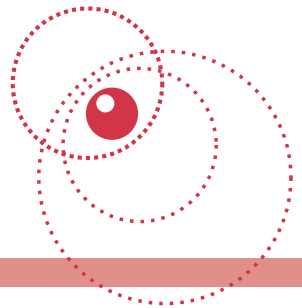


CS4221: Database Applications Design and Tuning

Week 7: NoSQL Databases (1/2)

3 March 2016

Pierre Senellart

TELECOM ParisTech

SQL, NoSQL, NewSQL

- Based on the relational model
- A standard query language: SQL
- Data stored on disk
- Relations (tables) stored row after row
- Centralized systems, limited distribution possibilities

- independence between:
  - data model and storage structure
  - declarative queries and execution

- Independence between:
  - data model and storage structure
  - declarative queries and execution
- Complex queries

- Independence between:
  - data model and storage structure
  - declarative queries and execution
- Complex queries
- Very fine query optimization, index allowing quick access to data

- Independence between:
  - data model and storage structure
  - declarative queries and execution
- Complex queries
- Very fine query optimization, index allowing quick access to data
- Mature, stable, efficient software, wealth of features and of interfaces

- Independence between:
  - data model and storage structure
  - declarative queries and execution
- Complex queries
- Very fine query optimization, index allowing quick access to data
- Mature, stable, efficient software, wealth of features and of interfaces
- Integrity constraints ensuring invariants on data

- Independence between:
  - data model and storage structure
  - declarative queries and execution
- Complex queries
- Very fine query optimization, index allowing quick access to data
- Mature, stable, efficient software, wealth of features and of interfaces
- Integrity constraints ensuring invariants on data
- Efficient management of large data volume (gigabytes, even terabytes)

- Independence between:
  - data model and storage structure
  - declarative queries and execution
- Complex queries
- Very fine query optimization, index allowing quick access to data
- Mature, stable, efficient software, wealth of features and of interfaces
- Integrity constraints ensuring invariants on data
- Efficient management of large data volume (gigabytes, even terabytes)
- Transactions (set of elementary operations) for concurrency control, user isolation, error recovery

Transactions of classical relational DBMSs respect the ACID properties.

Transactions of classical relational DBMSs respect the ACID properties.

Atomicity: The set of operations of a transaction is either executed as a block, or canceled as a block

Transactions of classical relational DBMSs respect the ACID properties.

Atomicity: The set of operations of a transaction is either executed as a block, or canceled as a block

Consistency: Transactions respect integrity constraints of the database

Transactions of classical relational DBMSs respect the ACID properties.

Atomicity: The set of operations of a transaction is either executed as a block, or canceled as a block

Consistency: Transactions respect integrity constraints of the database

Isolation: Two concurrent executions of transactions result in a state equivalent to serial execution

Transactions of classical relational DBMSs respect the ACID properties:

Atomicity: The set of operations of a transaction is either executed as a block, or canceled as a block

Consistency: Transactions respect integrity constraints of the database

Isolation: Two concurrent executions of transactions result in a state equivalent to serial execution

Durability: Once a transaction is committed, data remain durably stored in the database, even in case of (e.g., hardware) failure

- Inability to manage very large data volumes (order of magnitude of petabytes)

- Inability to manage very large data volumes (order of magnitude of petabytes)
- Impossible to manage extreme loads (thousands of queries per seconds and more)

- Inability to manage very large data volumes (order of magnitude of petabytes)
- Impossible to manage extreme loads (thousands of queries per seconds and more)
- The relational model is not suitable to storage and querying of some data types (hierarchical data, weakly structured data, semi-structured data)

- Inability to manage very large data volumes (order of magnitude of petabytes)
- Impossible to manage extreme loads (thousands of queries per seconds and more)
- The relational model is not suitable to storage and querying of some data types (hierarchical data, weakly structured data, semi-structured data)
- ACID properties lead to serious overheads in latency, disk access, CPU time (locks, logging, etc.)

