

Traitement des données NoSQL

STI8 – EA Big Data, 2018–2019

Radu Ciucanu

radu.ciucanu@insa-cvl.fr



Page Celene du cours :

<https://celene.insa-cvl.fr/course/view.php?id=361>

Modalités de Contrôle des Connaissances

100% Contrôle Continu

- 40% : Qualité des rendus sur Celene suite aux TD sur machine
 - 4 sujets, chacun traité pendant 2 séances de TD
- 60% : Devoir Surveillé (DS) lors de la dernière séance de cours

Bibliographie

Les supports de cours sont largement hérités de :

- Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart. *Web Data Management*, Cambridge University Press, 2011. <http://webdam.inria.fr/Jorge/>
- Jure Leskovec, Anand Rajaraman, Jeff Ullman. *Mining of Massive Datasets*, Cambridge University Press, 2011. <http://mmds.org/>
- Yves Roos. *Langages Avancés pour les Bases de Données*, Université Lille 1, 2016.
- George H. L. Fletcher. *Data Engineering*, TU Eindhoven, 2016.
- Volker Markl. *Big Data Management*, TU Berlin, 2016.

Contenu du cours et Planning prévisionnel

- Introduction (CM1)
- XML, DTD, XML Schema (CM2, CM3, TD1, TD2) → **Rendu 1**
- Requêtes XPath (CM4, TD3, TD4) → **Rendu 2**
- Bases de données orientées graphes (CM5, TD5, TD6) → **Rendu 3**
- Calcul parallèle avec MapReduce (CM6, CM7, TD7, TD8) → **Rendu 4**
- **Devoir Surveillé (CM8)**

Big data

Every day 2.5 quintillion bytes of data are generated.¹

As of August 2015, there are 3.2 billion people online.

Every minute of every day in 2015²

- 1,736,111 photos were liked on Instagram
- 51,000 apps were downloaded from the Apple store
- 110,040 calls were made on Skype
- 347,222 tweets were sent on Twitter
- 4,166,667 posts were liked on Facebook
- ...

¹ <http://www-01.ibm.com/software/data/bigdata/what-is-big-data.html>

² <https://www.domo.com/blog/2015/08/data-never-sleeps-3-0/>

Big data

Our ability to generate and capture data only continues to explode.

Indeed, we live in a world that is

- increasingly driven by this generation and consumption of massive ever-growing data sets (leading to an emerging “second economy”),
- which in turn is driven by both technology and a rapidly increasing “datafication” of all aspects of our public and private lives.

Managing this **Big data** is now central to commerce, entertainment, government, education, ...

... and, is a core research issue in the emerging discipline of **Data science**.

Big data & Data science

Due to datafication and technology trends, applications now must face data scale, heterogeneity, and distribution as **the norm**, rather than as exceptions, as in the past

- often captured as the four V's of "volume", "variety", "veracity", and "velocity"

Furthermore, there is increased potential for extracting significant untapped **value from data**

- e.g., the successes of Google and Facebook are essentially built on extracting value from data

Big data: NoSQL systems

However, there is increased difficulty in extracting “value” (a fifth V), due to the other four V’s

In data management, embracing and confronting these challenges has led to an explosion of **NoSQL** systems, as alternatives to the dominant paradigm of structured data (i.e., “SQL” stores)

Big data: NoSQL systems

Serious relaxations in “traditional” data management assumptions include

- structural heterogeneity
 - ▶ semi-structured, denormalized data
 - ▶ query paradigms for such loosely structured data
- uncertain data
- looser data consistency
- highly dynamic, evolving, streaming data

NoSQL systems typically embrace one or more of these

Big data: NoSQL systems

Let's consider data **structural heterogeneity**, driven by the need for so-called “polyglot persistence”

Much modern data doesn't cleanly map to a tight relational structure

- missing or multi-valued attributes
 - ▶ e.g., not all site visitors have exactly one phone number
- nested/hierarchical structure
 - ▶ ontologies
 - ▶ XML, JSON
- loose structure
 - ▶ social networks
 - ▶ chem-/bio- networks

Big data: NoSQL systems

Four broad classes of approaches taken by “NoSQL” systems:

- ① **key-value databases** (essentially scalable hash tables)
 - ▶ example: Oracle BerkeleyDB, Redis, ...
- ② **document databases** (generalization of key-value model to include nested structure, e.g., XML and JSON)
 - ▶ example: MongoDB, Apache CouchDB, Couchbase, BaseX, ...
- ③ **column-family stores** (hybrid of key-value and relational model, where rows can have different schemas)
 - ▶ example: Apache Cassandra, Apache HBase, ...
- ④ **graph databases**
 - ▶ example: Neo4j, Virtuoso, Apache Jena, ...

All of these are firmly rooted in applications arising in Web data management... The explosion of the Web was both a precursor to and accelerant for the rise of Big data.

Introduction

Web Data Management and Distribution

Serge Abiteboul Ioana Manolescu Philippe Rigaux
Marie-Christine Rousset Pierre Senellart



Web Data Management and Distribution
<http://webdam.inria.fr/textbook>

June 12, 2012

Web data handling

Web data = by far the **largest** information system ever seen, and a fantastic means of sharing information.

- Billions of textual documents, images, PDF, multimedia files, provided and updated by millions of institutions and individuals.
- An anarchical process which results in highly heterogeneous data organization, steadily growing and extending to meet new requirements.
- New usage and applications appear every day: yesterday P2P file sharing, today social networking, tomorrow ?

The challenge, under a data management perspective: master the size and extreme variability of Web information, and make it **usable**.

The role of XML

Web data management has been primarily based on HTML, which describes presentation.

- HTML is appropriate for humans: allows sophisticated output and interaction with textual documents and images;
- HTML falls short when it comes to software exploitation of data.

XML describes content, and promotes machine-to-machine communication and data exchange

- XML is a generic data format, apt to be specialized for a wide range of fields,
⇒ (X)HTML is a specialized XML dialect for data presentation
- XML makes easier data integration, since data from different sources now share a common format;
- XML comes equipped with many software products, APIs and tools.

Perspective of the course

The course develops an XML perspective of the management of heterogeneous data (e.g., Web data) in a distributed environment.

We introduce **models, languages, architectures** and **techniques** to fulfill the following goals:

- **flexible data representation and retrieval**

XML is viewed as a new **data model**, both more powerful and more flexible than the relational one

- **data exchange and integration**

XML data can be serialized and restructured for better exchange between systems, and integration of multiple data sources

- **efficient distributed computing and storage**

XML can be the glue for high-level description of distributed repositories; this calls for efficient storage, indexing of management.

Semi-structured data model

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 a heterogeneous environment.

Self-describing data

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

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

Natural extension: values may themselves be other structures:

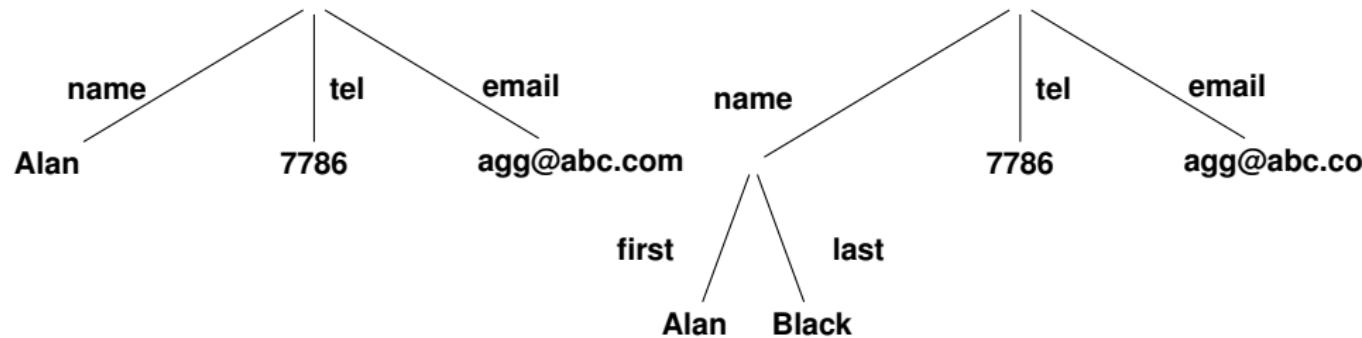
```
{name: {first: "Alan", last: "Black"},  
tel: 2157786,  
email: "agb@abc.com"}
```

Further extension: allow duplicate labels.

```
{name: "alan'", tel: 2157786, tel: 2498762 }
```

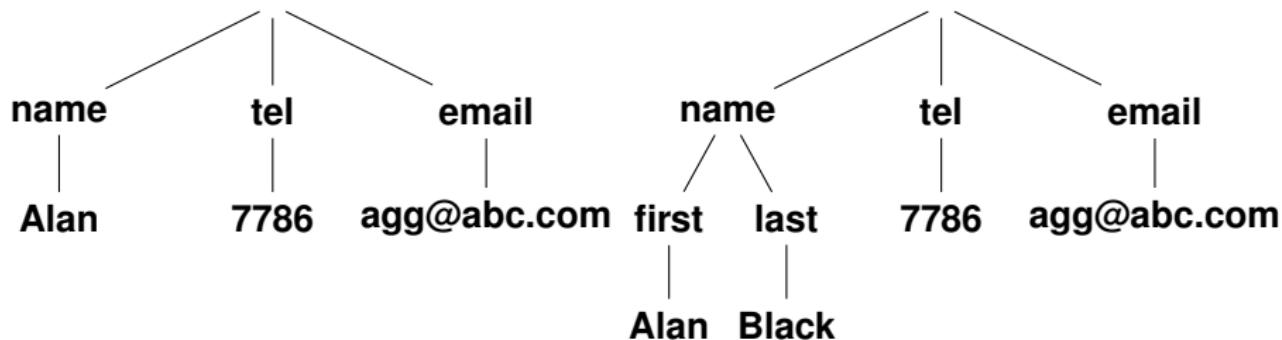
Tree-based representation

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



Tree-based representation: labels as nodes

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



Remark

The XML data model adopts this latter representation.

Representation of regular data

The syntax 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.

Representation of irregular data

Many possible variations in the structure: missing values, duplicates, 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}}
```

Node identity

Nodes can be **identified**, and referred to by their identity. Cycles and objects models can be described as well.

XML in brief

XML 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, see further)
- The W3C promotes companion standards: DOM (object model), XSchema (typing), XPath (path expression), XSLT (restructuring), XQuery (query language), and many others.

Remark

1. XML is a simplified version of **SGML**, a long-term used language for technical documents.
2. HTML, up to version 4.0, is **also** a variant of SGML. The successor of HTML 4.0, is **XHTML**, an XML dialect.

XML documents

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.

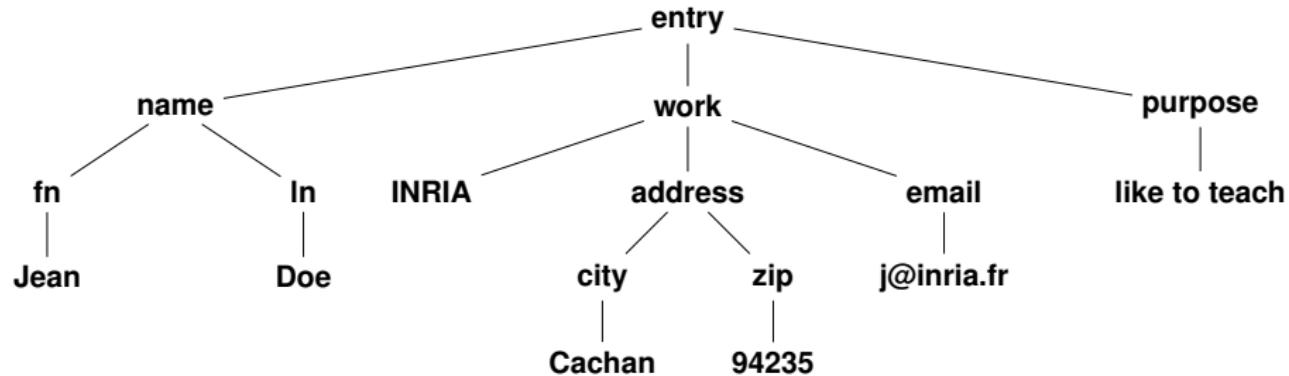
Ordered means that there is an order between the children of each node.

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).

XML documents are trees

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).

Serialized representation of XML document

Documents can be serialized, such as, for instance:

```
<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>
```

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>
```

XML describes structured content

Applications cannot interpret unstructured content:

The book ``Fundations of Databases'', written by Serge Abiteboul, Rick 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.

Applications associate semantics to XML docs

The description of a letter

Letter document

```
<letter>
  <header>
    <author>...</author>
    <date>...</date>
    <recipient>...</recipient>
    <cc>...<cc>
  </header>
  <body>
    <text>...</text>
    <signature>...</signature>
  </body>
</letter>
```

Applications associate semantics to XML docs (2)

Letter style sheet

- if *letter* then ...
- if *header* then ...
- if *author* then ...
- if *date* then ...
- if *recipient* then ...
- if *cc* then ...
- if *body* then ...
- if *text* then ...
- if *signature* then ...

Some software then produces the actual letter to mail or email.

Outline

1 Preliminaries

2 XML, a semi-structured data model

3 XML syntax

- Essential XML Syntax
- XML Syntax: Complements

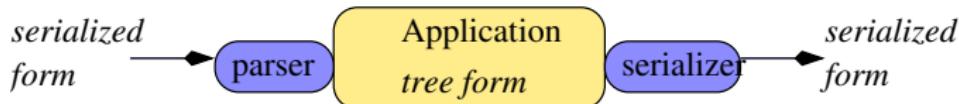
4 Typing

5 The XML World

6 Use cases

Serialized form, tree form

Typically, an application gets a document in *serialized form*, parse it in *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.

The syntax for serialized document, in brief

Four examples of XML documents (separated by blank lines) are:

```
<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 *prologue*, useful for XML parsers (it gives in particular the document encoding).

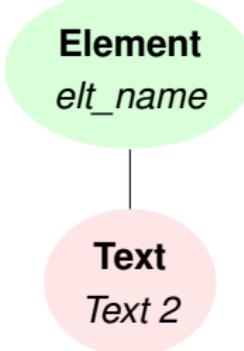
From serialized to tree form: text and elements

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**.



From serialized to tree form: nesting elements

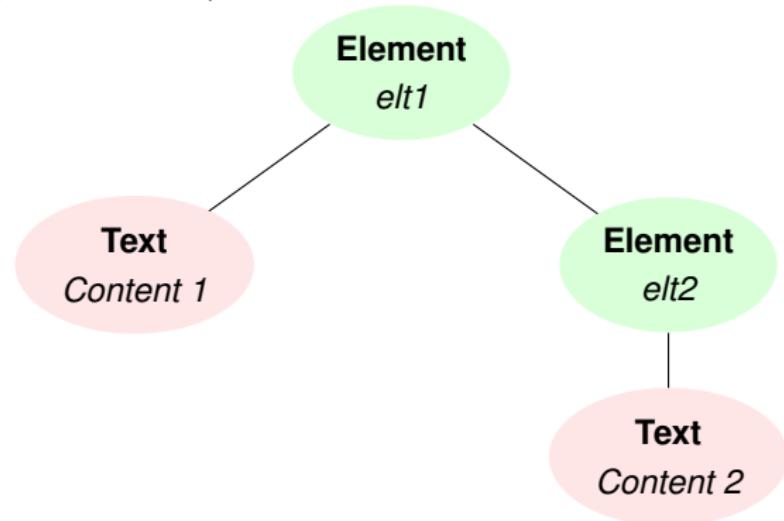
The content of an element is

- 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>
  Textual content
  <elt2>
    Another content
  </elt2>
</elt1>
```



From serialized to tree form: attributes

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

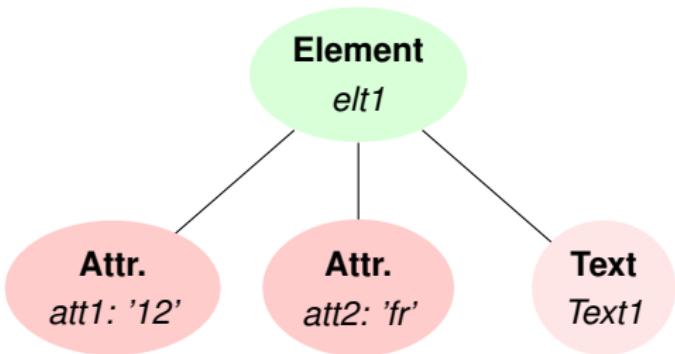
- ① as part of the opening tag in the serialized form,
- ② as special child nodes of the **Element** node (in DOM).

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

An element with two attributes.

```
<elt1 att1='12' att2='fr'>  
    Textual content  
</elt1>
```

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



From serialized to tree form: the document root

The first line of the serialized form must *always* be the *prologue* if there is one:

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

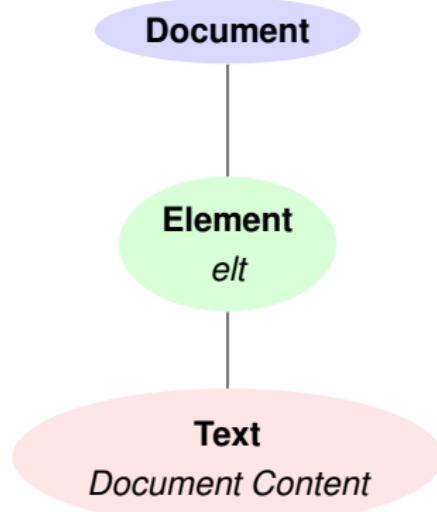
and the document content must *always* be enclosed in a single opening/ending tag, called the *element root*.

A document with its prologue, and element root.

```
<?xml version="1.0"  
       encoding="utf-8" ?>  
<elt>  
    Document content.  
</elt>
```

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

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



Summary: syntax and vocabulary

Serialized form

- 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. See the lecture notes.

Summary: syntax and semantics

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 and references

Entities are used for the physical organization of a document.

An entity is **declared** (in the document type), 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>
```

Predefined entities

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 "<">	<	<
<!ENTITY gt ">">	>	>
<!ENTITY amp "&">	&	&
<!ENTITY apos "'">	'	'
<!ENTITY quot """>	"	"

Comments and instructions

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"?>
```

Literal sections

Usually, the content of an element is *parsed* to analyse its structure.

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

```
<?xml version='1.0'?>
<program>
if ((i < 5) && (j > 6))
    printf("error");
</program>
```

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

```
<?xml version='1.0'?>
<program>
<! [CDATA[if ((i < 5) && (j > 6))
    printf("error");
]]>
</program>
```

Outline

- 1 Preliminaries
- 2 XML, a semi-structured data model
- 3 XML syntax
- 4 Typing
- 5 The XML World
- 6 Use cases

To type or not to type

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.

Type declaration

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

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

Document Type Definition

An example of valid document (the root element matches the DOCTYPE declaration).

```
<?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>
```

Document Type Definition

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.)

