</j></tuple>
<tuple><i>3</i><j>1 2 3</j></tuple>

Using a much more realistic example:
we give the therapist's name for each patient:
for $b in doc("patients.xml")//patient
let $c := $b/therapist
return <patient>{ $b/name, <count>{ count($c) }</patient>}</book>
```

Its important to underline one more behaviour of the variables: **immutability**. All variables bidden by the let keyword could be treated as a named constant, so cannot be modified. The following example show that not so intuitive behaviour:

```
As we would expect in an imperative way:              As we should solve in XQuery:
let $x :=                                             let $x := ... the same ...
<x>
 <y>12</y>
 <y>10</y>
 <y>7</y>
 <y>13</y>
</x>
let $sum := 0                                         for $y in $x//y
for $y in $x//y                                       return 0 + $y
let $sum := $sum + $y
return $sum
```

3.3. Quantified Expressions

There are situations when we need to investigate that some of the sequence's elements are fulfilling or not a given condition. This could be done with qualifiers. With the *existentialquantification* we could check that at least one element passes the condition. It could be seen in the following example:

```
// look for the companies which have AT LEAST ONE registered Hungarian phone lines:
for $x in doc("companys.xml")/company
where some $y in $x/phone satisfies
      starts-with($y/text(),"06") or starts-with($y/text(),"+36")
return $x/name
```

In contrast, if we use *universal quantification* we could check that all of the items in the sequence are met with the condition. Using the same example:

```
// look for the companies which have ONLY Hungarian phone lines:
for $x in doc("companys.xml")/company
where every $y in $x/phone satisfies
      starts-with($y/text(),"06") or starts-with($y/text(),"+36")
return $x/name
```

Its important to note that we use *universal quantification* on an empty sequence the result will be true because the effective boolean value of it is true. (It means companies without a filled out phone element value will be also listed!)

4. Functions

When constructing queries we could reach a level when we should raise the abstraction. This is the point of the functions. We could break down complexity with them and could be reused in any point. In an XQuery document we could create as many as we want just need to be care about the name conflicts. (To avoid this its recommended not to use the built in 'fn' namespace, rather than create locale ones.)

The declaration is done by the following syntax:

```
declare function NAMESPACE:FUNCTION_NAME( $parameter1 [ as DATATYPE[CARDIANALTY] ) ], …
) as RETURNTYPE { };
```

The NAMESPACE cannot be empty. If we do not want to use our custom one than we could use the built-in default namespace called 'local'. The name of the function could be freely selected. The parameter list is

declares the type of the input parameters and we could specify one return type. The cardinality shows the acceptable number of elements for one parameter.

We could use the following cardinalities:

- **? :** zero or one
- **+:** one or more
- ***:** zero or more

For the type of the parameters we could use the following ones:

- **node()**: any node
- **element()**:
  - **element(NAME):** only elements with a NAME start tag accepted
  - without any parameter it will accept all elements
- **scheme-element(NAME)**: only elements defined in a scheme
- **attribute()**: attributes

examples for functions:

```
declare default function namespace "http://www.inf.unideb.hu/FAT/XQuery/functions";

declare function path($n as node()) as xs:string
{ fn:string-join(for $a in $n/ancestor-or-self::*
                    return fn:name($a), "/")
};

declare function reverse($seq)
{ for $i at $p in $seq
    order by $p descending
    return $i
};

// using our default examples
// countng the average high of the patients
declare function local:avg-height($p as element(persons) ) as xs:double {
  avg($p/person/information/medicalInformation/height)
};
// look for a patient with a given name
declare function local:filter($persons as element(persons), $filter as xs:string )   as
xs:string* {
    $persons/person/information/personalInformation/name[../contains(name,$filter)]
};
```

As we see trees during all the solutions and recursion is the most trivial way to deal with them we can see an example for it to simulate thee ancestor axe:

```
declare function ancestors($n as node()?) as node()*
{ if (fn:empty($n)) then ()
else (ancestors($n/..), $n/..)
}
```

and a more complex one - renaming a given attribute in all the elements in a tree:

```
declare function local:rplc_attr($n as node(),
                                 $from as xs:string,
                                 $to as xs:string)
{ typeswitch ($n)
    case $e as element() return
        let $a := ($e/@*)[name(.) eq $from]
        return
            element
```