- Inability to manage very large data volumes (order of magnitude of petabytes)
- Impossible to manage extreme loads (thousands of queries per seconds and more)
- The relational model is not suitable to storage and querying of some data types (hierarchical data, weakly structured data, semi-structured data)
- ACID properties lead to serious overheads in latency, disk access, CPU time (locks, logging, etc.)
- Performance limited by disk accesses

- No SQL or Not Only SQL
- DBMSs with other trade-offs than those made by classical systems
- Very diversified ecosystem
- Desiderata: different data model, transparent scaling up, extreme performances
- Features abandoned: ACID, (possibly) complex queries

Type	Organization	Queries	Examples of systems
XML	Treelike, hierarchical data	XQuery	 

Type	Organization	Queries	Examples of systems
XML	Treelike, hierarchical data	XQuery	 
Object	Complex data, with properties and methods	OQL, VQL	 

Type	Organization	Queries	Examples of systems
XML	Treelike, hierarchical data	XQuery	BaseX, eXistdb
Object	Complex data, with properties and methods	OQL, VQL	Versant
Graph	Graph with vertices, edges, labels	Cypher, Gremlin	Neo4j the graph database

Type	Organization	Queries	Examples of systems
XML	Treelike, hierarchical data	XQuery	BaseX, eXistdb
Object	Complex data, with properties and methods	OQL, VQL	Versant
Graph	Graph with vertices, edges, labels	Cypher, Gremlin	Neo4j the graph database
Triples	RDF triples from the Semantic Web	SPARQL	

Simple queries:

get retrieves the value mapped to a key

put adds a new key/value pair

- Stress put on transparent scaling up, low latency, very high bandwidth
- Example of implementation: distributed hash table

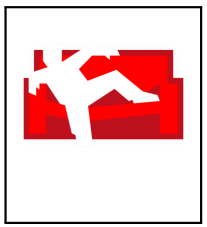
Chord MemcacheDB

still very simple queries.

get retrieves the document (JSON, XML, YAML) mapped to a key

put maps a new document to a key

- Additional indexes allow retrieval of documents containing a keyword, having a given property, etc.
- Documents organized in collections, metadata (versions, dates) management, etc.
- Accent put on interface simplicity, ease of handling in a programming language



Instead of storing data row after row, store it column after column

- Richer organization than key-value stores (several column by stored object)
- Makes aggregating or scanning the values of a given column more efficient
- Transparent distribution, scaling up thanks to distributed search trees or distributed hash tables

BigTable


Some applications require:

- rich query languages (joins, aggregation)
- conformity to ACID properties
- but higher performances than classical DBMSs

- Some applications require:
  - rich query languages (joins, aggregation)
    - conformity to ACID properties
    - but higher performances than classical DBMSs
- Possible solutions:
  - Get rid of classical bottlenecks of DBMSs: locks, logging, cache management
  - Main-memory database, with asynchronous copy to disk
  - Lock-free concurrence management (MVCC)
  - Shared-nothing distributed architecture, transparent load balancing

Google Spanner Clustrix VOLTDB

Extreme latency or bandwidth requirements

Extreme latency or bandwidth requirements

Extreme data volumes

- Extreme latency or bandwidth requirements
- Extreme data volumes
- When the relational model and SQL poorly suit storage and data access needs (not that frequent!)

- Extreme latency or bandwidth requirements
- Extreme data volumes
- When the relational model and SQL poorly suit storage and data access needs (not that frequent!)
- When, after extensive tests, performances of classical DBMSs prove insufficient

- Extreme latency or bandwidth requirements
- Extreme data volumes
- When the relational model and SQL poorly suit storage and data access needs (not that frequent!)
- When, after extensive tests, performances of classical DBMSs prove insufficient
- Know what you lose: (depending on the case) ACID, possibility of complex querying, stability of well-established software, etc.