Interpreting labels: Namespaces

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 vs. XML schema

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)>
```

DTD vs. XML schema

The same structure in XML schema (an XML dialect)

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

Outline

- 1 Preliminaries
- 2 XML, a semi-structured data model
- 3 XML syntax
- 4 Typing
- 5 The XML World
- 6 Use cases

Popular XML dialects

- RSS** is an XML dialect for describing content updates that is heavily used for blog entries, news headlines or podcasts.
- WML** (Wireless Mark-up Language) is used in Web sites by wireless applications based on the Wireless Application Protocol (WAP).
- MathML** (Mathematical Mark-up Language) is an XML dialect for describing mathematical notation and capturing both its structure and content.
- Xlink** (XML Linking Language) is an XML dialect for defining hyperlinks between XML documents. These links are expressed in XML and may be introduced inside XML documents.
- SVG** (Scalable Vector Graphics) is an XML dialect for describing two-dimensional vector graphics, both static and animated. With SVG, images may contain outbound hyperlinks in XLinks.

XML standards

SAX (Simple API for XML) sees an XML document as a sequence of tokens (its serialization).

DOM (Document Object Model) is an object model for representing (HTML and) XML document independently of the programming language.

XPath (XML Path Language) that we will study, is a language for addressing portions of an XML document.

XQuery (that we will study) is a flexible query language for extracting information from collections of XML documents.

XSLT (Extensible Stylesheet Language Transformations), that we will study, is a language for specifying the transformation of XML documents into other XML documents.

Web services provide interoperability between machines based on Web protocols.

Zoom: DOM

Document Object Model - DOM

Document model that is tree-based

Used in APIs for different programming languages (e.g. Java)

Object-oriented access to the content:

- Root of the document
- First child of a node (with a particular label)
- Next one (with a particular label)
- Parent of a node
- Attribute of a node...

Zoom: XPATH

Language for expressing **paths** in an XML document

- Navigation: child, descendant, parent, ancestor
- Tests on the nature of the node
- More complex selection predicates

Means to specify portions of a document

Basic tool for other XML languages: Xlink, XSLT, Xquery

Zoom: XLINK

XML Linking Language

Advanced hypertext primitives

Allows inserting in XML documents descriptions of links to external Web resources

Simple monodirectional links ala (HREF) HTML

Mulridirectional links

XLink relies on XPath for addressing portions of XML documents

Remark

XML trees + XLINK \Rightarrow graph

Zoom: XSLT

Transformation language: “Perl for XML”

An XSLT style sheet includes a set of transformation rules: pattern/template

Pattern: based on XPATH expressions; it specifies a structural context in the tree

Template: specifies what should be produced

Principle: when a pattern is matched in the source document, the corresponding templates produces some data

Zoom: XQuery

Query language: “SQL for XML”

Like SQL: select portions of the data and reconstruct a result

Query structure: **FLW** (pronounced "flower")

Exemple

```
FOR $p IN document("bib.xml")//publisher  
LET $b := document("bib.xml")//book[publisher = $p]  
WHERE count($b) > 100  
RETURN $p
```

\$p : scans the sequence of publishers

\$b : scans the sequence of books for a publisher

WHERE filters out some publishers

RETURN constructs the result

Generic XML tools

API

Parsers and type checkers

GUI (Graphical User Interfaces)

Editors

XML diff

XML wiki

Etc.

Facilitate the development of applications specific to particular XML dialects.

Outline

- 1 Preliminaries
- 2 XML, a semi-structured data model
- 3 XML syntax
- 4 Typing
- 5 The XML World
- 6 Use cases

Exploiting XML content

Publishing: an XML document can easily be converted to another XML document (same content, but another structure)

⇒ **Web publishing** is the process of transforming XML documents to XHTML.

Integration: XML documents from many sources can be transformed in a common dialect, and constitute a **collection**.

⇒ **Search engines**, or **portals**, provide browsing and searching services on collections of XML documents.

Distributed Data Processing: many softwares can be adapted to consume/produce XML-represented data.

⇒ **Web services** provide remote services for XML data processing.

Genericity of softwares and APIs

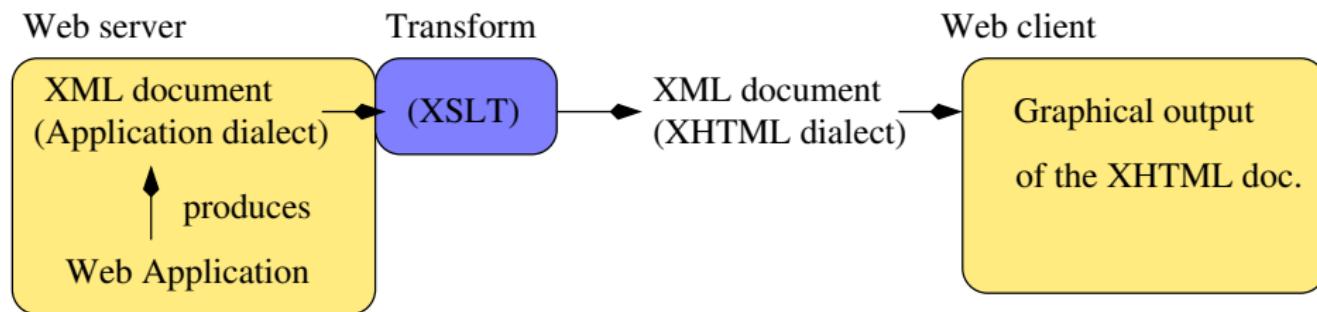
XML comes with many tools that are **generic**: A software or API for XML documents applies to **all** the possible XML dialects.

Web Publishing: restructuring to XHTML

The **Web application** produces some XML content, structured in some application-dependent dialect, on the server.

In a second phase, the XML content is transformed in an XHTML document that can be visualized by humans.

The transformation is typically expressed in XSLT, and can be processed either on the server or on the client.



Web publishing: content + presentation instructions

The following document is an XHTML version of the bibliographic content presented above:

```
<h1> Bibliography </h1>
<p> <i> Foundations of Databases </i>
    Abiteboul, Hull, Vianu
    <br/> Addison Wesley, 1995 </p>
<p> <i> Data on the Web </i>
    Abiteboul, Buneman, Suciu
    <br/> Morgan Kaufmann, 1999 </p>
```

Now the labels belong to the XHTML dialect, and can be interpreted by a Web browser.

[Test: biblio.html](#)

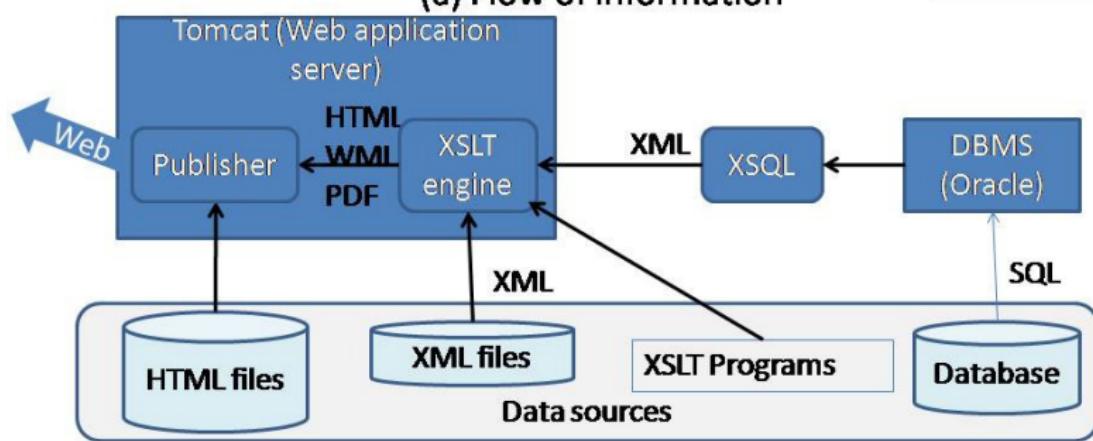
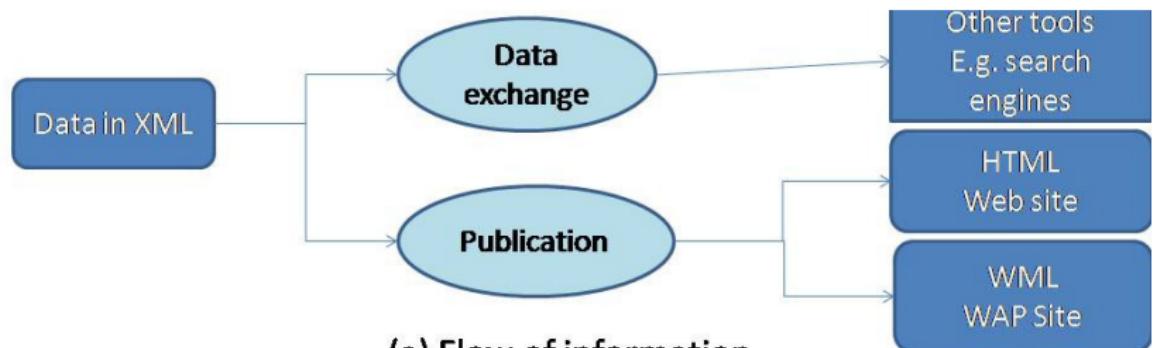
Note that the “meaning” of labels is restricted to presentation purposes. It becomes complicated for a software to distinguish the name of authors.

Web publishing

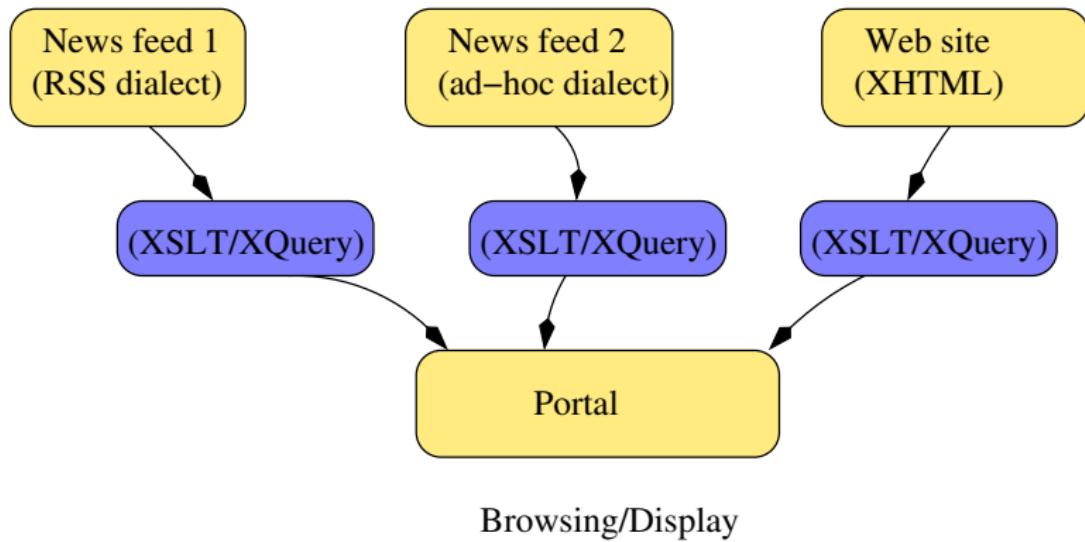
The same content may be published using different means:

- Web publishing: XML \Rightarrow XHTML
- WAP (Wireless Application Protocol): XML \Rightarrow WML

Web publishing, the big picture



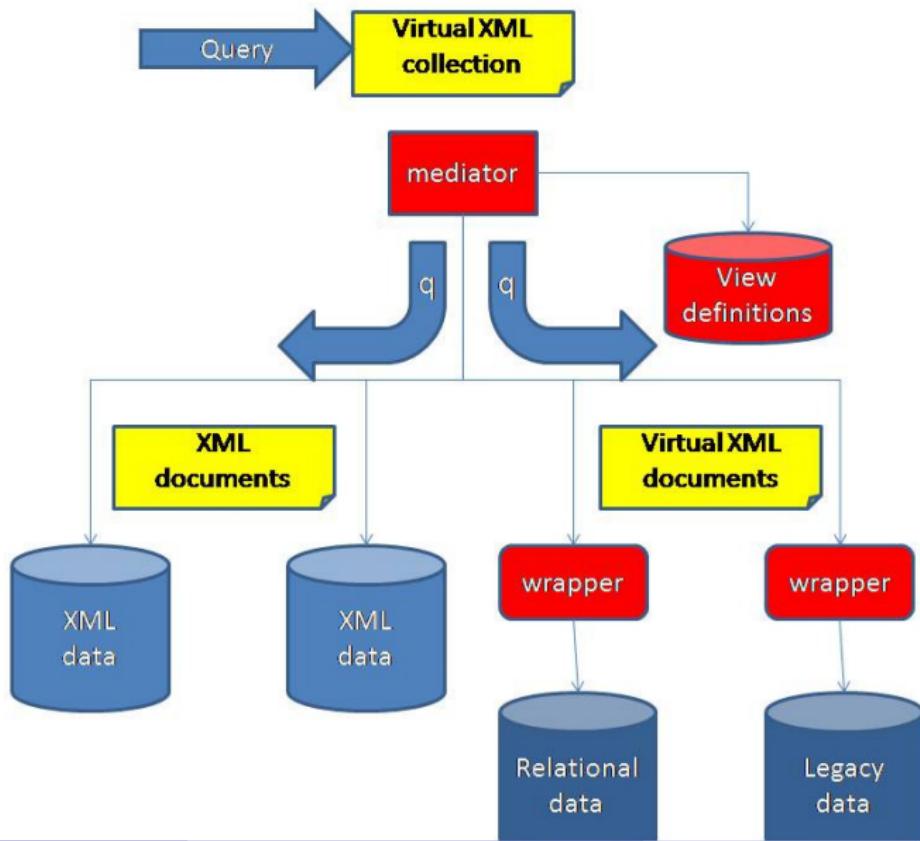
Web Integration: gluing together heterogeneous sources



The **portal** receives (possibly continuously) XML-structured content, each source using its own dialect.

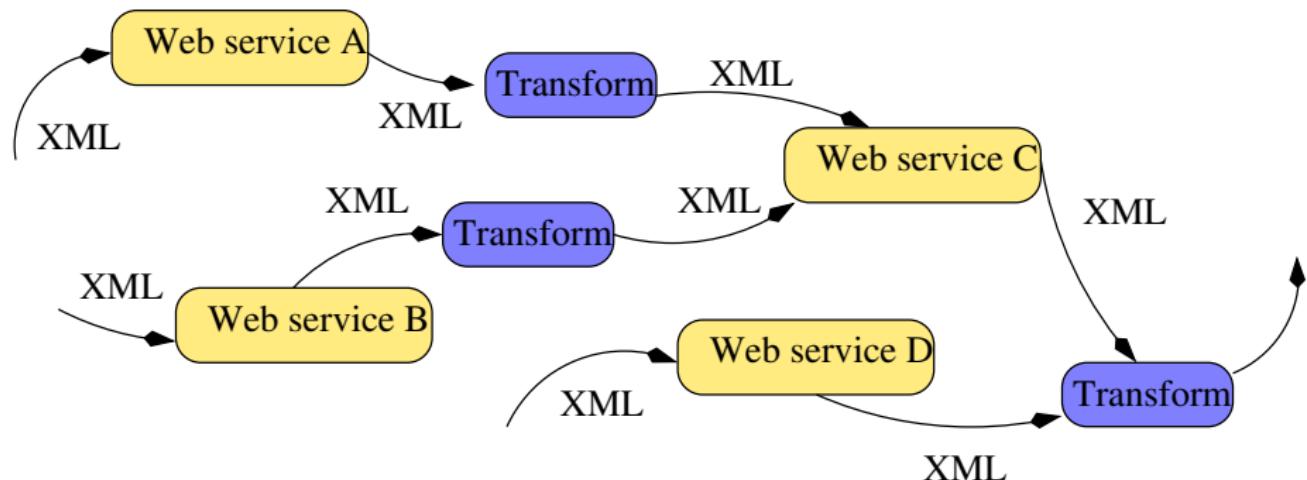
Each feed provides some content, extracted with XSLT or XQuery, or any convenient XML processing tool (e.g., SAX).

Data integration, a larger perspective



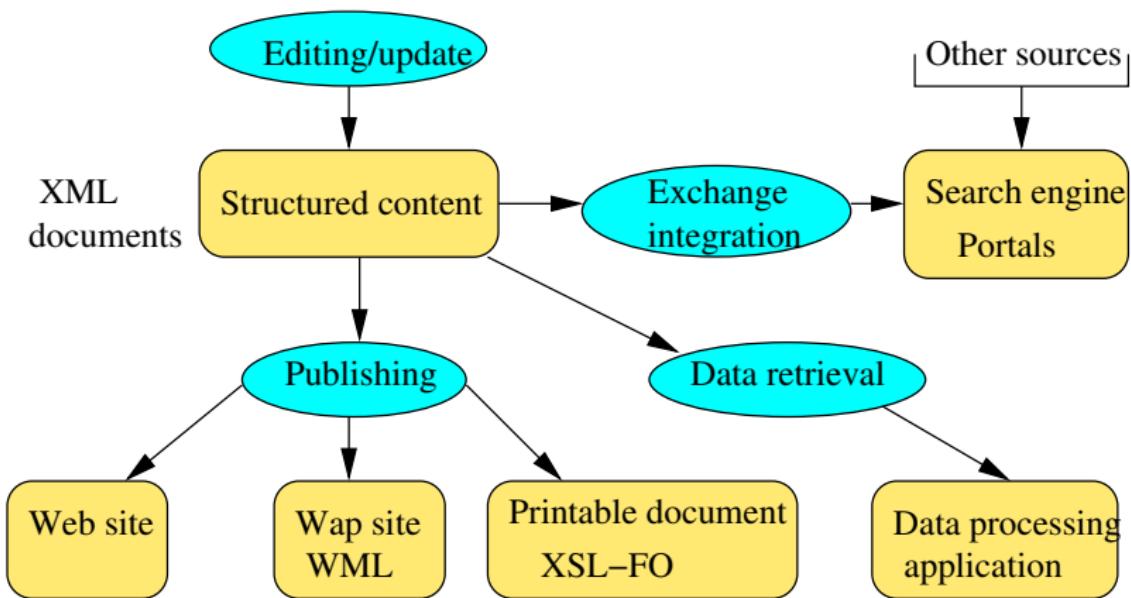
Distributed Data Management with XML

XML encoding is used to *exchange* information between applications.



A specific dialect, WDSL, is used to describe Web Services Interfaces.

Exploiting XML documents: the big picture



... and many other possible use of XML flexibility.

Définir des DTD

DTD

= **grammaire** pour la structure des documents

= un ensemble de **règles**,

chacune d'entre-elles décrivant le **contenu autorisé** d'un élément ou l'ensemble des attributs existant pour un élément.

Validation d'un fichier XML

A partir du moment où une **DTD** est associée au document à valider, on peut **valider** à l'aide :

- d'un **logiciel spécialisé** dans le traitement des documents **XML** (*XMLSpy, Editix, Eclipse, ...*)
- par **programme** en utilisant les bibliothèques de traitement de **XML** disponibles dans beaucoup de langages (*java, php, perl, ...*)

Structure d'une DTD

Une DTD contient :

- des déclarations d'**éléments**,
 - des déclarations d'**attributs**,
 - des déclarations d'**entités**,
 - des **commentaires**
- <!-- comme dans les documents XML -->

Déclaration d'élément

<!ELEMENT nom modèle>

- **ELEMENT** (en **majuscule**) est un mot clef,
- **nom** est un nom **valide** d'élément,
- **modèle** est le **modèle de contenu** de l'élément.

vide l'élément n'a pas de contenu (mais peut avoir des attributs)

libre le contenu de l'élément est un contenu quelconque bien formé

données l'élément contient du texte

éléments l'élément est composé d'autre éléments (ses fils)

mixte l'élément contient un mélange de texte et de sous-éléments

Déclaration d'élément

<!ELEMENT nom EMPTY>

- ELEMENT (en majuscule) est un mot clef,
- nom est un nom valide d'élément,
- modèle est le **modèle de contenu** de l'élément.

vide l'élément n'a pas de contenu (mais peut avoir des attributs)
libre le contenu de l'élément est un contenu quelconque bien formé
données l'élément contient du texte
éléments l'élément est composé d'autre éléments (ses fils)
mixte l'élément contient un mélange de texte et de sous-éléments

Déclaration d'élément

<!ELEMENT nom ANY>

- ELEMENT (en majuscule) est un mot clef,
- nom est un nom valide d'élément,
- modèle est le **modèle de contenu** de l'élément.

vide l'élément n'a pas de contenu (mais peut avoir des attributs)

libre le contenu de l'élément est un contenu quelconque bien formé

données l'élément contient du texte

éléments l'élément est composé d'autre éléments (ses fils)

mixte l'élément contient un mélange de texte et de sous-éléments

Déclaration d'élément

```
<!ELEMENT nom (#PCDATA)>
```

- ELEMENT (en majuscule) est un mot clef,
- nom est un nom valide d'élément,
- **modèle** est le **modèle de contenu** de l'élément.

vide l'élément n'a pas de contenu (mais peut avoir des attributs)

libre le contenu de l'élément est un contenu quelconque bien formé

données l'élément contient du texte

éléments l'élément est composé d'autre éléments (ses fils)

mixte l'élément contient un mélange de texte et de sous-éléments

Déclaration d'élément

<!ELEMENT nom modèle>

- **ELEMENT** (en **majuscule**) est un mot clef,
- **nom** est un nom **valide** d'élément,
- **modèle** est le **modèle de contenu** de l'élément.

vide l'élément n'a pas de contenu (mais peut avoir des attributs)

libre le contenu de l'élément est un contenu quelconque bien formé

données l'élément contient du texte

éléments l'élément est composé d'autre éléments (ses fils)

mixte l'élément contient un mélange de texte et de sous-éléments

Modèle de contenu d'élément

On définit le contenu à l'aide d'une **expression régulière** de sous-éléments :

- **séquence**

```
<!ELEMENT chapitre (titre,intro,section)>
```

- **choix**

```
<!ELEMENT chapitre (titre,intro,(section|sections))>
```

- **indicateurs d'occurrence** * (0-n) + (1-n) ? (0-1)

```
<!ELEMENT chapitre (titre,intro?,section+)>
```

```
<!ELEMENT section (titre-section,texte-section)+>
```

```
<!ELEMENT texte-section (p|f)*>
```

Modèle de contenu d'élément

La syntaxe précise des expressions régulières de sous-éléments est :

- cp ::= (**Name** | choice | seq) ('?' | '*' | '+')?
- seq ::= '(' cp (',' cp)* ')'
- choice ::= '(' cp ('|' cp)+ ')'

Contenu mixte

Une seule façon de mélanger texte **#PCDATA** et des sous-éléments est acceptée : **#PCDATA** doit être le premier membre d'un choix placé sous une étoile.