```
            { node-name($e) }
            { $e/(@* except $a),
                if ($a) then attribute {$to} {data($a)}
                        else (),
                for $c in $e/node()
                 return local:rplc_attr($c, $from, $to) }
    default return $n
};
```

5. Modifying XML documents

XQuery not only could be used for query operations, rather than we could use it for inserting, deleting or renaming nodes The "**insert**" statement is used to insert a node into an other one with the following syntax:

```
insert NODE into [ as [ first | last ] ] TARGET_NODE
insert NODE [ after | before ] TARGET_NODE
```

Inserting a person as a last one:

```
insert <person id="P3"><information/></person> as last into /persons
```

The „**replace**” statement is used to replace one node to a entirely new one with the following syntax:

```
replace node CSOMÓPONT with NEW_NODE              :replacing the whole node
replace value of node CSOMÓPONT with NEW_VALUE    :replacing only the value
```

Replacing a person with P1 id to a person with id P2:

```
replace node /persons/person[@id="P1"] with <person id="P2"></person>
```

Changing patient P1's name:

```
replace value of node /persons/person[@id="P1"]/information/personalInformation/name
with "Ödön"
```

The "**delete**" statement is used to remove nodes:

```
delete NODE
```

Deleting all the persons where the SSN is empty:

```
delete /persons/person[not(information/medicalInformation/SSN)]
```

Chapter 7. Exercises

We will see some exercises regarding to the well-know Company scheme from the relational world (and so, compared to the SQL version as well):

1. Task: Count all the employees!

   Solution:

```
SQL: select count(*) from emp;
XQuery: count(//Emp)
```

2. Task: Show all Employees with a salary greater than 1000.

   Solution:

```
SQL: select * from emp where sal > 1000;
XQuery: //Emp[Sal>1000]
```

3. Task: Show all Employees with a salary greater than 1000 and less than 2000.

   Solution:

```
SQL: select * from emp where sal between 1000 and 2000;
XQuery: //Emp[Sal>1000][Sal<2000]

(: there is no between function in XQuery, so write one :)
declare function local:between($value as xs:decimal, $min as xs:decimal, $max as
xs:decimal) as xs:boolean
{
 $value >= $min and $value <= $max
};
(: the modified solution :)
//Emp[local:between(Sal,1000,2000)]
```

4. Task: Show all employees with no Commission.

   Solution:

```
SQL: select * from emp where comm is null;
XQuery: //Emp[empty(Comm/text())]
```

   ### Note

   using simply the expression //Emp[empty(Comm)] could result wrong output. Its not acceptable because the empty(Comm) function returns true if the element is missing.

5. Task: Select the first 5 employees.

   Solution:

```
SQL: select * from emp limit 5;
XQuery: //Emp[position() <=5]
```

6. Task: Compute the Annual Salaries of all employees. The Annual Salary is computed from 12 times the Monthly salary plus Commission. Since commission may be null, it must be replaced by a suitable numeric value.

   Solution:

```
SQL: select 12 * sal + ifnull(comm,0) from emp;
XQuery: //Emp/(12*number(Sal)+(if(exists(Comm/text())) then number($Comm) else 0))
```

7. Task: List the employee names with their Annual Salary.

   Solution:

```
SQL: select ename, 12 * sal + ifnull(comm,0) as "Annual Salary" from emp;
XQuery:
for $emp in //Emp
return <Emp> {$emp/Ename} <AnnualSalary>
 {12*number($emp/Sal)+
 (if (exists($emp/Comm/text())) then number($emp/Comm)
  else 0)
  }
 </AnnualSalary>
</Emp>
```

8. Task: List all Employees whose name contains "AR".

   Solution:

```
SQL: select * from emp where ename like "%AR%"
XQuery://Emp[contains(Ename,"AR")]
```

9. Task: Find the name of the department that employee 'SMITH' works in!

   Solution:

```
SQL: select dept.dname from emp, dept where dept.deptno = emp.deptno and
ename='SMITH';
XQuery: let $dept := //Emp[Ename='SMITH']/DeptNo return //Dept[DeptNo = $dept ]/Dname
XPath:  //Dept[DeptNo = //Emp[Ename='SMITH']/DeptNo]/Dname
```

10. Task: List the departments and the number of employees in each department

    Solution:

```
SQL: select dname, count(*) from dept natural join emp group by emp.deptno;
XQuery:
for $dept in //Dept let $headCount := count(//Emp[DeptNo=$dept/DeptNo])
return <Dept>
 {$dept/Dname}
 <HeadCount>{$headCount}</HeadCount>
 </Dept>
```

11. Task: List the name of each employee together with the name of their manager.

    Solution:

```
SQL: select e.ename, m.ename from emp e, emp m where e.mgr = m.empno;
XQuery:
for $emp in //Emp
 let $manager := //Emp[EmpNo = $emp/MgrNo] return <Emp>
 {$emp/Ename} <Manager>{string($manager/Ename)}</Manager>
 </Emp>
```

12. Task: Show the number, average (rounded), min and max salaries for Managers.

    Solution:

```
SQL: select count(*), round(avg(sal)), min(sal), max(sal) FROM emp WHERE
job='MANAGER';
XQuery:
let $managers := //Emp[Job='MANAGER'] return <Statistics>
<Count>{(count($managers}</Count> <Average>{round(avg($managers/Sal))}</Average>
<Min>{min($managers/Sal)}</Min> <Max>{max($managers/Sal))}</Max>
 </Statistics>
```

13. Task: Show the number, average (rounded), min and max salaries for each Job where there are at least 2 employees in the group.

    Solution:

```
SQL: select job, count(*), round(avg(sal)), min(sal), max(sal) FROM emp GROUP BY job
HAVING count(*) > 1;
XQuery:
```

```
for $job in distinct-values(//Emp/Job)
let $employees := //Emp[Job=$job]
 where count($employees) > 1
 return <Statistics>
  <Job>{$job}</Job>
  <Count>{count($employees )}</Count> <Average>{round(avg($employees /Sal))}</Average>
<Min>{min($employees /Sal)}</Min>
  <Max>{max($employees /Sal)}</Max>
 </Statistics>
```

14. Task: List the departments , their employee names and salaries and the total salary in each department.

Solution:

```
SQL: This must generate a nested table.
XQuery:
<Report> {
 for $dept in //Dept
 let $subtotal := sum(//Emp[DeptNo = $dept/DeptNo]/Sal)
 return
  <Department>
   {$dept/Dname}
   {for $emp in //Emp[DeptNo = $dept/DeptNo]
    return <Emp> {$emp/Ename} {$emp/Sal} </Emp>
    }
    <SubTotal>{$subtotal}</SubTotal>
  </Department> }
  <Total>{sum(//Emp/Sal)}</Total>
 </Report>
```

Bibliography

Books

[FAWCET2012] Fawcett Joe, R. E. Quin Liam, and Ayers Danny. *Beginning XML*. 2012. 5. 1118162137. 864.

[KAY2008] Michael Kay. *XSLT 2.0 and XPath 2.0*. Programmer's Reference. 2008. 4. 0470192747 . 1368.

[POWELL2006] Gavin Powell. *Beginning XML Databases*. 2006. 1. 0471791202. 470.

[WALMSEY2012] Priscilla Walmsley. *Definitive XML Schema*. 2012. 2. 0132886723. 768.

[TOMPA2011] Frank Tompa and Airi Salminen. *Communicating with XML*. 2011. 1. 1461409918. 240.

[JIAHENG2013] Jiaheng Lu. *An Introduction to XML Query Processing and Keyword Search*. 2013. 1. 3642345549. 300.

[SIEGEL2014] Erik Siegel and Adam Rette. *eXist*. 2014. 1. 1449337104. 400.

Web resources

[W3] *The World Wide Web Consortium (W3C)*. Web page .

[W3XPATH] *XML Path Language (XPath) Version 2.0*. Web page .

[W3XDM] *XQuery 1.0 and XPath 2.0 Data Model (XDM) (Second Edition)*. Web page .

[W3XFUNCT] *XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)*. Web page .

[W3XSEMANTCS] *XQuery 1.0 and XPath 2.0 Formal Semantics (Second Edition))*. Web page .

[W3XML] *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. Web page .

[W3XNAMESPC] *Namespaces in XML 1.0 (Third Edition)*. Web page .

[W3XINFOSET] *XML Information Set (Second Edition)*. Web page .

[IBMDEVXMLDB] *The XML Database Blog* . Web page .