- Extreme latency or bandwidth requirements
- Extreme data volumes
- When the relational model and SQL poorly suit storage and data access needs (not that frequent!)
- When, after extensive tests, performances of classical DBMSs prove insufficient
- Know what you lose: (depending on the case) ACID, possibility of complex querying, stability of well-established software, etc.
- NoSQL and NewSQL databases answer real needs but needs are often overestimated

- Focus on the data model and query language aspects
- No consideration for the systems aspects (important, but out of scope, see CS5344)
- This lecture:
  - the semi-structured data model
  - XML
  - XPath
- Next lecture:
  - Graph databases and triple stores: data representations and query languages (Cypher, SPARQL)
  - Document store (MongoDB): query facilities
  - BigTable/HBase: data organization and access

Semi-Structured Data

XML

A data model, based on graphs, for representing both regular and irregular data.

Basic ideas

Self-describing data. The content comes with its own description; contrast with the relational model, where schema and content are represented separately.

Flexible typing. Data may be typed (i.e., "such nodes are integer values" or "this part of the graph complies to this description"); often no typing, or a very flexible one

Serialized form. The graph representation is associated to a serialized form, convenient for exchanges in an heterogeneous environment.

Starting point: association lists, i.e., records of label-value pairs.

```
{name: "Alan", tel: 2157786,  email: "agg@abc.com"}
```

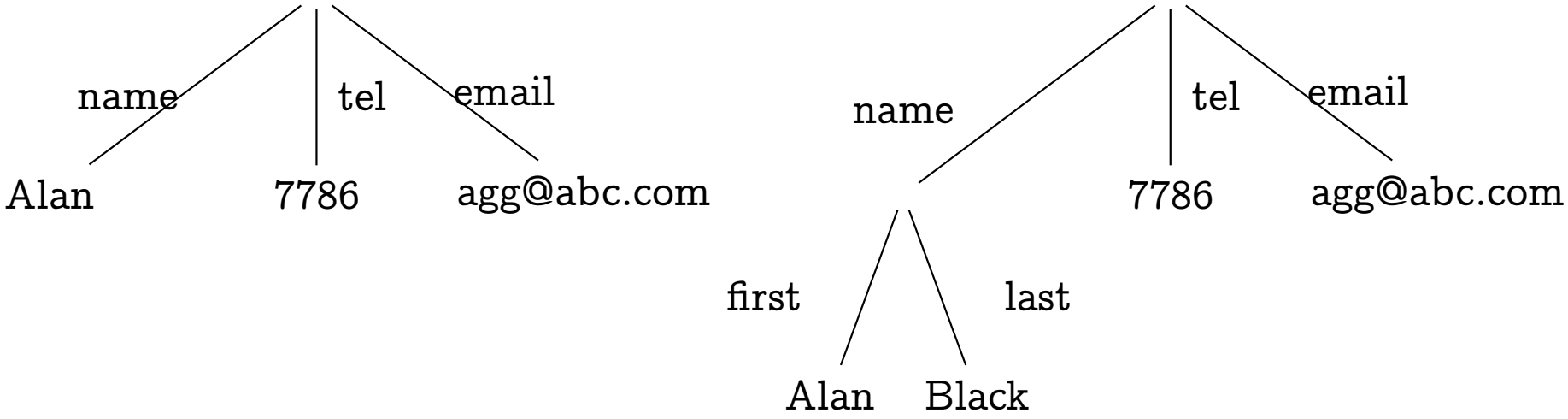
Natural extension: values may themselves be other structures:

```
{name: {first: "Alan",  last: "Black"},
 tel: 2157786,
 email: "agg@abc.com"}
```