```
<!ELEMENT p (#PCDATA | em | exposant | indice | renvoi ) *>
```

Exemple

```
<!ELEMENT catalogue ( stage )*>
<!ELEMENT stage ( intitule , prerequis ?)>
<!ELEMENT intitule(#PCDATA)>
<!ELEMENT prerequis (#PCDATA | xref )*>
<!ELEMENT xref EMPTY>
```

Exemple

```
<catalogue>
  <stage>
    <intitule>XML et les bases de données</intitule>
    <prerequis>
      connaitre les langages SQL et HTML
    </prerequis>
  </stage>
  <stage>
    <intitule>XML programmation</intitule>
    <prerequis>
      avoir suivi le stage de XML et les bases de données
    </prerequis>
  </stage>
</catalogue>
```

Exemple

```
<catalogue>
  <stage>
    <intitule>XML et les bases de données</intitule>
    </stage>
  <stage>
    <intitule>XML programmation</intitule>
    <prerequis>
      avoir suivi le stage de XML et les bases de données
    </prerequis>
  </stage>
</catalogue>
```

Exemple

```
<catalogue>
  <stage>
    <intitule>XML et les bases de données</intitule>
    <prerequis>
      connaitre les <xref/> et <xref/>
    </prerequis>
  </stage>
  <stage>
    <intitule>XML programmation</intitule>
    <prerequis>
      avoir suivi le stage de XML et les bases de données
    </prerequis>
  </stage>
</catalogue>
```

Exemple

```
<catalogue>
  <stage>
    <intitule>XML et les bases de données</intitule>
    <prerequis>
      connaitre les <xref> langages SQL et HTML </xref>
    </prerequis>
  </stage>
  <stage>
    <intitule>XML programmation</intitule>
    <prerequis>
      avoir suivi le stage de XML et les bases de données
    </prerequis>
  </stage>
</catalogue>
```

Exemple

```
<catalogue>
</catalogue>
```

La mystérieuse propriété UPA

Unique Particle Attribution

Dans le cas où le contenu d'un élément est défini sous la forme d'une expression régulière, celle-ci doit respecter la règle UPA.

La mystérieuse propriété UPA

Définition du W3C

A content model must be formed such that during validation of an element information item sequence, the particle component contained directly, indirectly or implicitly therein with which to attempt to validate each item in the sequence in turn can be uniquely determined without examining the content or attributes of that item, and without any information about the items in the remainder of the sequence.

Exemple

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE stage SYSTEM "./dtd1.dtd">
<stage>
  <intitule>T1</intitule>
  <prerequis>T2</prerequis>
</stage>

<!-- dtd1.dtd ci-dessous -->

<!ELEMENT stage ((intitule*| prerequis),(intitule*| prerequis)*)>
<!ELEMENT intitule (#PCDATA)>
<!ELEMENT prerequis (#PCDATA)>
```

Exemple

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE stage SYSTEM "./dtd2.dtd">
<stage>
  <intitule>T1</intitule>
  <prerequis>T2</prerequis>
</stage>

<!-- dtd2.dtd ci-dessous -->

<!ELEMENT stage (intitule*| prerequis)+>
<!ELEMENT intitule (#PCDATA)>
<!ELEMENT prerequis (#PCDATA)>
```

La mystérieuse propriété UPA

Une **vraie** définition

*Une expression régulière de sous-éléments satisfait la propriété UPA si et seulement si son **automate de Glushkov** est déterministe.*

Automate de Glushkov (rappel ?)

Construction de cet automate pour (a , (a | b)* , b)

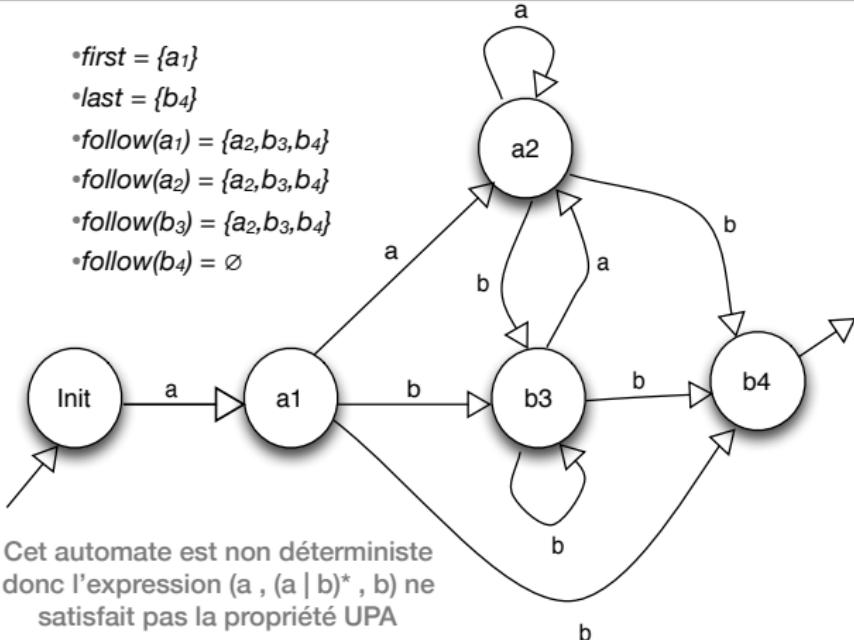
1. distinguer dans l'expression toutes les occurrences des mêmes noms d'élément:
(a_1 , (a_2 | b_3)* , b_4).

2. Calculer les ensembles

- **first** = ensemble des lettres qui peuvent apparaître au début des mots qui matchent l'expression $\text{first} = \{a_1\}$
- **last** = ensemble des lettres qui peuvent apparaître à la fin des mots qui matchent l'expression $\text{last} = \{b_4\}$
 - pour chaque lettre x calculer son ensemble **follow** qui contient les lettres qui peuvent apparaître immédiatement après x dans au moins un des mots qui matchent l'expression $\text{follow}(a_1) = \{a_2, b_3, b_4\}$, $\text{follow}(a_2) = \{a_2, b_3, b_4\}$, $\text{follow}(b_3) = \{a_2, b_3, b_4\}$, $\text{follow}(b_4) = \emptyset$
 - **c'est tout**, on en déduit alors directement l'automate.

$$(a_1, (a_2 \mid b_3)^*, b_4)$$

- $first = \{a_1\}$
- $last = \{b_4\}$
- $follow(a_1) = \{a_2, b_3, b_4\}$
- $follow(a_2) = \{a_2, b_3, b_4\}$
- $follow(b_3) = \{a_2, b_3, b_4\}$
- $follow(b_4) = \emptyset$



Cet automate est non déterministe
donc l'expression $(a, (a \mid b)^*, b)$ ne
satisfait pas la propriété UPA

Unique Particle Attribution

En définitive

La très grande majorité des logiciels de validation se moquent complètement de savoir si la DTD satisfait ou non la propriété UPA, mais comme certains validateurs exploitent cette propriété dans leur algorithme de validation, dans le cas d'une DTD qui ne satisfait pas cette propriété,

- *certaines validateurs fonctionnent parfaitement*
- *d'autres donnent des réponses erronées !*

Déclarations d'attributs

```
<!ATTLIST element nom-attribut1 type1 default1  
          nom-attribut2 type2 default2  
          ...>
```

Le type d'un attribut définit les valeurs qu'il peut prendre

- **CDATA** : valeur chaîne de caractères,
- **ID**, **IDREF**, **IDREFS** permettent de définir des références à l'intérieur du document,
- Une **liste de choix** possibles parmi un ensemble de noms symboliques.

Déclarations d'attributs

```
<!ATTLIST element nom-attribut1 type1 default1  
          nom-attribut2 type2 default2  
          ...>
```

La déclaration par défaut peut prendre quatre formes :

- la valeur par défaut de l'attribut,
- **#REQUIRED** indique que l'attribut est obligatoire,
- **#IMPLIED** indique que l'attribut est optionnel,
- **#FIXED valeur** indique que l'attribut prend toujours la même valeur, dans toute instance de l'élément **si l'attribut y apparaît**.

Exemples de déclarations d'attributs

```
<!ATTLIST document version CDATA "1.0">
```

```
<document version="1.0" >  
  ...  
</document>
```

OK

Exemples de déclarations d'attributs

```
<!ATTLIST document version CDATA "1.0">
```

```
<document version="2.0" >  
  ...  
</document>
```

OK

Exemples de déclarations d'attributs

```
<!ATTLIST document version CDATA "1.0">
```

```
<document>
  ...
</document>
```

OK

Exemples de déclarations d'attributs

```
<!ATTLIST document version CDATA #FIXED "1.0">
```

```
<document version="1.0" >  
  ...  
</document>
```

OK

Exemples de déclarations d'attributs

```
<!ATTLIST document version CDATA #FIXED "1.0">
```

```
<document version="2.0" >  
  ...  
</document>
```

KO

Exemples de déclarations d'attributs

```
<!ATTLIST document version CDATA #FIXED "1.0">
```

```
<document>
  ...
</document>
```

OK

Exemples de déclarations d'attributs

```
<!ATTLIST nom  
        titre (Mlle|Mme|M.) #REQUIRED  
        nom-epouse CDATA #IMPLIED  
>
```

```
<nom titre="Mme" nom-epouse="Lenoir">  
    Martin  
</nom>
```

OK

Exemples de déclarations d'attributs

```
<!ATTLIST nom
        titre (Mlle|Mme|M.) #REQUIRED
        nom-epouse CDATA #IMPLIED
    >
```

```
<nom titre="M." nom-epouse="Lenoir">
    Martin
</nom>
```

OK

Exemples de déclarations d'attributs

```
<!ATTLIST nom
        titre (Mlle|Mme|M.) #REQUIRED
        nom-epouse CDATA #IMPLIED
    >
```

```
<nom titre="M.">
    Martin
</nom>
```

OK

Exemples de déclarations d'attributs

```
<!ATTLIST nom  
        titre (Mlle|Mme|M.) #REQUIRED  
        nom-epouse CDATA #IMPLIED  
>
```

```
<nom titre="Madame" nom-epouse="Lenoir">  
    Martin  
</nom>
```

KO

Attributs ID, IDREF, IDREFS

- Un attribut **ID** sert à référencer un élément, la valeur de cette référence pouvant être rappelée dans des attributs **IDREF** ou **IDREFS**.
- Un élément ne peut avoir au plus qu'un attribut **ID** et la valeur associée doit être unique dans le document XML. Cette valeur doit être un **nom XML** (donc pas un nombre).
- La valeur de défaut pour un attribut **ID** est obligatoirement **#REQUIRED** ou **#IMPLIED**
- Une valeur utilisée dans un attribut **IDREF** ou **IDREFS** doit obligatoirement correspondre à celle d'un attribut **ID**.

Exemples ID, IDREF, IDREFS

```
<!ELEMENT document (personne*,livre*)>
<!ELEMENT personne (nom , prenom)>
  <!ATTLIST personne id ID #REQUIRED>
<!ELEMENT nom (#PCDATA)>
<!ELEMENT prenom (#PCDATA)>
<!ELEMENT livre (#PCDATA)>
  <!ATTLIST livre auteur IDREF #IMPLIED>
```

document.dtd

Exemples ID, IDREF, IDREFS

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE document SYSTEM "document.dtd">
<document>
  <personne id="id-1">
    <nom>Dupond</nom>
    <prenom>Martin</prenom>
  </personne>
  <personne id="id-2">
    <nom>Durand</nom>
    <prenom>Helmut</prenom>
  </personne>
  <livre auteur="id-1">Ma vie, mon oeuvre</livre>
</document>
```

OK

Exemples ID, IDREF, IDREFS

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE document SYSTEM "document.dtd">
<document>
    <personne id="id-1">
        <nom>Dupond</nom>
        <prenom>Martin</prenom>
    </personne>
    <personne id="id-1">
        <nom>Hardailepick</nom>
        <prenom>Helmut</prenom>
    </personne>
    <livre auteur="id-1">Ma vie, mon oeuvre</livre>
</document>
```

KO

Exemples ID, IDREF, IDREFS

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE document SYSTEM "document.dtd">
<document>
    <personne id="id-1">
        <nom>Gaisellepick</nom>
        <prenom>Elmer</prenom>
    </personne>
    <personne id="id-2">
        <nom>Hardailepick</nom>
        <prenom>Helmut</prenom>
    </personne>
    <livre auteur="id-3">Ma vie, mon oeuvre</livre>
</document>
```

KO

Exemples ID, IDREF, IDREFS

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE document SYSTEM "document.dtd">
<document>
    <personne id="id-1">
        <nom>Gaisellepick</nom>
        <prenom>Elmer</prenom>
    </personne>
    <personne id="id-2">
        <nom>Hardailepick</nom>
        <prenom>Helmut</prenom>
    </personne>
    <livre auteur="id-1 id-2">Ma vie, mon oeuvre</livre>
</document>
```

KO

Exemples ID, IDREF, IDREFS

```
<!ELEMENT document (personne*,livre*)>
<!ELEMENT personne (nom , prenom)>
  <!ATTLIST personne id ID #REQUIRED>
<!ELEMENT nom (#PCDATA)>
<!ELEMENT prenom (#PCDATA)>
<!ELEMENT livre (#PCDATA)>
  <!ATTLIST livre auteur IDREFS #IMPLIED>
```

document.dtd

Exemples ID, IDREF, IDREFS

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE document SYSTEM "document.dtd">
<document>
    <personne id="id-1">
        <nom>Gaisellepick</nom>
        <prenom>Elmer</prenom>
    </personne>
    <personne id="id-2">
        <nom>Hardailepick</nom>
        <prenom>Helmut</prenom>
    </personne>
    <livre auteur="id-1 id-2">Ma vie, mon oeuvre</livre>
</document>
```

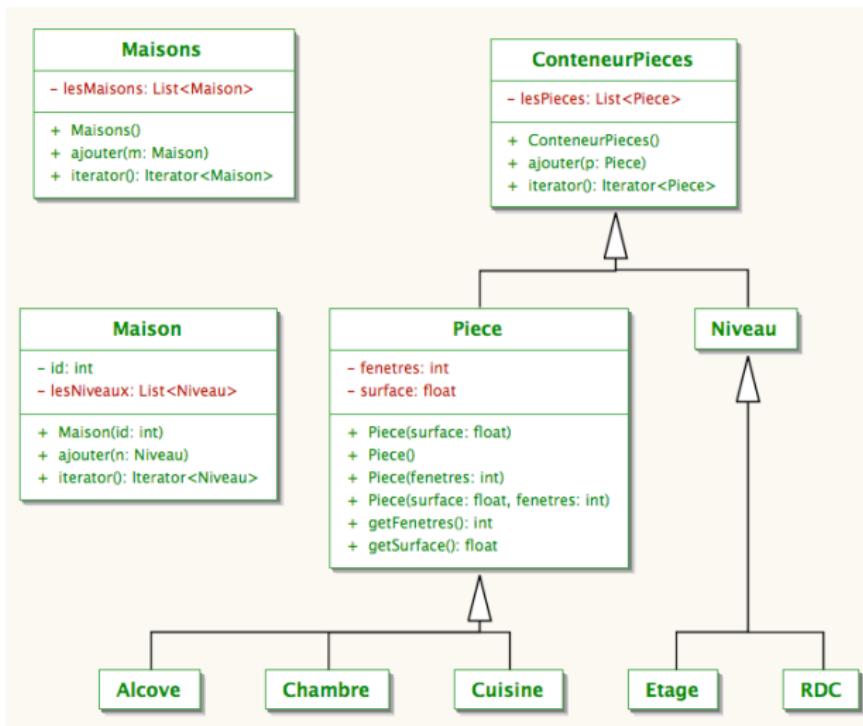
OK

XML Schema

maisons.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<maisons>
    <maison id="1">
        <RDC>
            <cuisine surface-m2="12">Evier Inox. Mobilier encastré</cuisine>
            <WC>Lavabo.</WC>
            <séjour surface-m2="38">Cheminée en pierre. Baie vitrée</séjour>
            <bureau surface-m2="14">Bibliothèque</bureau>
            <garage/>
        </RDC>
        <étage>
            <terrasse/>
            <chambre surface-m2="28" fenetre="3">
                <alcove surface-m2="8"/>
            </chambre>
            <chambre surface-m2="18"/>
            <salledeBain surface-m2="15">Douche, baignoire, lavabo</salledeBain>
        </étage>
    </maison>
    <maison id="2">
        <RDC>
            <cuisine surface-m2="12">en ruine</cuisine>
            <garage/>
        </RDC>
        <étage>
            <mirador surface-m2="1">Vue sur la mer</mirador>
            <salledeBain surface-m2="15">Douche</salledeBain>
        </étage>
    </maison>
    <maison id="3">
```

Typage



Comparaison entre DTD et XML Schema

- DTD

- Essentiellement, définition de l'**imbrication des éléments**, et **définition des attributs**.
- Types pauvres
- Pas de gestion des espaces de nom
- Pas beaucoup de contraintes sur le contenu d'un document

- XML-Schema

- Notion de **type**, indépendamment de la notion d'élément
- **Contraintes d'intégrité d'entité et d'intégrité référentielle**, plus précises que les ID/IDREF des DTD.
- Contraintes de **cardinalité**
- Gestion des **espaces de noms**
- **Réutilisation** de mêmes types d'attributs pour des éléments différents
- Format **XML**

Lier un schéma à un document

Un peu comme pour une DTD

La **balise ouvrante de l'élément racine** du fichier XML contient des informations sur le schéma.