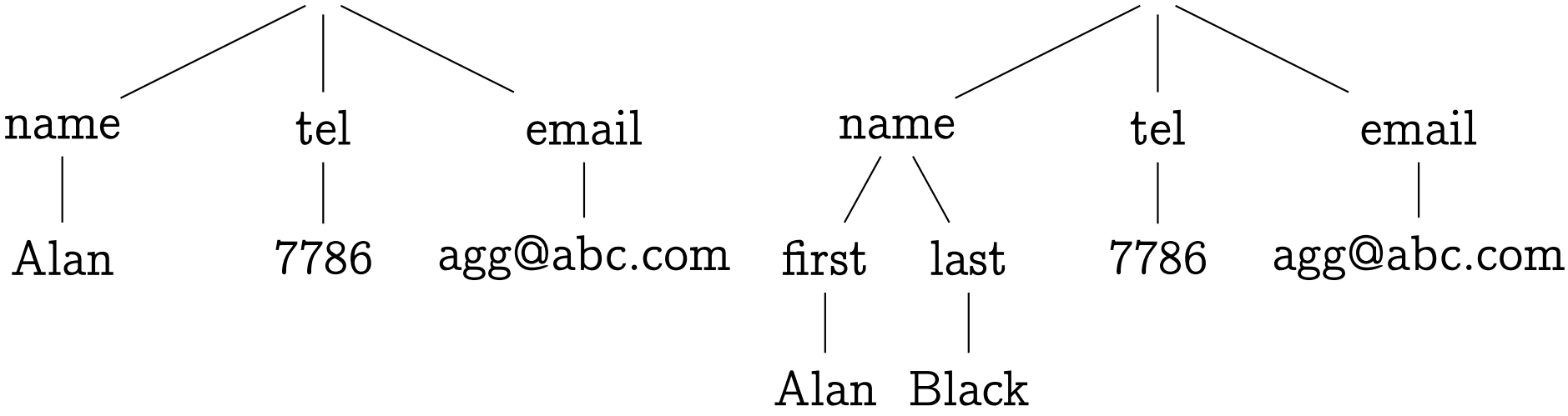
Further extension: allow duplicate labels.

```
{name: "Alan", tel:  2157786, tel: 2498762}
```

Data can be graphically represented as trees: label structure can be captured by tree edges, and values reside at leaves.



Another choice is to represent both labels and values as vertices.



Remark

The XML data model adopts this latter representation.

The key idea that makes it easy to describe sets of tuples as in:

```
{ person:  {name: "Alan", phone: 3127786, email: "alan@abc.com"},
  person:  {name: "Sara", phone: 2136877, email: "sara@xyz.edu"},
  person:  {name: "Fred", phone: 7786312, email: "fd@ac.uk"} }
```

Remark

1. *relational data can be represented*
2. *for regular data, the semi-structure representation is highly redundant.*

changes, etc.

```
{person: {name: "Alan", phone: 3127786, email: "agg@abc.com"},
 person: &314
         {name: {first: "Sara", last: "Green"                 },
          phone: 2136877,
          email: "sara@math.xyz.edu",
          spouse: *443                                        },
 person: &443
         {name: "Fred", Phone: 7786312,  Height: 183,
          spouse: *314                                        }}
```

(this is a hypothetical syntax, not an an actual language!)
Nodes can be identified, and referred to by their identity. Cycles and objects models can be described as well.

Semi-Structured Data

XML

- Introduction
- XML syntax
- Typing

Semi-Structured Data

XML

Introduction

XML syntax

Typing

XML (eXtensible Markup Language) is the World-Wide-Web Consortium (W3C) standard for Web data exchange.

- XML documents can be serialized in a normalized encoding (typically iso-8859-1, or utf-8), and safely transmitted on the Internet.
- XML is a generic format, which can be specialized in "dialects" for specific domain (e.g., XHTML, RSS, SVG, GedML, MathML, etc.)
- The W3C promotes companion standards: DOM (object model), XML Schema (typing), XPath (path expression), XSLT (restructuring), XQuery (query language), and many others.

Remark

1. *XML is a simplified version of SGML (Standard Generalized Markup Language), a long-term used language for technical documents.*
2. *HTML, up to version 4.01, is also a variant of SGML. Some modern versions of HTML (XHTML 1.0, XHTML5) are XML dialects.*

An XML document is a labeled, unranked, ordered tree:

Labeled means that some annotation, the label, is attached to each node.

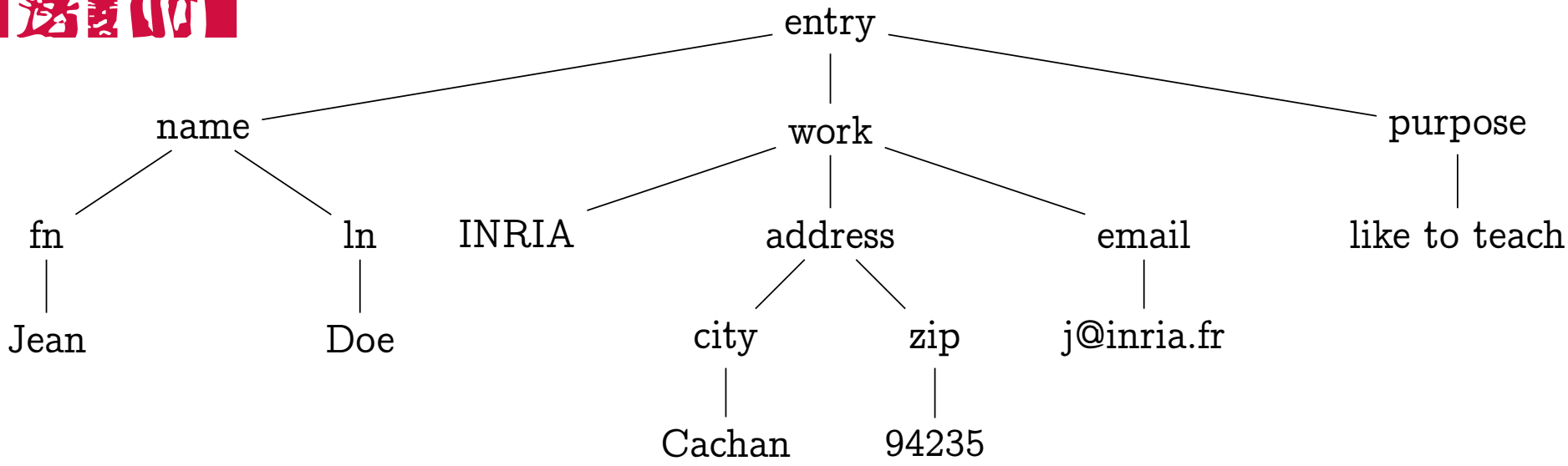
Unranked means that there is no a priori bound on the number of children of a node.

Ordered means that there is an order between the children of each node.

XML specifies nothing more than a syntax: no meaning is attached to the labels.

A dialect, on the other hand, associates a meaning to labels (e.g., `title` in XHTML).

Applications view an XML document as a labeled, unranked, ordered tree:



Remark

Some low-level software works on the serialized representation of XML documents, notably SAX (a parser and an API).

```
<address><city>Cachan</city><zip>94235</zip></adress><email>
j@inria.fr</email></work><purpose>like to teach</purpose></entry>
```

or with some beautification as:

```
<entry>
   <name>
      <fn>Jean</fn>
      <ln>Doe</ln> </name>
   <work>
      INRIA
      <adress>
         <city>Cachan</city>
         <zip>94235</zip> </adress>
      <email>j@inria.fr</email> </work>
   <purpose>like to teach</purpose>
</entry>
```


The book "Foundations of Databases", written by Serge Abiteboul, Richard Hull, and Victor Vianu, published in 1995 by Addison-Wesley.

XML provides a means to structure this content:

```
<bibliography>
  <book>
    <title>Foundations of Databases</title>
    <author>Abiteboul</author>
    <author>Hull</author>
    <author>Vianu</author>
    <publisher>Addison Wesley</publisher>
    <year>1995</year> </book>
  <book>...</book>
</bibliography>
```

Now, an application can access the XML tree, extract some parts, rename the labels, reorganize the content into another structure, etc.

Semi-Structured Data

XML

- Introduction
- XML syntax
- Typing

Typically, an application gets a document in *serialized form*, parse it in the *tree form*, and *serializes* it back at the end.

- The serialized form is a textual, linear representation of the tree; it complies to a (sometimes complicated) syntax;
- There exist an object-oriented model for the tree form: the *Document Object Model* (W3C).