```
<maisons xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:noNamespaceSchemaLocation="maisons.xsd">
```

Lier un schéma à un document

Le schéma `maisons.xsd` est lui-même un fichier XML et doit donc être associé au schéma qui définit ce qu'on peut utiliser dans un schéma !

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <xsd:element name="maisons">
        ...
    </xsd:element>
    ...
</xsd:schema>
```

Contenu d'un schéma

Un XML-Schema est composé de

- Définitions de **types**
- Déclaration **d'attributs**
- Déclaration **d'éléments**
- Définitions de **groupes d'attributs**
- Définitions de **groupes de modèles**
- Définitions de **contraintes d'unicité ou de clés**
- ...

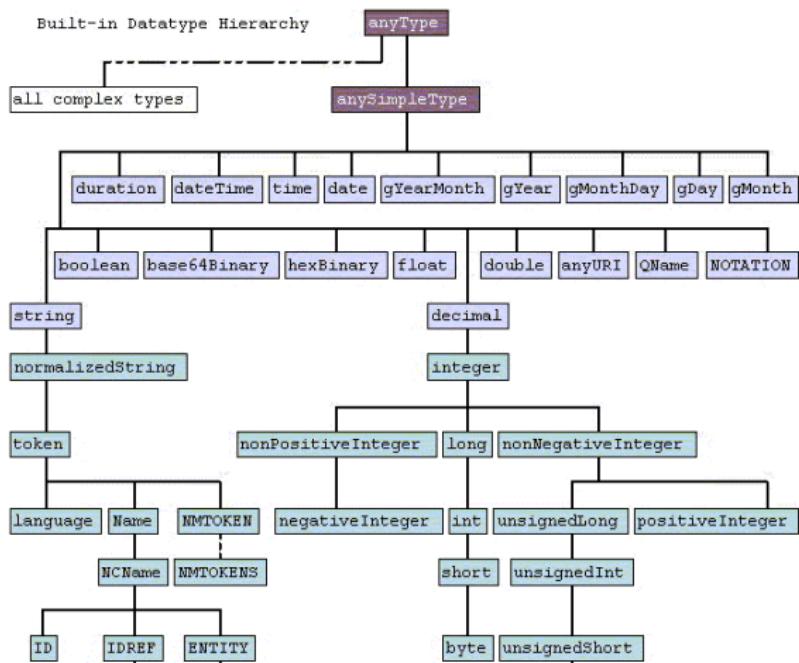
Définitions de types

Typage

- Existence de **types prédéfinis** : hiérarchie de types, dont la racine est le type **anyType**.
- Possibilité de définir de **nouveaux types**
- Distinction **types simples** et **types complexes**
 1. Les types simples sont utilisés pour les déclarations **d'attributs**, les déclarations d'**éléments** dont le **contenu** se limite à des **données atomiques**, et qui n'ont **pas d'attributs**.
 2. Les types complexes s'utilisent dans tous les autres cas.

Les types simples prédéfinis

<http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes#built-in-datatypes>



Exemple d'utilisation d'un type simple prédéfini

```
<xsd:attribute name="surface-m2" type="xsd:decimal"/>  
  
<xsd:attribute name="fenetre" type="xsd:positiveInteger"/>
```

Les types simples

Un type simple est caractérisé par 3 ensembles : son **espace de valeurs**, son **espace lexical**, ses **facettes**.

1.espace de valeurs = les **valeurs autorisées** pour ce type

2.espace lexical = **syntaxe des littéraux**. Par exemple **100** et **1.0E2** sont deux littéraux qui représentent la même valeur, de type **float**.

3.facette = **propriété disponible sur ce type**. Toutes les facettes ne s'appliquent pas à tous les types.

Facettes de contrainte

Les facettes de contraintes sont optionnelles et permettent de restreindre l'espace des valeurs. Elles ne sont pas toutes disponibles sur tous les types.

1.**length** : **longueur**, qui peut être le nombre de caractères pour un type **string**, le nombre d'éléments pour un type **list**, le nombre d'octets pour un type binaire.

2.**maxLength** : longueur maximale

3.**minLength** : longueur minimale

4.**pattern** : expression régulière qui **décrit les littéraux** du type (donc les valeurs)

5.**maxExclusive** : valeur **maximale au sens strict** ($<$)

6.**maxInclusive** : valeur **maximale au sens large** (\leq)

7.**minExclusive** : valeur **minimale au sens strict**

8.**minInclusive** : valeur **minimale au sens large**

Facettes de contrainte

9.enumeration : énumération des valeurs possibles

10.fractionDigits : nombre maximal de décimales après le point

11.totalDigits : nombre maximal de chiffres pour une valeur décimale

12.whiteSpace : règle pour la normalisation des espaces dans une chaîne

Les types simples créés par l'utilisateur

On dérive un type simple **à partir d'un autre type**. Il existe **3** façons de dériver un type simple :

- 1.par **restriction** : on crée un type dont l'espace de valeurs est inclus dans l'espace de valeurs d'un type existant. Pour cela, **on utilise les facettes** pour restreindre l'espace des valeurs,

- 2.par **liste**,

- 3.par **union**.

Exemples (XHTML) de restrictions d'un type atomique

```
<xs:simpleType name="tabindexNumber">
  <xs:restriction base="xs:nonNegativeInteger">
    <xs:minInclusive value="0" />
    <xs:maxInclusive value="32767" />
  </xs:restriction>
</xs:simpleType>
```

Exemples de restrictions d'un type atomique

```
<xsd:simpleType name="jour-de-la-semaine">
  <xsd:restriction base="xsd:token">
    <xsd:enumeration value="lundi"/>
    <xsd:enumeration value="mardi"/>
    <xsd:enumeration value="mercredi"/>
    <xsd:enumeration value="jeudi"/>
    <xsd:enumeration value="vendredi"/>
    <xsd:enumeration value="samedi"/>
    <xsd:enumeration value="dimanche"/>
  </xsd:restriction>
</xsd:simpleType>
```

Exemples (XHTML) de restrictions d'un type atomique

```
<!-- single or comma-separated list of media descriptors -->
<xs:simpleType name="MediaDesc">
  <xs:restriction base="xs:string">
    <xs:pattern value="[^,]+(, \s*[^\n, ]+)*"/>
  </xs:restriction>
</xs:simpleType>
```

Exemple de type union

```
<!-- permet de faire <font size="34"> ou <font size="medium">-->

<xsd:simpleType name="fontSize">
  <xsd:union>
    <xsd:simpleType>
      <xsd:restriction base="xsd:positiveInteger">
        <xsd:minInclusive value="8"/>
        <xsd:maxInclusive value="72"/>
      </xsd:restriction>
    </xsd:simpleType>
    <xsd:simpleType>
      <xsd:restriction base="xsd:NMTOKEN">
        <xsd:enumeration value="small"/>
        <xsd:enumeration value="medium"/>
        <xsd:enumeration value="large"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:union>
</xsd:simpleType>
```

Exemples de types list

```
<xsd:simpleType name="sizes">
  <xsd:list itemType="xsd:decimal"/>
</xsd:simpleType>
```

Permet d'écrire <dimensions>12 5.6 67</dimensions>

```
<xsd:simpleType name='listOfString'>
  <xsd:list itemType='xsd:string'/>
</xsd:simpleType>
```

Attention, la liste ci-dessous, de type **listOfString**, a 18 éléments, pas 3 (l'espace est un séparateur)

```
<someElement>
this is not list item 1
this is not list item 2
this is not list item 3
</someElement>
```

Exemple de restriction d'un type list

```
<xs:simpleType name='myList'>
  <xs:list itemType='xs:integer' />
</xs:simpleType>

<xs:simpleType name='myRestrictedList'>
  <xs:restriction base='myList'>
    <xs:pattern value='123 (\d+\s)*456' />
  </xs:restriction>
</xs:simpleType>

<someElement>123 456</someElement>
<someElement>123 987 456</someElement>
<someElement>123 987 567 456</someElement>
```

Exercice XML Schema – Types simples

Définir les types simples suivants en les dérivant à partir des types simples prédéfinis les plus pertinents :

- ① une heure comprise entre 2h30 (du matin) et 16h50;
- ② un nombre réel en précision simple, supérieur ou égal à -3476.4 et strictement inférieur à 5;
- ③ une chaîne de quatre caractères;
- ④ une chaîne de caractères qui ne peut être égale qu'à "jpg", "gif" ou "png";
- ⑤ un type de numéro ISBN : c'est un entier à 13 chiffres comme 9782744072369.

Les types complexes

Type complexe

- La définition d'un type complexe est un **ensemble de déclarations d'attributs** et un **type de contenu**.
- Type complexe **à contenu simple** : type d'un élément dont le contenu est de **type simple mais qui possède un attribut** (un type simple n'a pas d'attribut). On le définit par extension d'un type simple par ajout d'un attribut.
- Type complexe **à contenu complexe** : permet de déclarer **des sous-éléments et des attributs**. On le construit à partir de rien ou on le définit par **restriction** d'un type complexe, ou par **extension** d'un type.

Exemple de type complexe à contenu simple

```
<xs:complexType name="TypePiece">
  <xs:simpleContent>
    <xs:extension base="xsd:string">
      <xsd:attribute name="surface-m2" type="xsd:decimal"/>
      <xsd:attribute name="fenetre" type="xsd:positiveInteger"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>

<!-- exemple de déclaration d'élément chambre
                  de type TypePiece -->
<xs:element name="chambre" type="TypePiece"/>

<!-- exemple d'élément chambre -->
<chambre surface-m2="17.5">Exposition plein sud</chambre>
```

Exemples de types complexes à contenu complexe (1)

Construit à partir de zéro.

```
<xsd:complexType name="type-duree">
  <xsd:sequence>
    <xsd:element name="du" type="xsd:date"/>
    <xsd:element name="au" type="xsd:date"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:element name="duree" type="type-duree"/>

<duree>
  <du>2010-09-13</du>
  <au>2010-09-25</au>
</duree>
```

Exemples de types complexes à contenu complexe (1)

Construit à partir de zéro.

```
<xs:complexType name="personName">
  <xs:sequence>
    <xs:element name="title" minOccurs="0" />
    <xs:element name="forename" minOccurs="0"
                maxOccurs="unbounded" />
    <xs:element name="surname" />
  </xs:sequence>
</xs:complexType>
```

Exemple de type complexe à contenu complexe (2)

Obtenu par extension d'un type simple ou complexe

```
<xs:complexType name="extendedName">
  <xs:complexContent>
    <xs:extension base="personName">
      <xs:sequence>
        <xs:element name="generation" minOccurs="0"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:element name="addressee" type="extendedName"/>

<addressee>
  <forename>Albert</forename>
  <forename>Arnold</forename>
  <surname>Gore</surname>
  <generation>Jr</generation>
</addressee>
```

Groupe de Modèles

Pour construire un type complexe, on peut utiliser des **constructeurs de groupes de modèles**. On a déjà rencontré le constructeur **sequence**, il existe 3 constructeurs de groupes de modèles :

- **sequence** : les éléments d'une séquence doivent apparaître **dans l'ordre** où ils sont déclarés.
- **choice** : **un seul** élément **parmi ceux du choice** doit apparaître
- **all** : les éléments contenus dans un **all** peuvent apparaître dans **n'importe quel ordre**.

Exemple d'utilisation de **choice**

```
<xsd:complexType name="SurfaceOuVolume">
  <xsd:choice>
    <xsd:element name="surface" type="length0"/>
    <xsd:element name="volume" type="length0"/>
  </xsd:choice>
</xsd:complexType>

<!-- element occupation du type SurfaceouVolume -->
<occupation>
  <surface>453</surface>
</occupation>
```

Exemple d'utilisation de all

```
<xsd:complexType name="Identite">
  <xsd:all>
    <xsd:element name="nom" type="xsd:string"/>
    <xsd:element name="prenom" type="xsd:string"/>
    <xsd:element name="datenaiss" type="xsd:date"/>
  </xsd:all>
</xsd:complexType>
```

Exemple d'utilisation de all

```
<!-- elements de type Identite -->
<identite>
    <nom>meurisse</nom>
    <prenom>paul</prenom>
    <datenaiss>1912-12-21</datenaiss>
</identite>

<identite>
    <datenaiss>1948-05-31</datenaiss>
    <nom>bonham</nom>
    <prenom>john</prenom>
</identite>
```

Déclaration d'attribut

- Un **attribut** est de **type simple**, par exemple un type prédéfini, une énumération, une liste ...
- On peut **préciser le caractère obligatoire ou facultatif** de l'attribut (**attribut use**)
- On peut lui donner une **valeur par défaut** (**attribut default**) ou une **valeur fixe** (**attribut fixed**)

Exemples de déclarations d'attributs

```
<xsd:attribute name="size"
    type="fontSize"
    use="required"/>

<xs:attribute name="jour" default="lundi"
    type="jour-de-la-semaine"/>

<xs:attribute name="version"
    type="xs:number"
    fixed="1.0"/>
```

Déclaration d'élément

- On définit le contenu de l'élément grâce aux types
- Lorsqu'on n'attribue **pas de type** à un élément, il est considéré comme de type **xs:anyType** et peut donc contenir n'importe quoi
- Pour un **élément de contenu mixte** (texte et sous-éléments), on utilise l'attribut **mixed** de **xs:complexType**.
- Pour un **élément vide**, on définit un type complexe **qui n'a pas de sous-élément**

Exemple 1 : éléments de contenu simple

```
<xsd:element name="long" type="length0"/>

<xsd:element name="nom" type="xsd:string"/>

<xsd:element name="font">
  <xsd:complexType>
    <xsd:simpleContent>
      <xsd:extension base="xsd:string">
        <xsd:attribute name="size" type="fontSize"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
</xsd:element>
```

Exemple 1 : éléments de contenu simple

Permet d'écrire dans le document XML

```
<long unit="cm">45</long>  
  
<nom>durand</nom>  
  
<font size="medium">machin</font>
```

Eléments, Attributs et Types

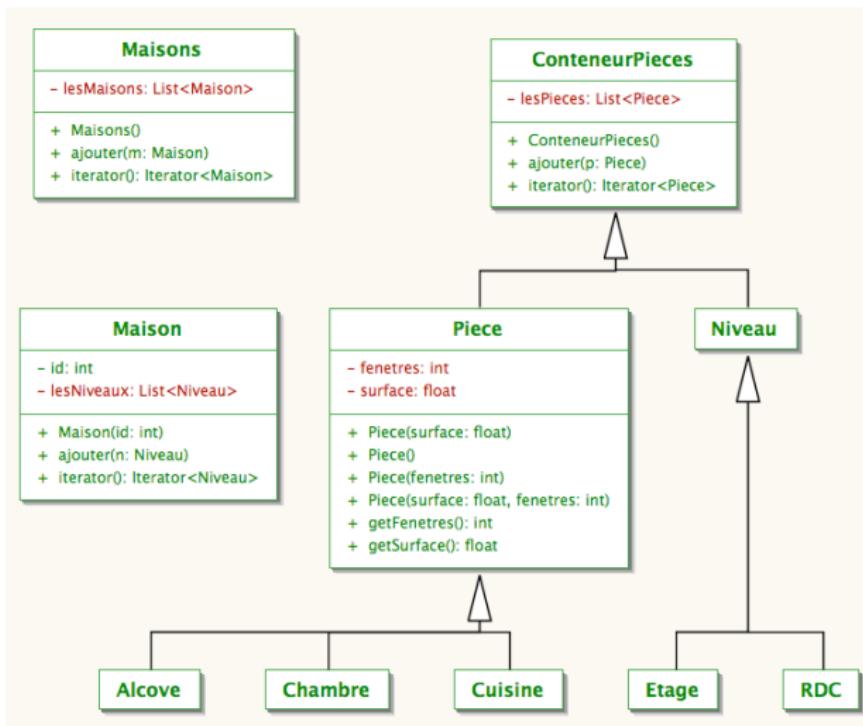
- On doit associer les types et les noms d'éléments ou d'attributs.
- Un type peut contenir des déclarations d'éléments ou d'attributs
- Un élément ou attribut peut contenir une déclaration de type
- On distingue : déclaration locale (dans une autre déclaration) ou globale (fils de la racine `xs:schema`)
- On peut faire référence à un type, élément, attribut déjà défini grâce à l'attribut `ref`

Eléments, Attributs et Types

```
<xsd:element name="trimestre">
  <xsd:complexType>
    <xsd:complexContent>
      <xsd:extension base="type-duree">
        <xsd:attribute name="num" type="type-trimestre"
                      use="required"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:element>

<xsd:element name="trimestres">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="trimestre" minOccurs="4" maxOccurs="4"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Typage



maisons.xsd

```
<xsd:element name="maison">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:choice minOccurs="0" maxOccurs="unbounded">
        <xsd:element name="étage" type="TypeNiveau"/>
        <xsd:element name="RDC" type="TypeNiveau"/>
      </xsd:choice>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:positiveInteger"
      use="required"/>
  </xsd:complexType>
</xsd:element>

<xsd:element name="maisons">
  <xsd:complexType>
    <xsd:sequence minOccurs="0" maxOccurs="unbounded">
      <xsd:element ref="maison"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

maisons.xsd

```
<xsd:complexType name="TypeNiveau">
  <xsd:sequence>
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:element name="alcove" type="TypePiece"/>
      <xsd:element name="cuisine" type="TypePiece"/>
      <xsd:element name="séjour" type="TypePiece"/>
      <xsd:element name="bureau" type="TypePiece"/>
      <xsd:element name="garage" type="TypePiece"/>
      <xsd:element name="terrasse" type="TypePiece"/>
      <xsd:element name="chambre" type="TypePiece"/>
      <xsd:element name="salledeBain" type="TypePiece"/>
      <xsd:element name="mirador" type="TypePiece"/>
      <xsd:element name="WC" type="TypePiece"/>
    </xsd:choice>
  </xsd:sequence>
</xsd:complexType>
```

maisons.xsd

```
<xsd:complexType name="TypePiece" mixed="true">
  <xsd:complexContent>
    <xsd:extension base="TypeNiveau">
      <xsd:attribute name="surface-m2" type="xsd:decimal"/>
      <xsd:attribute name="fenetre" type="xsd:positiveInteger"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Impossible car un type dont l'attribut **mixed** vaut **true** ne peut pas étendre un type dont l'attribut **mixed** vaut **false**

maisons.xsd

```
<xsd:complexType name="TypePiece" mixed="true">
  <xsd:sequence>
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:element name="alcove" type="TypePiece"/>
      <xsd:element name="cuisine" type="TypePiece"/>
      <xsd:element name="séjour" type="TypePiece"/>
      <xsd:element name="bureau" type="TypePiece"/>
      <xsd:element name="garage" type="TypePiece"/>
      <xsd:element name="terrasse" type="TypePiece"/>
      <xsd:element name="chambre" type="TypePiece"/>
      <xsd:element name="salledeBain" type="TypePiece"/>
      <xsd:element name="mirador" type="TypePiece"/>
      <xsd:element name="WC" type="TypePiece"/>
    </xsd:choice>
  </xsd:sequence>
  <xsd:attribute name="surface-m2" type="xsd:decimal"/>
  <xsd:attribute name="fenetre" type="xsd:positiveInteger"/>
</xsd:complexType>
```

Lourd et redondant avec le type TypeNiveau

xsd:group

```
<xsd:group name="TypeConteneurPieces">
  <xsd:sequence>
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:element name="alcove" type="TypePiece"/>
      <xsd:element name="cuisine" type="TypePiece"/>
      <xsd:element name="séjour" type="TypePiece"/>
      <xsd:element name="bureau" type="TypePiece"/>
      <xsd:element name="garage" type="TypePiece"/>
      <xsd:element name="terrasse" type="TypePiece"/>
      <xsd:element name="chambre" type="TypePiece"/>
      <xsd:element name="salledeBain" type="TypePiece"/>
      <xsd:element name="mirador" type="TypePiece"/>
      <xsd:element name="WC" type="TypePiece"/>
    </xsd:choice>
  </xsd:sequence>
</xsd:group>
```

Utilisation d'un groupe de modèle nommé

xsd:group