Remark

We present here the most significant aspects of both the syntax and the DOM. Details can be found in the W3C documents.

```
<document/>

<document> Hello World! </document>

<document>
  <salutation> Hello World! </salutation>
</document>

<?xml version="1.0" encoding="utf-8" ?>
<document>
  <salutation color="blue"> Hello World! </salutation>
</document>
```

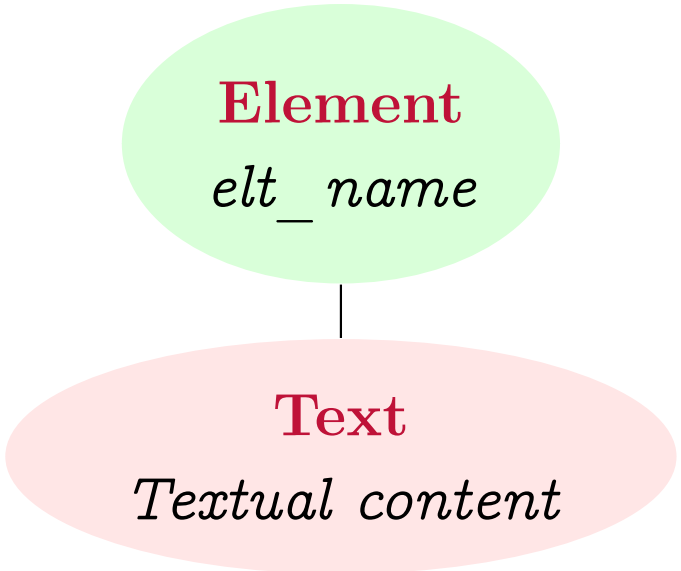
Last example shows the optional *prologue*.

The basic components of an XML document are *element* and *text*.

Here is an *element*, whose content is a *text*.

```
<elt_name>
  Textual content
</elt_name>
```

The tree form of the document, modeled in DOM: each node has a type, either Document or Text.

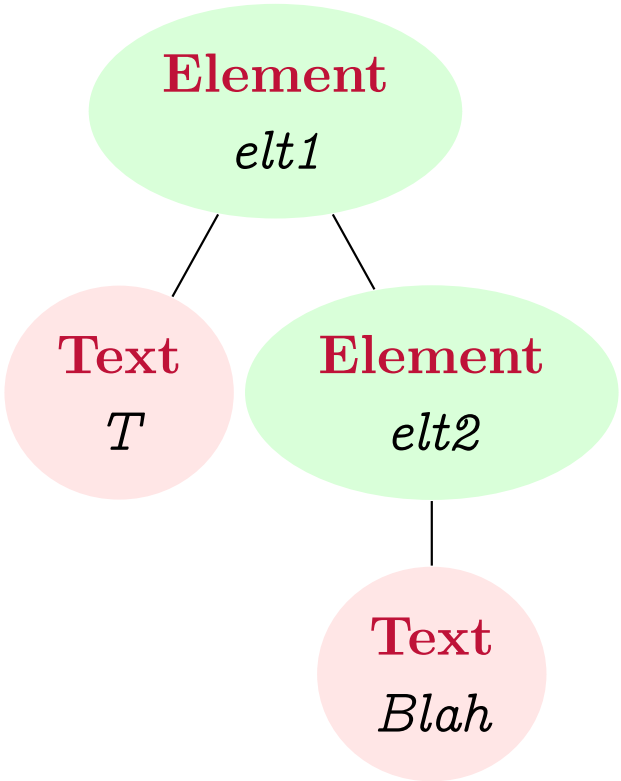


1. the part between the opening and ending tags (in serialized form),
2. the subtree rooted at the corresponding Element node (in DOM).

The content may range from atomic text, to any recursive combination of text and elements (and gadgets, e.g., comments).

Example of an element nested in another element.

```
<elt1>
  T
  <elt2>
    Blah
  </elt2>
</elt1>
```



Attributes are pairs of name/value attached to an element.