```
<xsd:complexType name="TypeNiveau">
  <xsd:group ref="TypeConteneurPieces"/>
</xsd:complexType>

<xsd:complexType name="TypePiece" mixed="true">
  <xsd:group ref="TypeConteneurPieces"/>
  <xsd:attribute name="surface-m2" type="xsd:decimal"/>
  <xsd:attribute name="fenetre" type="xsd:positiveInteger"/>
</xsd:complexType>
```

Utilisation d'un groupe de **modèle nommé**

maisons.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:group name="TypeConteneurPieces">
    <xsd:sequence>
      <xsd:choice minOccurs="0" maxOccurs="unbounded">
        <xsd:element name="alcove" type="TypePiece"/>
        <xsd:element name="cuisine" type="TypePiece"/>
        <xsd:element name="séjour" type="TypePiece"/>
        <xsd:element name="bureau" type="TypePiece"/>
        <xsd:element name="garage" type="TypePiece"/>
        <xsd:element name="terrasse" type="TypePiece"/>
        <xsd:element name="chambre" type="TypePiece"/>
        <xsd:element name="salledeBain" type="TypePiece"/>
        <xsd:element name="mirador" type="TypePiece"/>
        <xsd:element name="WC" type="TypePiece"/>
      </xsd:choice>
    </xsd:sequence>
  </xsd:group>
```

maisons.xsd

```
<xsd:complexType name="TypeNiveau">
    <xsd:group ref="TypeConteneurPieces"/>
</xsd:complexType>

<xsd:complexType name="TypePiece" mixed="true">
    <xsd:group ref="TypeConteneurPieces"/>
    <xsd:attribute name="surface-m2" type="xsd:decimal"/>
    <xsd:attribute name="fenetre" type="xsd:positiveInteger"/>
</xsd:complexType>

<xsd:element name="maisons">
    <xsd:complexType>
        <xsd:sequence minOccurs="0" maxOccurs="unbounded">
            <xsd:element ref="maison"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

maisons.xsd

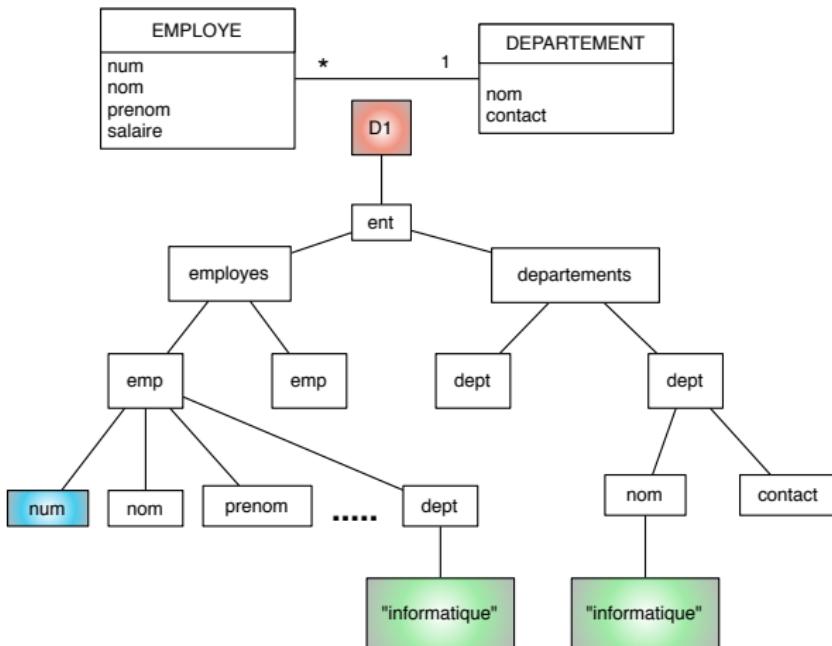
```
<xsd:element name="maison">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:choice minOccurs="0" maxOccurs="unbounded">
        <xsd:element name="étage" type="TypeNiveau"/>
        <xsd:element name="RDC" type="TypeNiveau"/>
      </xsd:choice>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:positiveInteger"
      use="required"/>
  </xsd:complexType>
</xsd:element>
</xsd:schema>
```

Contraintes d'intégrité en XML-Schema

- Modèle relationnel :
 - Contrainte d'unicité **UNIQUE**, de **clef primaire** (unicité et existence) **PRIMARY KEY**, de **clef étrangère FOREIGN KEY ... REFERENCES**
 - Une clef primaire est définie pour une relation, et elle composée d'un ou plusieurs attributs.
 - Une clef étrangère fait référence à des attributs d'une relation précise.
- Avec une DTD, on peut définir des identifiants (attribut de type **ID**), et des références d'identifiant (de type **IDREF**). L'existence ou non est définie en choisissant **IMPLIED** ou **REQUIRED**. Mais
 - les références ne sont pas typées (on ne sait pas à quel type de nœud on fait référence)
 - un **identifiant est global** au document
- XML-Schema : on se rapproche du modèle relationnel

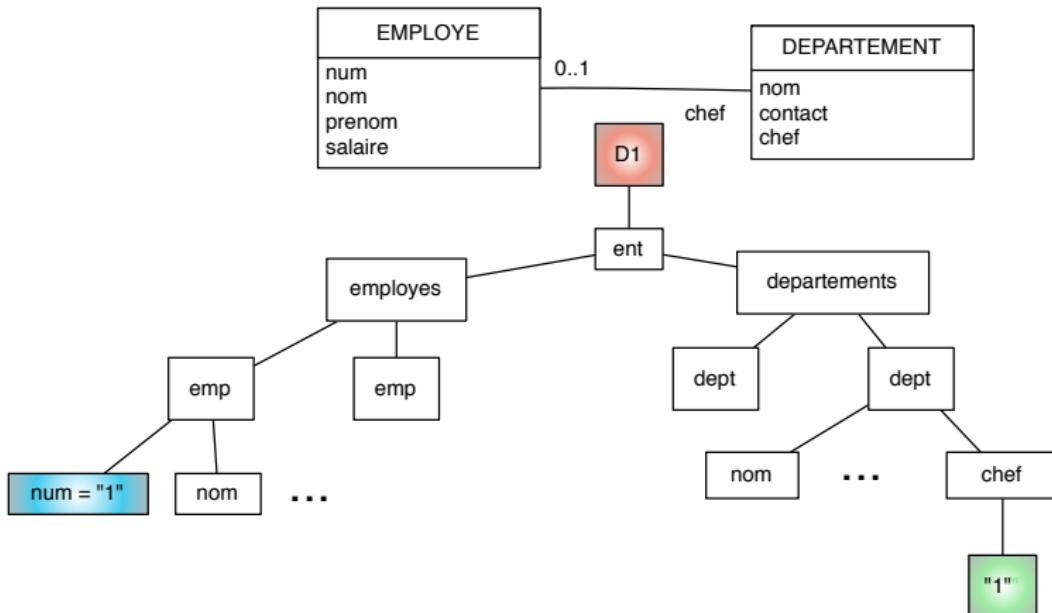
Exemple

Tout employé appartient à un département :



Exemple (suite)

Un département a au plus un chef qui est employé par l'entreprise



Exemple (suite)

On déclare un type pour un département, et pour une séquence de départements

```
<xsd:complexType name="TypeDept">
  <xsd:sequence>
    <xsd:element name="nom" type="xsd:string"/>
    <xsd:element name="contact" type="xsd:string"/>
    <xsd:element name="chef" type="xsd:int" minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="SeqDept">
  <xsd:sequence>
    <xsd:element name="dept" type="TypeDept" maxOccurs="unbounded"
      minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>
```

Exemple (suite)

Même chose pour les employés.

```
<xsd:complexType name="TypeEmploye">
  <xsd:sequence>
    <xsd:element name="nom" type="xsd:string"/>
    <xsd:element name="prenom" type="xsd:string"/>
    <xsd:element name="salaire" type="xsd:decimal"/>
    <xsd:element name="dept" type="xsd:string"/>
  </xsd:sequence>
  <xsd:attribute name="num" type="xsd:int"/>
</xsd:complexType>
```

Exemple (suite)

```
<xsd:complexType name="SeqEmploye">
  <xsd:sequence>
    <xsd:element name="emp"
      type="TypeEmploye"
      maxOccurs="unbounded"
      minOccurs="0" />
  </xsd:sequence>
</xsd:complexType>
```

Exemple (suite)

Déclaration des éléments **employes** et **departements** avec les clefs primaires pour les éléments **emp** et **dept**.

```
<xsd:element name="employes" type="SeqEmploye">
  <xsd:key name="clefEmp">
    <xsd:selector xpath="emp"/>
    <xsd:field xpath="@num"/>
  </xsd:key>
</xsd:element>

<xsd:element name="departements" type="SeqDept">
  <xsd:key name="clefDept">
    <xsd:selector xpath="dept"/>
    <xsd:field xpath="nom"/>
  </xsd:key>
</xsd:element>
```

Exemple (suite)

L'élément racine du document avec les clefs étrangères.

```
<xsd:element name="ent">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="employes"/>
      <xsd:element ref="departements"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:keyref name="refDept" refer="clefDept">
    <xsd:selector xpath="employes/emp"/>
    <xsd:field xpath="dept"/>
  </xsd:keyref>
  <xsd:keyref name="refEmp" refer="clefEmp">
    <xsd:selector xpath="departements/dept"/>
    <xsd:field xpath="chef"/>
  </xsd:keyref>
</xsd:element>
```

Explications

- L'endroit où l'on déclare une contrainte **détermine la zone où elle doit être vérifiée**. En effet, la contrainte s'applique à l'intérieur de **l'élément "contexte"** de cette contrainte.
- Le **selector** détermine **pour quel élément c'est une clef**. C'est un chemin XPath qui désigne un ensemble de noeuds (appelés noeuds cibles) contenus dans l'élément contexte de la contrainte.
- Les **field** qui suivent donnent **les composants** (attributs ou éléments) **de la clef**. Chaque composant **désigne un noeud unique** (élément ou attribut), par rapport à 1 noeud cible désigné par le **selector**. De plus, chaque composant doit être **de type simple**.

Explications

- La syntaxe des chemins **XPath** est réduite, voir la norme pour plus de précision.
- Dans une clef étrangère, l'attribut **refer** indique **à quelle clef primaire** elle fait référence.
- Pour définir une **contrainte d'unicité**, remplacer **xsd:key** par **xsd:unique**.

XPath

Web Data Management and Distribution

Serge Abiteboul Ioana Manolescu Philippe Rigaux
Marie-Christine Rousset Pierre Senellart



Web Data Management and Distribution
<http://webdam.inria.fr/textbook>

May 30, 2013

XPath

- An **expression language** to be used in another host language (e.g., XSLT, XQuery).
- Allows the description of **paths** in an XML tree, and the retrieval of nodes that match these paths.
- Can also be used for performing some (limited) operations on XML data.

Example

`2 * 3` is an XPath **literal expression**.

`// *[@msg="Hello world"]` is an XPath **path expression**, retrieving all elements with a `msg` attribute set to “Hello world”.

Content of this presentation

Mostly XPath 1.0: a W3C recommendation published in 1999, widely used. Also a *basic* introduction to XPath 2.0, published in 2007.

XPath Data Model

XPath expressions operate over **XML trees**, which consist of the following **node types**:

- **Document**: the **root node** of the XML document;
- **Element**: element nodes;
- **Attribute**: attribute nodes, represented as children of an **Element** node;
- **Text**: text nodes, i.e., leaves of the XML tree.

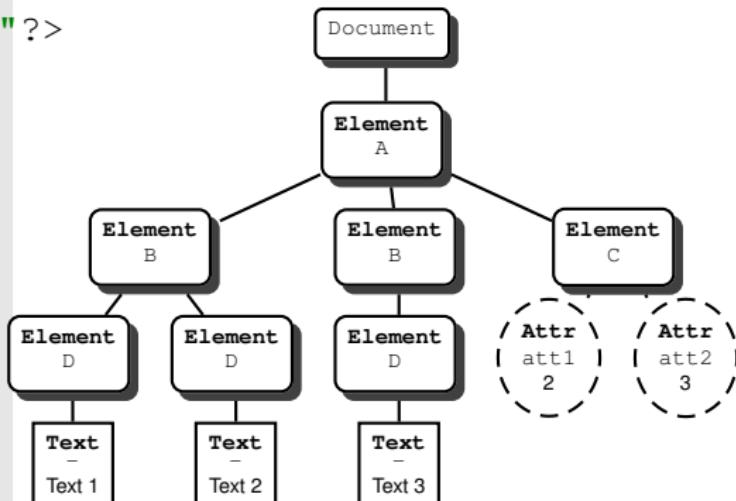
Remark

Remark 1 The XPath data model features also **ProcessingInstruction** and **Comment** node types.

Remark 2 Syntactic features specific to serialized representation (e.g., entities, literal section) are ignored by XPath.

From serialized representation to XML trees

```
<?xml version="1.0"  
      encoding="utf-8"?>  
<A>  
  <B att1='1'>  
    <D>Text 1</D>  
    <D>Text 2</D>  
  </B>  
  <B att1='2'>  
    <D>Text 3</D>  
  </B>  
  <C att2="a"  
      att3="b"/>  
</A>
```



(Some attributes not shown.)

XPath Data Model (cont.)

- The **root node** of an XML tree is the (unique) **Document** node;
- The **root element** is the (unique) **Element** child of the root node;
- A node has a **name**, or a **value**, or both
 - ▶ an **Element** node has a name, but no value;
 - ▶ a **Text** node has a value (a character string), but no name;
 - ▶ an **Attribute** node has both a name and a value.
- *Attributes are special!* Attributes are not considered as first-class nodes in an XML tree. They must be addressed specifically, when needed.

Remark

The expression “*textual value of an Element N*” denotes the concatenation of all the **Text** node values which are descendant of *N*, taken in the **document order**.

Outline

1 Introduction

2 Path Expressions

- Steps and expressions
- Axes and node tests
- Predicates

3 Operators and Functions

4 XPath examples

5 XPath 2.0

6 Reference Information

XPath Context

A step is evaluated in a specific **context** [$< N_1, N_2, \dots, N_n >, N_c$] which consists of:

- a context list $< N_1, N_2, \dots, N_n >$ of nodes from the XML tree;
- a context node N_c belonging to the context list.

Information on the context

- The **context length** n is a positive integer indicating the **size** of a contextual list of nodes; it can be known by using the function `last()`;
- The **context node position** $c \in [1, n]$ is a positive integer indicating the **position** of the context node in the context list of nodes; it can be known by using the function `position()`.

XPath steps

The basic component of XPath expression are **steps**, of the form:

axis::node-test [P₁] [P₂] ... [P_n]

axis is an **axis name** indicating what the direction of the step in the XML tree is (child is the default).

node-test is a **node test**, indicating the kind of nodes to select.

P_i is a **predicate**, that is, any XPath expression, evaluated as a boolean, indicating an additional condition. There may be no predicates at all.

Interpretation of a step

A step is evaluated with respect to a **context**, and returns a **node list**.

Example

descendant::C[@att1='1'] is a step which denotes all the **Element** nodes named C, descendant of the context node, having an **Attribute** node att1 with value 1

Path Expressions

A path expression is of the form: $[/]\text{step}_1/\text{step}_2/\dots/\text{step}_n$

A path that begins with $/$ is an **absolute** path expression;

A path that does not begin with $/$ is a **relative** path expression.

Example

`/A/B` is an **absolute** path expression denoting the **Element** nodes with name B, children of the root named A

`./B/descendant::text()` is a **relative** path expression which denotes all the **Text** nodes descendant of an **Element** B, itself child of the context node

`/A/B/@att1[.>2]` denotes all the **Attribute** nodes @att1 (of a B node child of the A root element) whose value is greater than 2

`.` is a special step, which refers to the context node. Thus, `./toto` means the same thing as `toto`.

Evaluation of Path Expressions

Each step step_i is interpreted with respect to a **context**; its result is a **node list**.

A step step_i is evaluated with respect to the context of step_{i-1} . More precisely:

For $i = 1$ (first step) if the path is **absolute**, the context is a singleton, the root of the XML tree; else (**relative paths**) the context is defined by the environment;

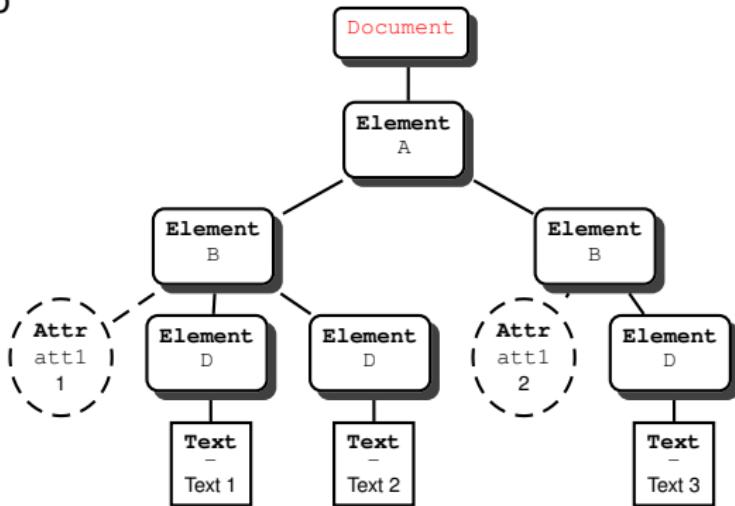
For $i > 1$ if $\mathcal{N} = \langle N_1, N_2, \dots, N_n \rangle$ is the result of step step_{i-1} , step_i is successively evaluated with respect to the context $[\mathcal{N}, N_j]$, for each $j \in [1, n]$.

The result of the path expression is the node set obtained after evaluating the last step.

Evaluation of /A/B/@att1

The path expression is absolute: the context consists of the root node of the tree.

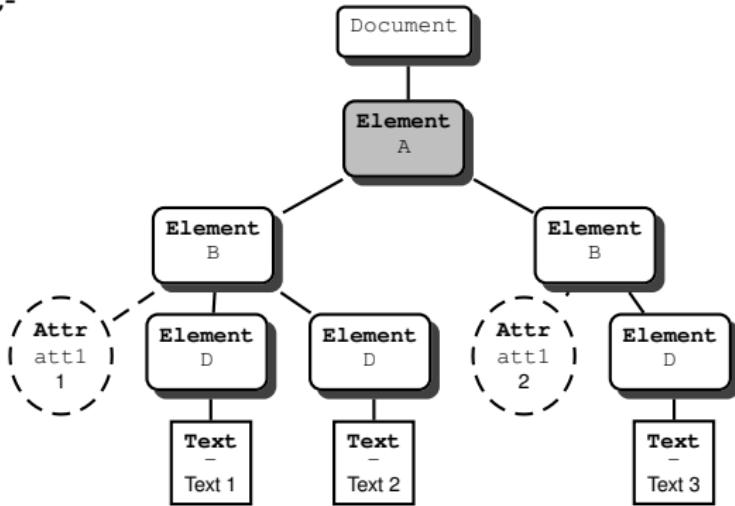
The first step, A, is evaluated with respect to this context.



Evaluation of /A/B/@att1

The result is A, the root element.

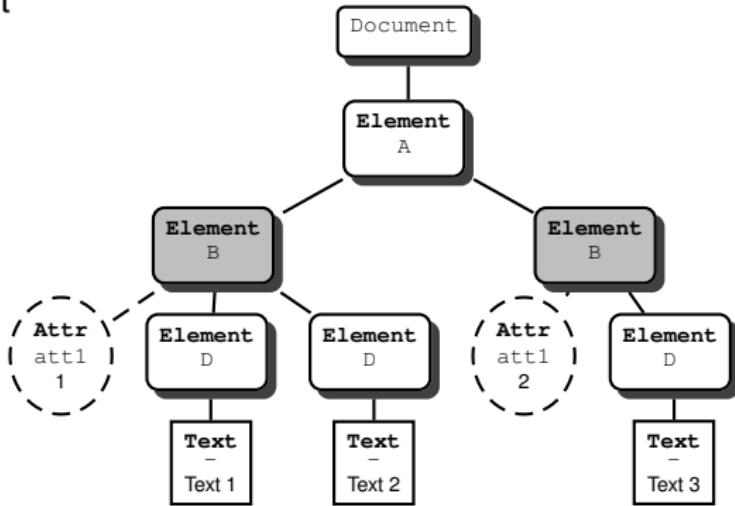
A is the context for the evaluation of the second step, B.



Evaluation of /A/B/@att1

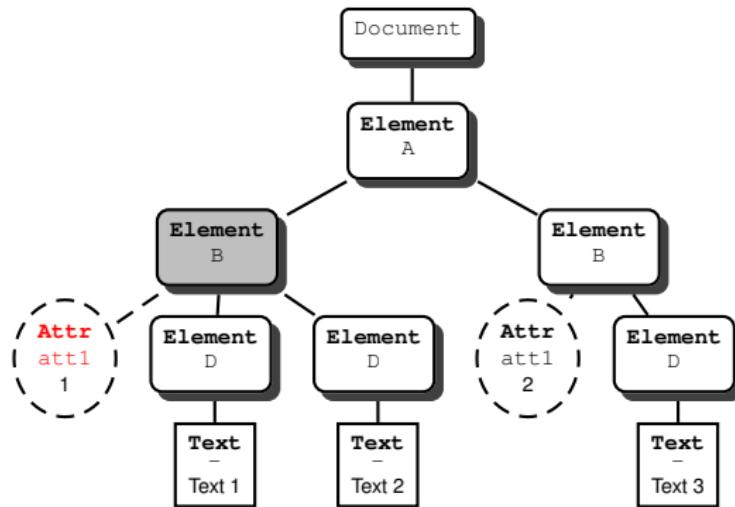
The result is a node list with two nodes B[1], B[2].

@att1 is first evaluated with the context node B[1].



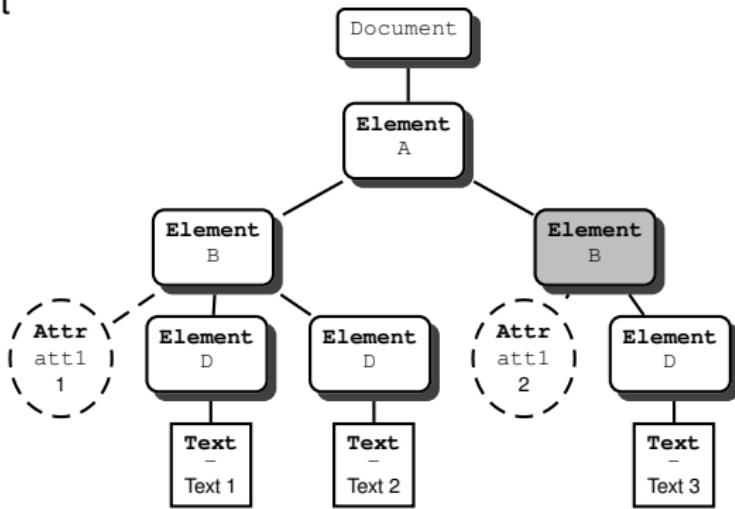
Evaluation of /A/B/@att1

The result is the attribute node of **B[1]**.



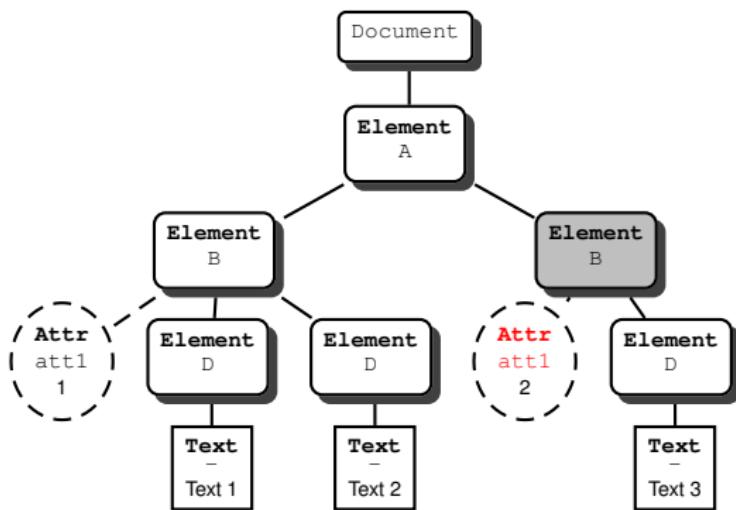
Evaluation of /A/B/@att1

@att1 is also evaluated with the context node **B[2]**.



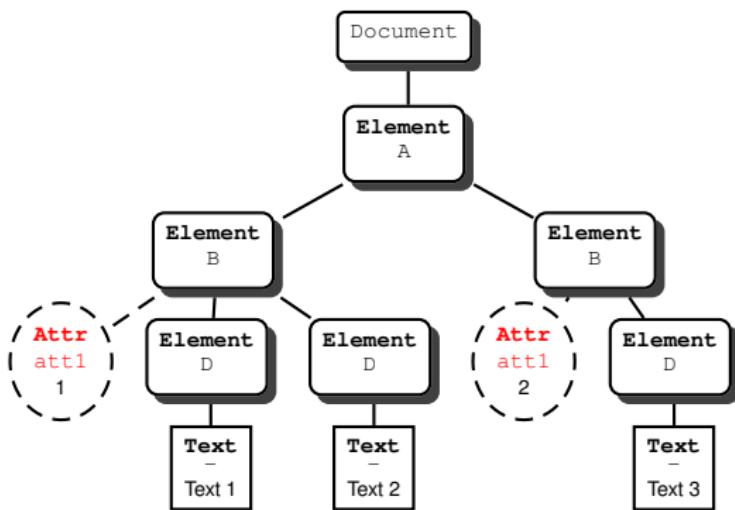
Evaluation of /A/B/@att1

The result is the attribute node of **B[2]**.



Evaluation of /A/B/@att1

Final result: the node set union of all the results of the last step, @att1.



Axes

An axis = a set of nodes determined from the context node, **and** an ordering of the sequence.

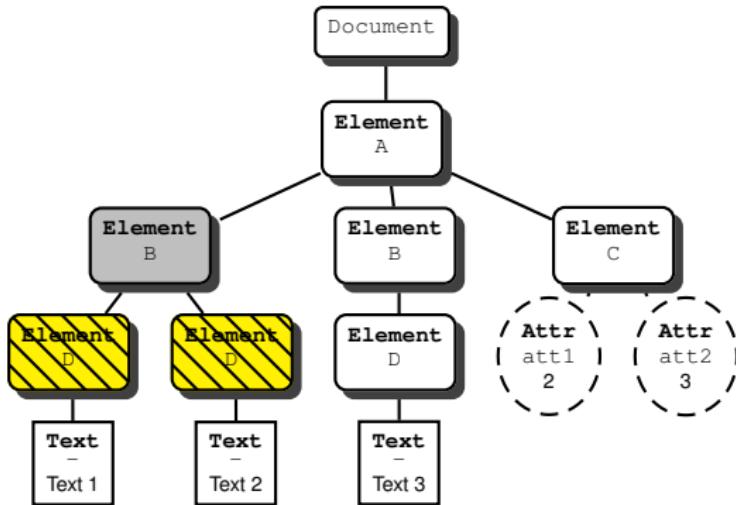
- `child` (**default axis**).
- `parent` Parent node.
- `attribute` Attribute nodes.
- `descendant` Descendants, excluding the node itself.
- `descendant-or-self` Descendants, including the node itself.
- `ancestor` Ancestors, excluding the node itself.
- `ancestor-or-self` Ancestors, including the node itself.
- `following` Following nodes in **document order**.
- `following-sibling` Following siblings in **document order**.
- `preceding` Preceding nodes in **document order**.
- `preceding-sibling` Preceding siblings in **document order**.
- `self` The context node itself.

Examples of axis interpretation

Child axis: denotes the **Element** or **Text** children of the context node.

Important: An **Attribute** node has a parent (the element on which it is located), but an attribute node is *not* one of the children of its parent.

Result of child::D (equivalent to D)



Examples of axis interpretation

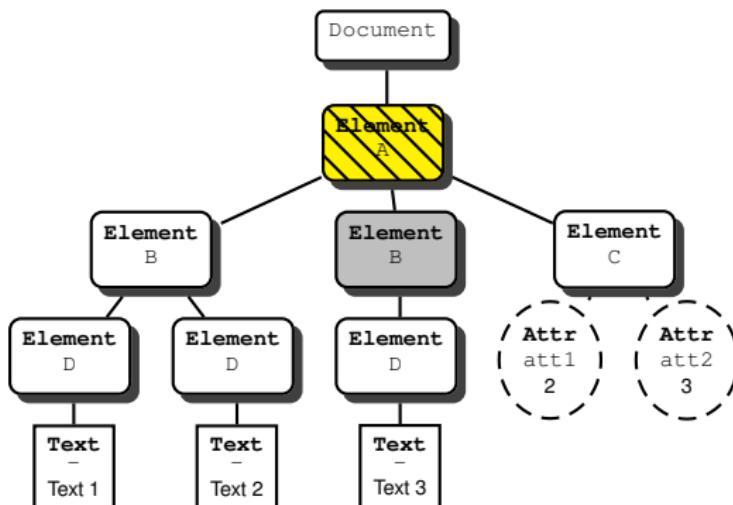
Parent axis: denotes the parent of the context node.

The node test is either an element name, or `*` which matches all names, `node()` which matches all node types.

Always a **Element** or **Document** node, or an empty node-set (if the parent does not match the node test or does not satisfy a predicate).

`...` is an abbreviation for `parent::node()`: the parent of the context node, whatever its type.

Result of `parent::node()` (may be abbreviated to `...`)

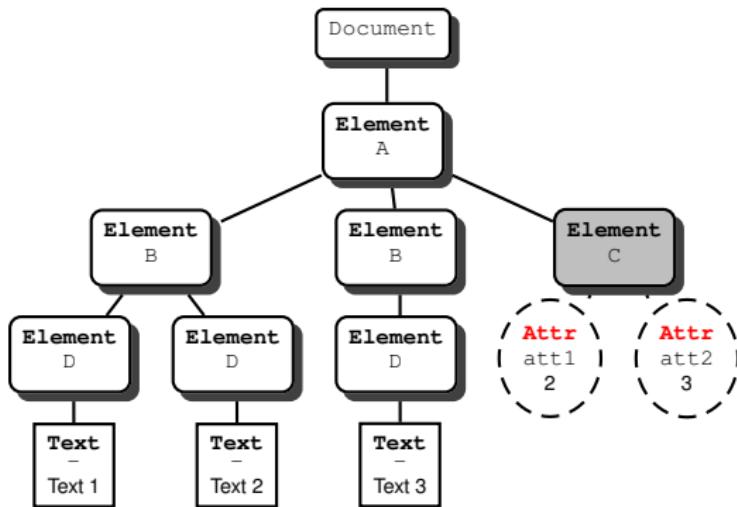


Examples of axis interpretation

Attribute axis: denotes the attributes of the context node.

The node test is either the attribute name, or `*` which matches all the names.

Result of `attribute::*` (equiv. to `@*`)



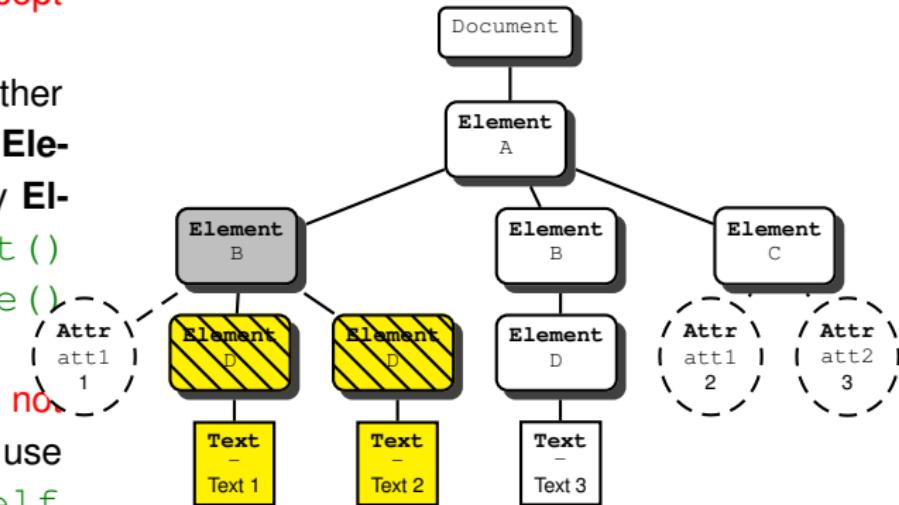
Examples of axis interpretation

Descendant axis: all the descendant nodes, **except** the **Attribute** nodes.

The node test is either the node name (for **Element** nodes), or `*` (any **Element** node) or `text()` (any **Text** node) or `node()` (all nodes).

The context node does **not** belong to the result: use `descendant-or-self` instead.

Result of `descendant::node()`



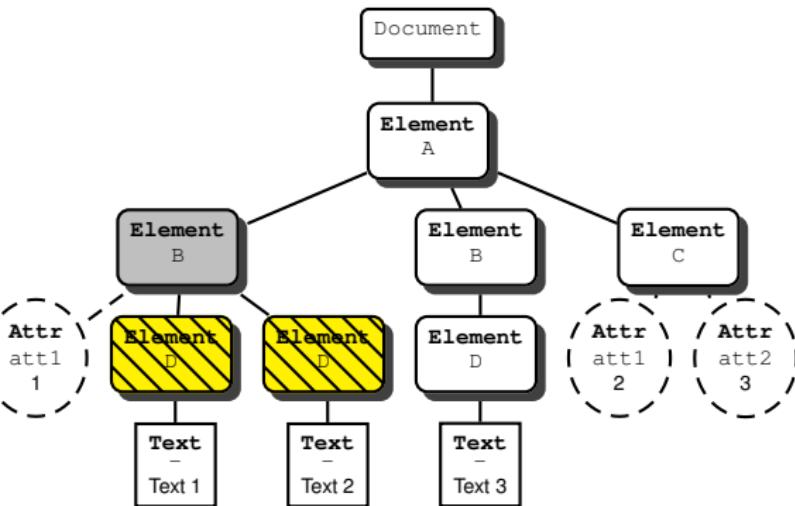
Examples of axis interpretation

Descendant axis: all the descendant nodes, **except** the **Attribute** nodes.

The node test is either the node name (for **Element** nodes), or ***** (any **Element** node) or **text()** (any **Text** node) or **node()** (all nodes).

The context node does **not** belong to the result: use **descendant-or-self** instead.

Result of `descendant::*`



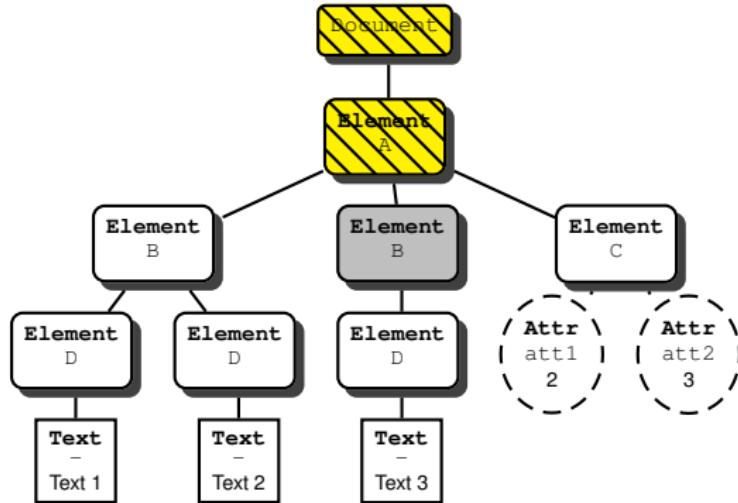
Examples of axis interpretation

Ancestor axis: all the ancestor nodes.

The node test is either the node name (for **Element** nodes), or `node()` (any **Element** node, and the **Document** root node).

The context node does **not** belong to the result: use `ancestor-or-self` instead.

Result of `ancestor::node()`



Examples of axis interpretation

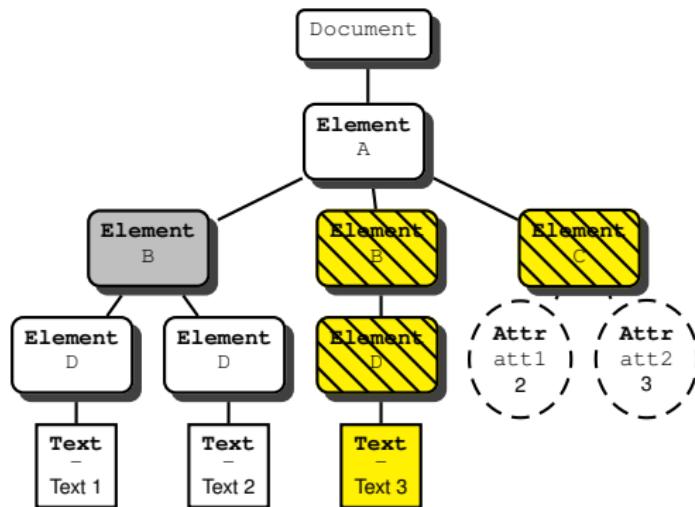
Following axis: all the nodes that follows the context node in the document order.

Attribute nodes are *not* selected.

The node test is either the node name, `* text()` or `node()`.

The axis `preceding` denotes all the nodes that precede the context node.

Result of `following::node()`



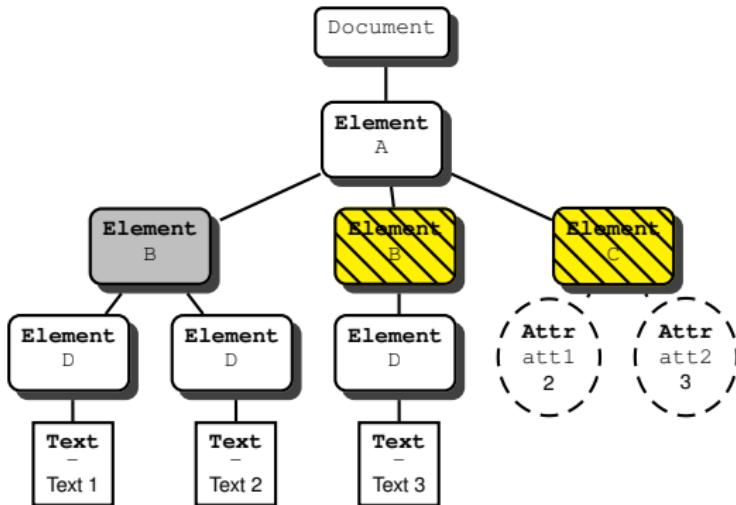
Examples of axis interpretation

Following sibling axis: all the nodes that follows the context node, and share the same parent node.

Same node tests as descendant or following.

The preceding-sibling axis denotes all the nodes that precede the context node.

Result of `following-sibling::node()`



Abbreviations (summary)

Summary of abbreviations:

somename	child::somename
.	self::node()
..	parent::node()
@someattr	attribute::someattr
a//b	a/descendant-or-self::node() /b
//a	/descendant-or-self::node() /a
/	/self::node()

Examples

`@b` selects the `b` attribute of the context node

`.../*` selects all element siblings of the context node, itself included (if it is an element node)

`//@someattr` selects all `someattr` attributes wherever their position in the document

Node Tests (summary)

A node test has one of the following forms:

`node()` any node

`text()` any text node

`*` any element (or any attribute for the `attribute` axis)

`ns:*` any element or attribute in the namespace bound to the prefix
`ns`

`ns:toto` any element or attribute in the namespace bound to the prefix
`ns` and whose name is `toto`

Examples

`a/node()` selects all nodes which are children of a `a` node, itself child of the context node

`xsl:*` selects all elements whose namespace is bound to the prefix `xsl` and that are children of the context node

`/*` selects the top-level element node

XPath Predicates

- Boolean expression, built with **tests** and the Boolean connectors **and** and **or** (negation is expressed with the **not ()** function);
- a **test** is
 - ▶ either an XPath expression, whose result is converted to a Boolean;
 - ▶ a comparison or a call to a Boolean function.

Important: predicate evaluation requires several rules for converting nodes and node sets to the appropriate type.

Example

- `//B[@att1=1]`: nodes **B** having an attribute **att1** with value 1;
- `//B[@att1]`: all nodes **B** having an attributes named **att1**!
⇒ `@att1` is an XPath expression whose result (a node set) is converted to a Boolean.
- `//B/descendant::text() [position()=1]`: the first **Text** node descendant of each node **B**.
Can be abbreviated to `//B/descendant::text() [1]`.

Predicate evaluation

A step is of the form

`axis::node-test[P]`.

Ex.: `/A/B/descendant::text() [1]`

- First

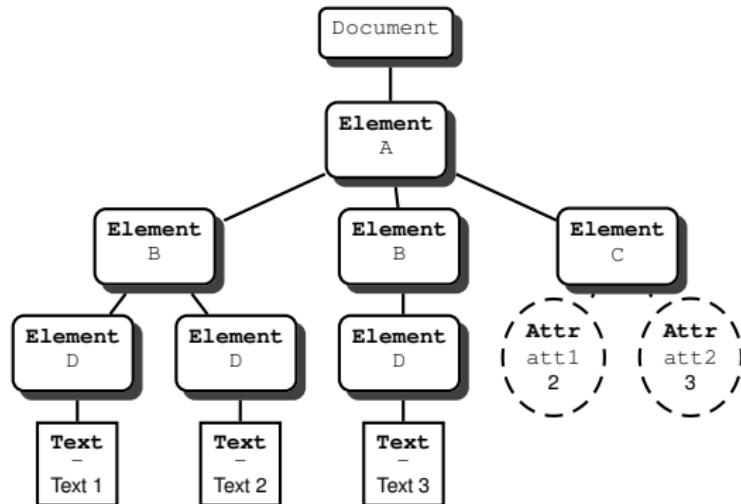
`axis::node-test`

is evaluated: one

obtains an

intermediate result I

- Second, for each node in I , P is evaluated: the step result consists of those nodes in I for which P is true.



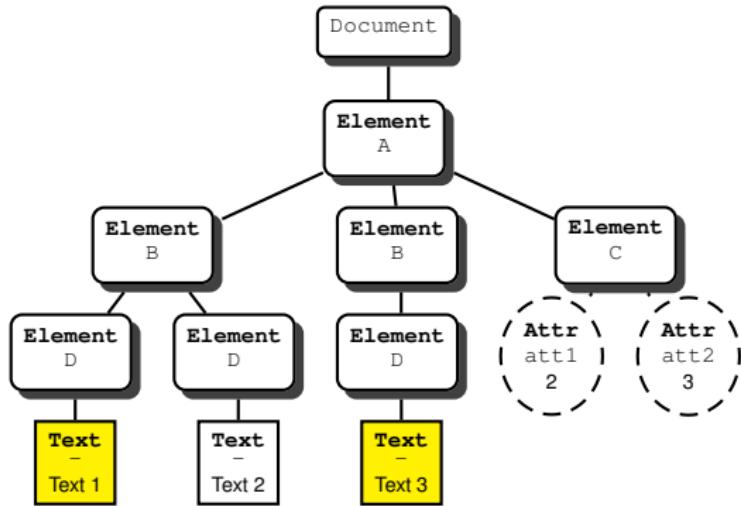
Predicate evaluation

Result

/A/B/descendant::text () [1]

Beware: an XPath step is **always** evaluated with respect to the context of the previous step.

Here the result consists of those **Text** nodes, first descendant (in the document order) of a node **B**.



XPath 1.0 Type System

Four primitive types:

Type	Description	Literals	Examples
boolean	Boolean values	<i>none</i>	<code>true()</code> , <code>not(\$a=3)</code>
number	Floating-point	12, 12.5	<code>1 div 33</code>
string	Ch. strings	"to", 'ti'	<code>concat('Hello', '!')</code>
nodeset	Node set	<i>none</i>	<code>/a/b[c=1 or @e]/d</code>

The `boolean()`, `number()`, `string()` functions **convert** types into each other (no conversion to nodesets is defined), but this conversion is done in an **implicit** way most of the time.

Rules for **converting to a boolean**:

- A number is true if it is neither 0 nor `NaN`.
- A string is true if its length is not 0.
- A nodeset is true if it is not empty.

Rules for converting a nodeset to a string:

- The string value of a nodeset is the string value of its first item in document order.
- The string value of an element or document node is the concatenation of the character data in all text nodes below.
- The string value of a text node is its character data.
- The string value of an attribute node is the attribute value.

Examples (Whitespace-only text nodes removed)