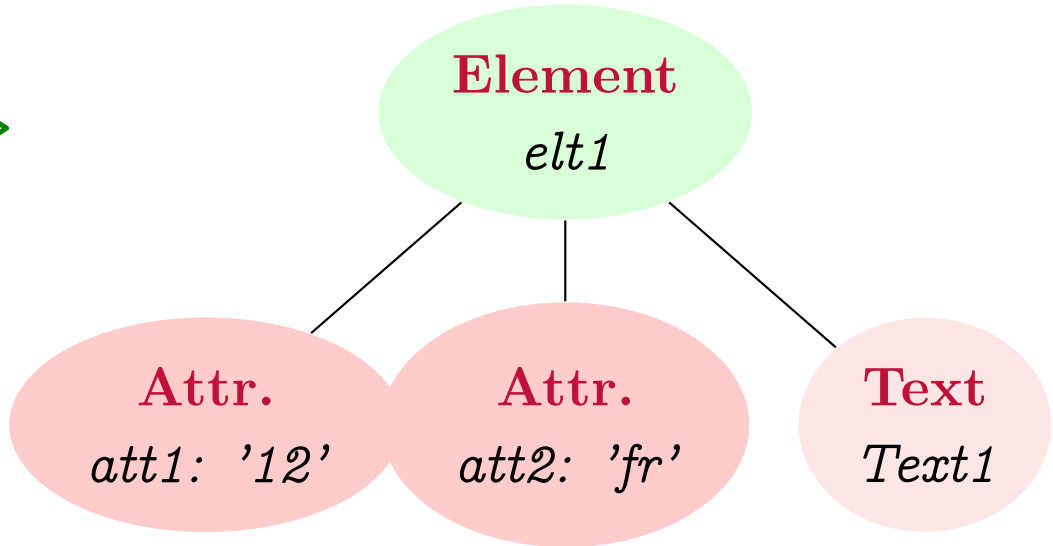
1. as part of the opening tag in the serialized form,
2. as special child nodes of the Element node (in DOM).

The content of an attribute is always atomic text (no nesting).

An element with two attributes.

```
<elt1 att1='12' att2='fr'>
  Text1
</elt1>
```

Unlike elements, attributes are *not* ordered, and there cannot be two attributes with the same name in an element.



<?xml version="1.0" encoding="utf-8" ?>

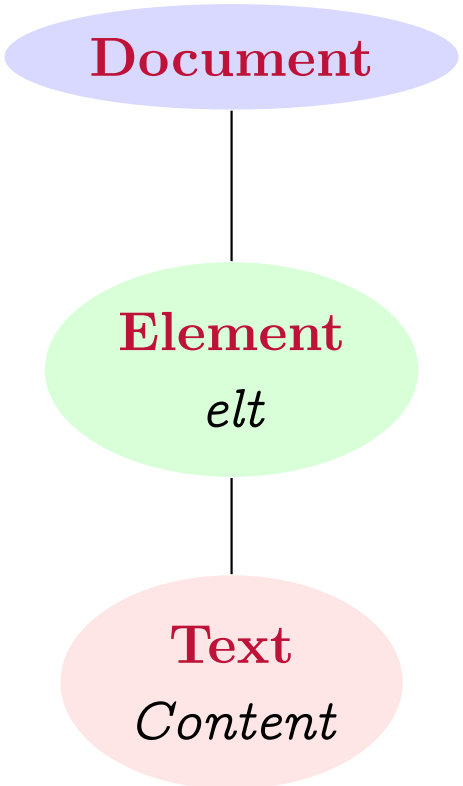
and that document content must *always* be enclosed in a single opening/ending tag, called the *element root*.

A document with its prologue, and element root.

```
<?xml version="1.0"
    encoding="utf-8" ?>
<elt>
  Content
</elt>
```

Note: there may be other syntactic objects after the prologue (processing instructions).

In the DOM representation, the document itself appears as a Document node, called the *root node*.



- A document begins with a prologue,
- It consists of a single upper-level tag,
- Each *opening tag* <name> has a corresponding *closing tag* </name>; everything between is either text or properly enclosed tag content.

Tree form

- A document is a tree with a *root node* (Document node in DOM),
- The root node has one and only one element child (Element node in DOM), called the *element root*)
- Each element node is the root of a *subtree* which represents its structured *content*

Remark

Other syntactic aspects, not detailed here, pertain to the physical organization of a document.

XML provides a syntax

The core of the syntax is the element name

Element names have no a priori semantics

Applications assign them a semantics

Entities are used for the physical organization of a document.