```
<a toto="3">
  <b titi='tutu'><c /></b>
  <d>tata</d>
</a>
```

string(/)	"tata"
string(/a/@toto)	"3"
boolean(/a/b)	true()
boolean(/a/e)	false()

Outline

- 1 Introduction
- 2 Path Expressions
- 3 Operators and Functions
- 4 XPath examples
- 5 XPath 2.0
- 6 Reference Information
- 7 Exercise

Operators

The following operators can be used in XPath.

`+, -, *, div, mod` standard arithmetic operators (Example: `1+2*3`).

Warning! `div` is used instead of the usual `/`.

`or, and` boolean operators (Example: `@a and c=3`)

`=, !=` equality operators. Can be used for strings, booleans or numbers. **Warning!** `//a !=3` means: there is an `a` element in the document whose string value is different from 3.

`<, <=, >=, >` relational operators (Example: `($a<2) and ($a>0)`).

Warning! Can only be used to compare numbers, not strings. If an XPath expression is embedded in an XML document, `<` must be escaped as `<`.

| union of nodesets (Example: `node() | @*`)

Remark

`$a` is a **reference** to the variable `a`. Variables can not be defined in XPath, they can only be referred to.

Node Functions

`count ($s)` returns the **number of items** in the nodeset `$s`

`local-name ($s)` returns the **name** of the first item of the nodeset `$s` in document order, **without** the namespace prefix; if `$s` is omitted, it is taken to be the context item

`namespace-uri ($s)` returns the **namespace URI** bound to the prefix of the name of the first item of the nodeset `$s` in document order; if `$s` is omitted, it is taken to be the context item

`name ($s)` returns the **name** of the first item of the nodeset `$s` in document order, **including** its namespace prefix; if `$s` is omitted, it is taken to be the context item

String Functions

`concat($s1, ..., $sn)` concatenates the strings \$s₁, ..., \$s_n

`starts-with($a, $b)` returns true () if the string \$a starts with \$b

`contains($a, $b)` returns true () if the string \$a contains \$b

`substring-before($a, $b)` returns the substring of \$a before the first occurrence of \$b

`substring-after($a, $b)` returns the substring of \$a after the first occurrence of \$b

`substring($a, $n, $l)` returns the substring of \$a of length \$l starting at index \$n (indexes start from 1). \$l may be omitted.

`string-length($a)` returns the length of the string \$a

`normalize-space($a)` removes all leading and trailing whitespace from \$a, and collapse all whitespace to a single character

`translate($a, $b, $c)` returns the string \$a, where all occurrences of a character from \$b has been replaced by the character at the same place in \$c.

Boolean and Number Functions

`not ($b)` returns the **logical negation** of the boolean \$b

`sum ($s)` returns the **sum** of the values of the nodes in the nodeset \$s

`floor($n)` rounds the number \$n to the **next lowest** integer

`ceiling($n)` rounds the number \$n to the **next greatest** integer

`round($n)` rounds the number \$n to the **closest** integer

Examples

`count (//*)` returns the number of elements in the document

`normalize-space(' titi toto ')` returns the string "titi
toto"

`translate('baba,'abcdef','ABCDEF')` returns the string "BABA"

`round(3.457)` returns the number 3

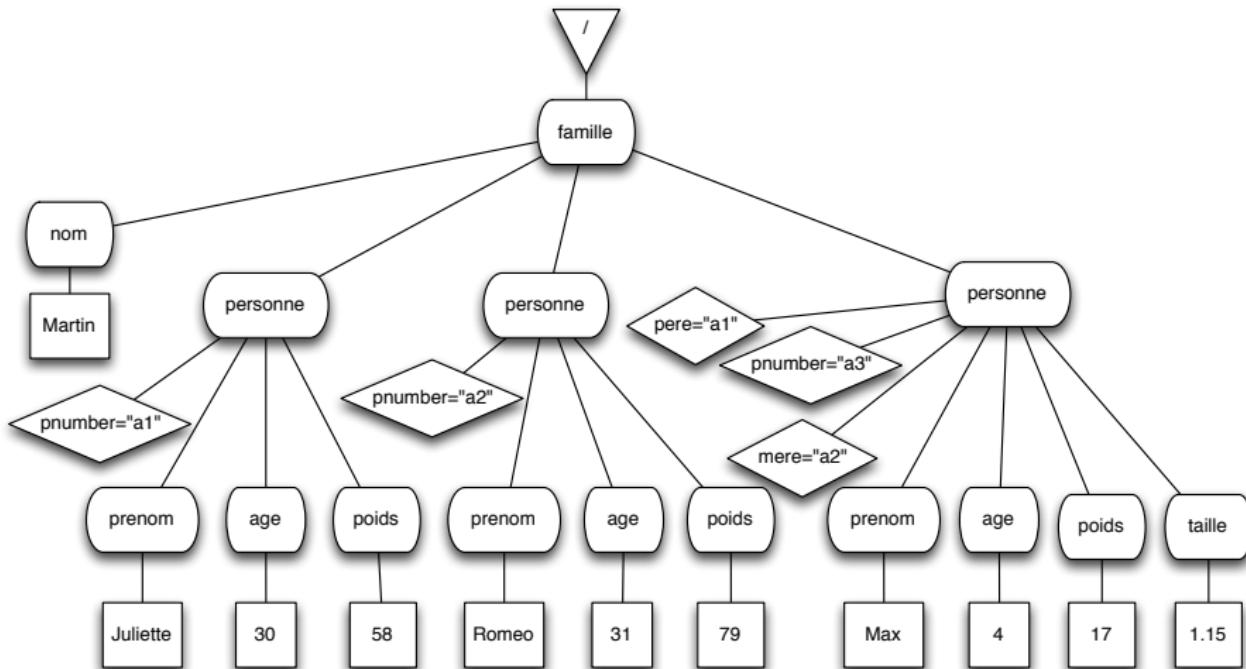
Outline

- 1 Introduction
- 2 Path Expressions
- 3 Operators and Functions
- 4 XPath examples
- 5 XPath 2.0
- 6 Reference Information
- 7 Exercise

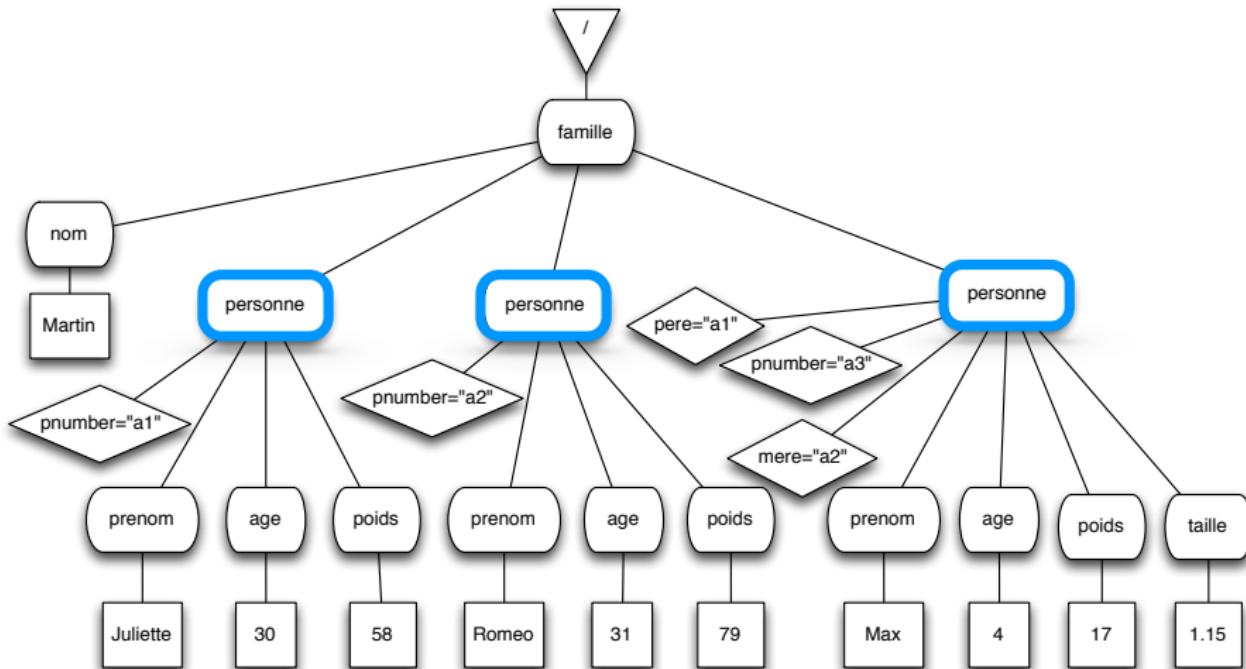
Exemple de document

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE famille SYSTEM "famille.dtd">
<famille>
    <nom>Martin</nom>
    <personne pnumber="a1">
        <prenom>Juliette</prenom>
        <age>30</age>
        <poids>58</poids>
    </personne>
    <personne pnumber="a2">
        <prenom>Romeo</prenom>
        <age>31</age>
        <poids>79</poids>
    </personne>
    <personne pnumber="a3" mere="a1" pere="a2">
        <prenom>Max</prenom>
        <age>4</age>
        <poids>17</poids>
        <taille>1.15</taille>
    </personne>
</famille>
```

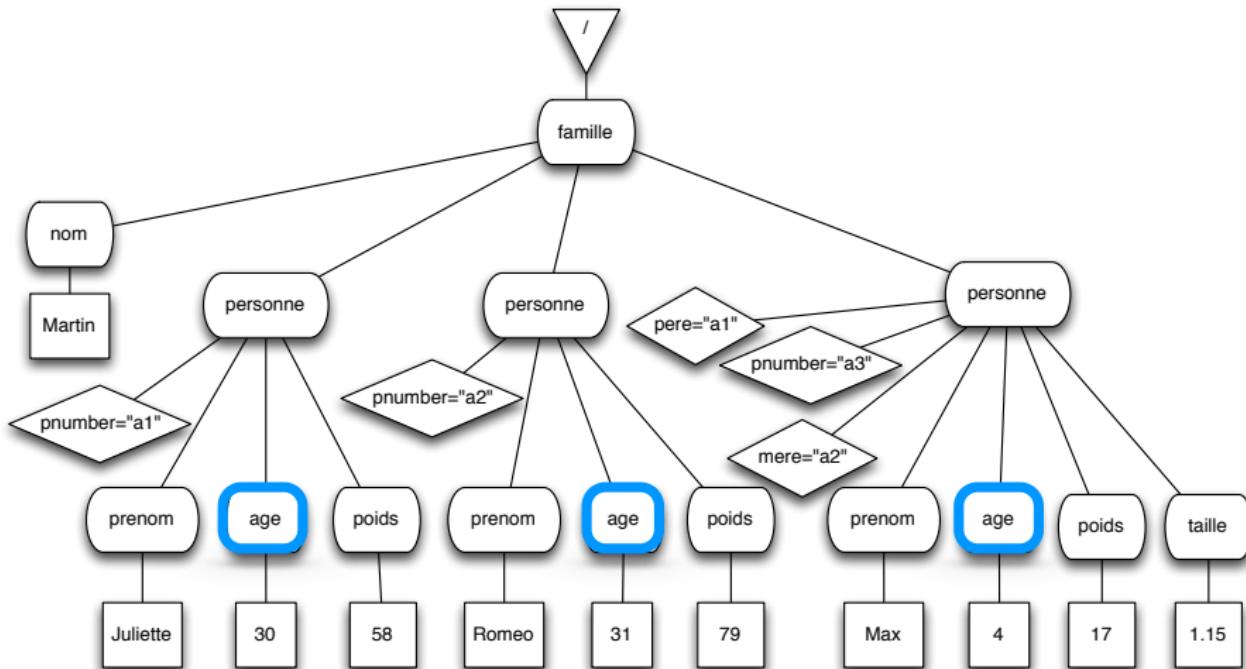
```
/descendant::personne/child::age/child::text()
```



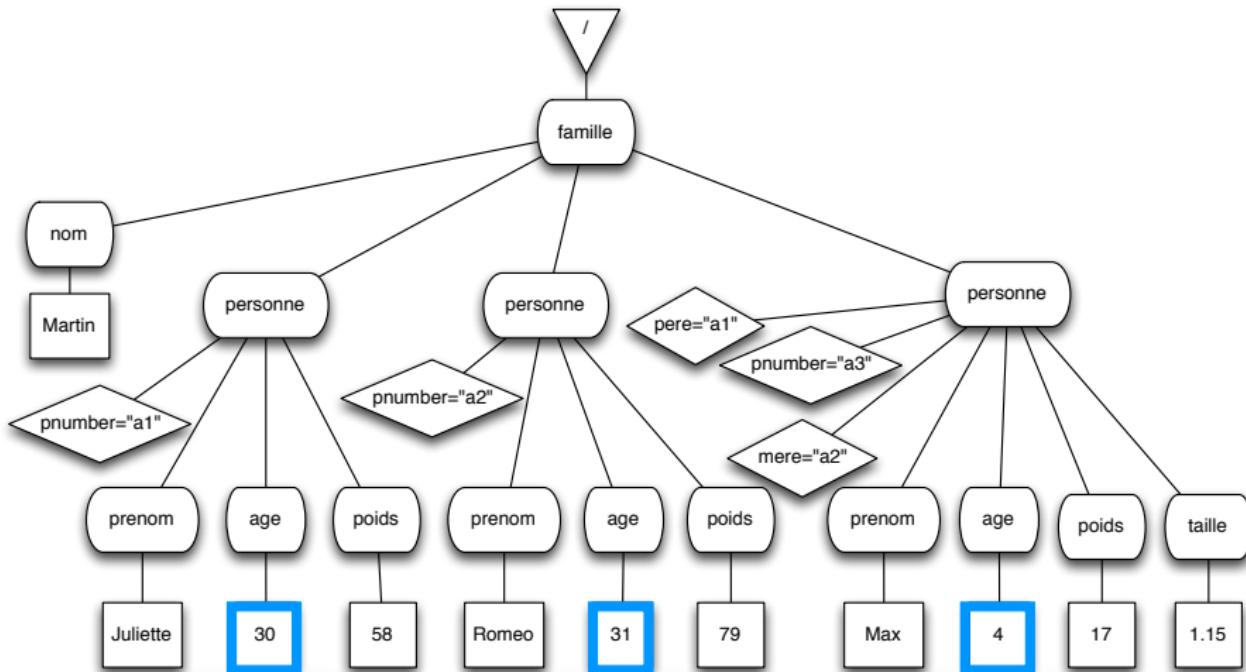
```
/descendant::personne/child::age/child::text()
```