An entity is declared (in the document type definition), then referenced.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE a [
  <!ENTITY  myName "John Doe">
  <!ENTITY  mySignature SYSTEM "signature.xml">
]>

<a>
  My name is &myName;.

  &mySignature;
</a>
```

Several symbols cannot be directly used in an XML document, since they would be misinterpreted by the parser.

They must be introduced as entity references.

Declaration	Reference	Symbol.
`<!ENTITY lt "&#60;">`	`&lt;`	<
`<!ENTITY gt "&#62;">`	`&gt;`	>
`<!ENTITY amp "&#38;">`	`&amp;`	&
`<!ENTITY apos "&#39;">`	`&apos;`	’
`<!ENTITY quot "&#34;">`	`&quot;`	"

Comments can be put at any place in the serialized form.

```
<!{-}- This is a comment {-}->
```

They appear as Comment nodes in the DOM tree (they are typically ignored by applications).

Processing instructions: specific commands, useful for some applications, simply ignored by others. The following instruction requires the transformation of the document by an XSLT stylesheet:

```
<?xml-stylesheet href="prog.xslt" type="text/xslt"?>
```


Problem: what if we do *not* want the content to be parsed?

```
<program>
if ((i < 5) && (j > 6))
   printf("error");
</program>
```

Solution: use entities for all special symbols; or prevent parsing with a *literal section*.

```
<program>
<![CDATA[if ((i < 5) && (j > 6))
           printf("error");
]]>
</program>
```

Semi-Structured Data

XML

- Introduction
- XML syntax
- Typing

What kind of data: very regular one (as in relational databases), less regular (hypertext systems) – all kind of data from very structured to very unstructured.

What kind of typing (unlike in relational systems):

- Possibly irregular, partial, tolerant, flexible
- Possibly evolving
- Possibly very large and complex
- Ignored by some applications such as keyword search.

Typing is not compulsory.

XML documents *may* be typed, although they do not need to. The simplest (and oldest) typing mechanism is based on *Document Type Definitions* (DTD).

A DTD may be specified in the prologue with the keyword DOCTYPE using an ad hoc syntax.

A document with proper opening and closing of tags is said to be well-formed.

- `<a><b>...</b><c>...</c></a>` is well-formed.
- `<a><b>...</b><c>...</a></c>` is not.
- `<a><b>...</b></a><a>...</a>` is not.

A document that conforms to its DTD is said to be valid

```
<?xml version "1.0" standalone "yes" ?>
<!-- Example of a DTD -->
<!DOCTYPE email [
  <!ELEMENT  email ( header, body )>
  <!ELEMENT header ( from, to, cc? )>
  <!ELEMENT to (#PCDATA)>
  <!ELEMENT from (#PCDATA)>
  <!ELEMENT cc (#PCDATA)>
  <!ELEMENT body (paragraph*) >
  <!ELEMENT paragraph (#PCDATA)> ]>
<email>
  <header>
     <from> af@abc.com </from>
     <to> zd@ugh.com </to> </header>
  <body> </body> </email>
```


A DTD may also be specified externally using an URI.

```
<!DOCTYPE docname SYSTEM "DTD-URI" [local-declarations]>
```

- `docname` is the name of the element root
- `DTD-URI` is the URI of the file that contains the DTD
- `local-declarations` are local declarations (mostly for entities.)

A particular label, e.g., *job*, may denote different notions in different contexts, e.g., a hiring agency or a computer ASP (application service provider).

The notion of namespace is used to distinguish them.

```
<doc xmlns:hire='http://a.hire.com/schema'
     xmlns:asp='http://b.asp.com/schema'  >
  ...
  <hire:job> ... </hire:job> ...
  <asp:job> ... </asp:job> ...
</doc>
```

DTD: old style typing, still very used

XML Schema: more modern, used e.g. in Web services

DTD:

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

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="note">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="to" type="xs:string"
          minOccurs='1' maxOccurs='1'/>
        <xs:element name="from" type="xs:string"/>
        <xs:element name="heading" type="xs:string"/>
        <xs:element name="body" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```