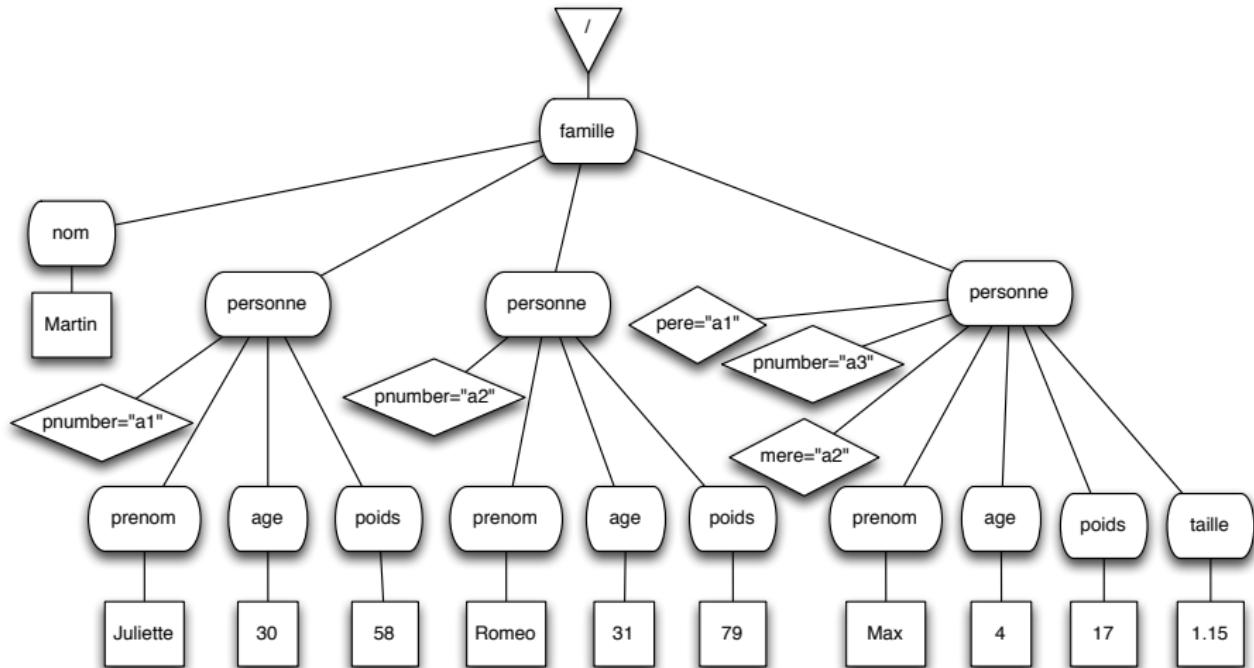
```
/descendant::personne/child::age/child::text()
```



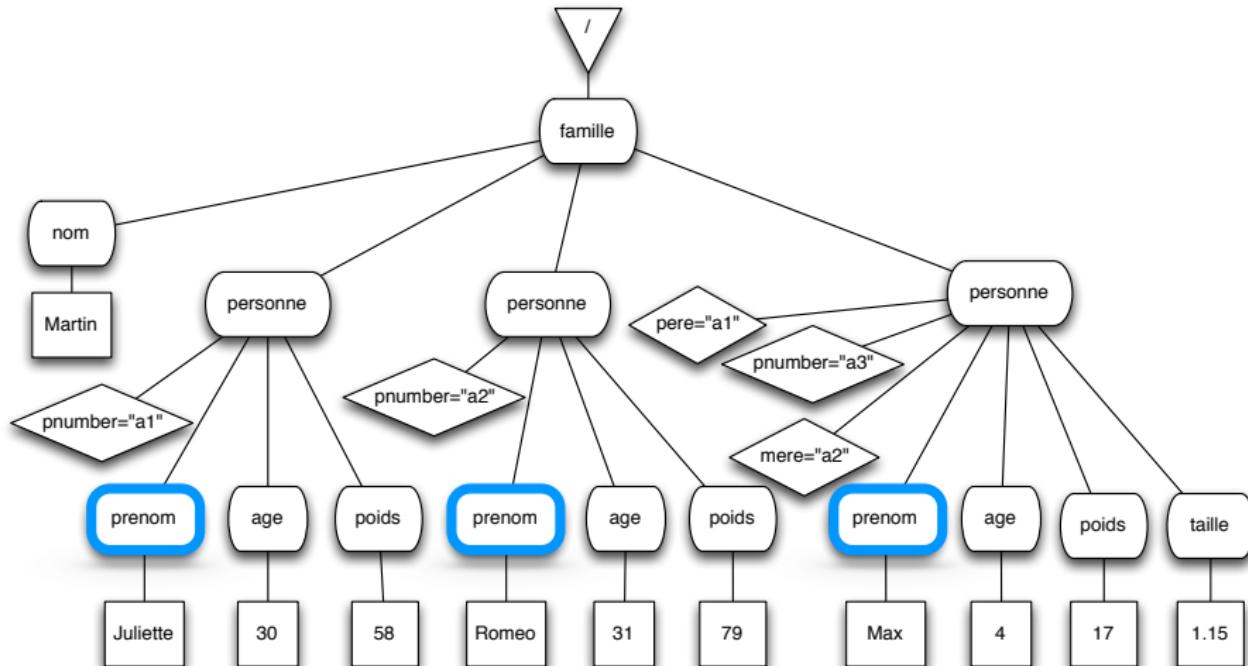
```
/descendant::personne/child::age/child::text()
```



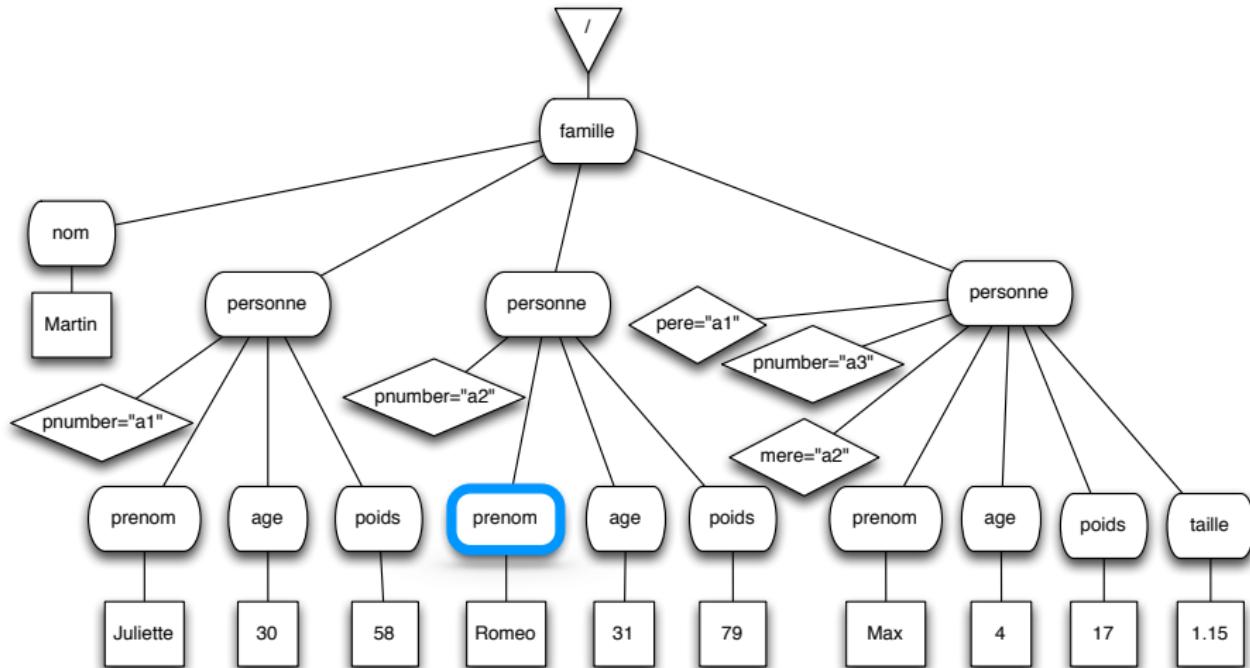
```
/descendant::prenom[child::text()="Romeo"]/parent::*[attribute::pnumber]
```



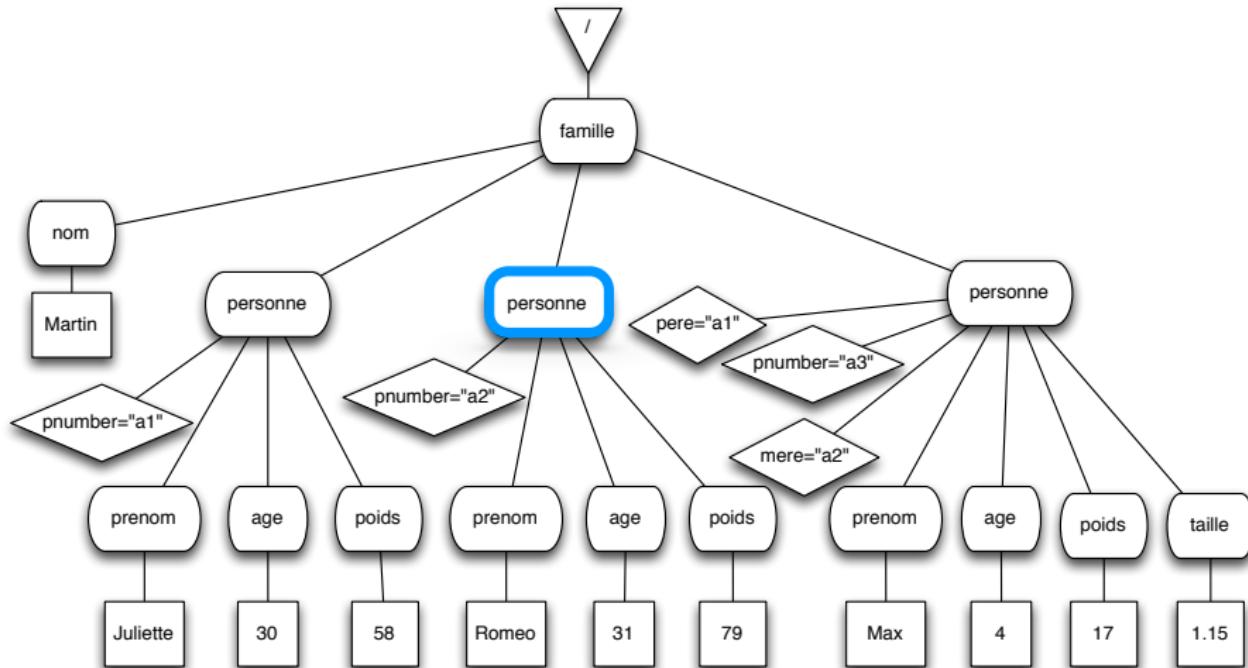
```
/descendant::prenom[child::text()="Romeo"]/parent::*[attribute::pnumber
```



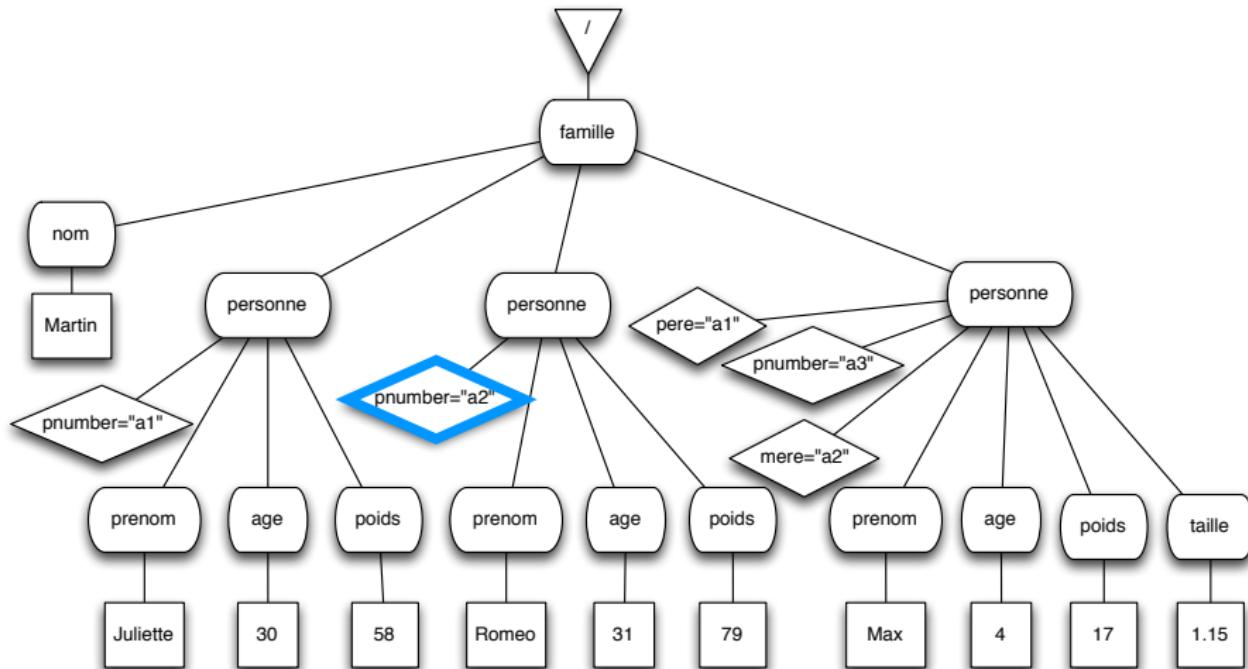
`/descendant::prenom[child::text()="Romeo"]//parent::*[attribute::pnumber]`



```
/descendant::prenom[child::text()="Romeo"]/parent::*[attribute::pnumber
```



```
/descendant::prenom[child::text()="Romeo"]/parent::*[attribute::pnumber
```



Sémantique existentielle des comparaisons

```
//livre[uteur = éiteur]
```

```
<?xml version="1.0" encoding="UTF-8"?>
<livres>
  <livre>
    <titre>Bonjour chez vous</titre>
    <auteur>Patrick McGoohan</auteur>
    <éditeur>Patrick McGoohan</éditeur>
  </livre>
</livres>
```

Sémantique existentielle des comparaisons

```
//livre[auteur = editeur]
```

```
<?xml version="1.0" encoding="UTF-8"?>
<livres>
  <livre>
    <titre>Bonjour chez vous</titre>
    <auteur>John Smith</auteur>
    <auteur>Patrick McGoohan</auteur>
    <editeur>James Kaith</editeur>
    <editeur>Patrick McGoohan</editeur>
  </livre>
</livres>
```

Sémantique existentielle des comparaisons

```
//livre[auteur = editeur]
```

```
<?xml version="1.0" encoding="UTF-8"?>
<livres>
  <livre>
    <titre>Bonjour chez vous</titre>
    <auteur>John Smith</auteur>
    <auteur>Will Stuart</auteur>
    <editeur>James Kaith</editeur>
    <editeur>Patrick McGoohan</editeur>
  </livre>
</livres>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<famille id="MARTIN">
  <femme id="1">
    <prenom>Juliette</prenom>
    <age>63</age>
    <poids>58</poids>
  </femme>
  <homme id="2">
    <prenom>Romeo</prenom>
    <age>65</age>
    <poids>97</poids>
  </homme>
  <homme id="3">
    <prenom>Max</prenom>
    <age>25</age>
    <poids>73</poids>
    <pere>2</pere>
    <mere>1</mere>
  </homme>
  <femme id="4">
    <prenom>Marie</prenom>
    <age>18</age>
    <poids>54</poids>
    <pere>2</pere>
    <mere>1</mere>
  </femme>
  <homme id="5">
    <prenom>Paul</prenom>
    <age>5</age>
    <poids>10</poids>
    <pere>3</pere>
  </homme>
</famille>
```

le prénom des personnes les plus lourdes

/famille/*[not(poids < //poids)]/prenom/text()

Romeo

/famille/*[poids >= //poids]/prenom/text()

Juliette
Romeo
Max
Marie
Paul

```

<?xml version="1.0" encoding="UTF-8"?>
<famille id="MARTIN">
    <femme id="1">
        <prenom>Juliette</prenom>
        <age>63</age>
        <poids>58</poids>
    </femme>
    <homme id="2">
        <prenom>Romeo</prenom>
        <age>65</age>
        <poids>97</poids>
    </homme>
    <homme id="3">
        <prenom>Max</prenom>
        <age>25</age>
        <poids>73</poids>
        <pere>2</pere>
        <mere>1</mere>
    </homme>
    <femme id="4">
        <prenom>Marie</prenom>
        <age>18</age>
        <poids>54</poids>
        <pere>2</pere>
        <mere>1</mere>
    </femme>
    <homme id="5">
        <prenom>Paul</prenom>
        <age>5</age>
        <poids>10</poids>
        <pere>3</pere>
    </homme>
</famille>

```

les id des hommes qui ne sont père d'aucune personne

~~//homme[@id != //pere]/@id~~

2

3

5

$\text{//homme[not(@id=//pere)]/@id}$

5

Examples (1)

`child::A/descendant::B` : B elements, descendant of an A element,
itself child of the context node;
Can be abbreviated to `A//B`.

`child::* / child::B` : all the B grand-children of the context node:
`descendant-or-self::B` : elements B descendants of the context
node, plus the context node itself if its name is B.

`child::B[position()=last ()]` : the last child named B of the
context node.
Abbreviated to `B[last ()]`.

`following-sibling::B[1]` : the first sibling of type B (in the
document order) of the context node,

Examples (2)

/descendant::B[10] the tenth element of type B in the document.

Not: the tenth element of the document, if its type is B!

child)::B[child)::C] : child elements B that have a child element C.

Abbreviated to B[C].

/descendant::B[@att1 or @att2] : elements B that have an

attribute att1 or an attribute att2;

Abbreviated to //B[@att1 or @att2]

* [self)::B or self)::C] : children elements named B or C

XPath 2.0

An extension of XPath 1.0, backward compatible with XPath 1.0. Main differences:

Improved data model tightly associated with XML Schema.

- ⇒ a new **sequence** type, representing ordered set of nodes and/or values, with duplicates allowed.
- ⇒ XSD types can be used for node tests.

More powerful new operators (loops) and better control of the output (limited tree restructuring capabilities)

Extensible Many new built-in functions; possibility to add user-defined functions.

XPath 2.0 is **also** a subset of XQuery 1.0.

Path expressions in XPath 2.0

New node tests in XPath 2.0:

`item()` any node or atomic value

`element()` any element (eq. to `child::*` in XPath 1.0)

`element(author)` any element named `author`

`element(*, xs:person)` any element of type `xs:person`

`attribute()` any attribute

Nested paths expressions:

Any expression that returns a sequence of nodes can be used as a step.

`/book/ (author | editor) /name`

XPath 1.0 Implementations

Large number of implementations.

`libxml2` Free **C** library for parsing XML documents, supporting XPath.

`java.xml.xpath` **Java** package, included with JDK versions starting from 1.5.

`System.Xml.XPath` **.NET** classes for XPath.

`XML::XPath` Free **Perl** module, includes a command-line tool.

`DOMXPath` **PHP** class for XPath, included in PHP5.

`PyXML` Free **Python** library for parsing XML documents, supporting XPath.

Exercise

```
<a>
  <b><c /></b>
  <b id="3" di="7">bli <c /><c><e>bla</e></c></b>
  <d>bou</d>
</a>
```

We suppose that all text nodes containing only whitespace are removed from the tree.

- Give the result of the following XPath expressions:
 - ▶ `//e/preceding::text()`
 - ▶ `count(//c|//b/node())`
- Give an XPath expression for the following problems, and the corresponding result:
 - ▶ Sum of all attribute values
 - ▶ Text content of the document, where every “b” is replaced by a “c”
 - ▶ Name of the child of the last “c” element in the tree

Graph databases

Why graph data?

Recall that a **graph** is a pair (N, E) where N is a finite set of **nodes** and $E \subseteq N \times N$ is a set of **edges** between nodes.

Depending on the application we can view edges as being **directed** or **undirected**.

Based on the application, we can also extend graphs with node and edge **labels** drawn from some domain \mathcal{L} :

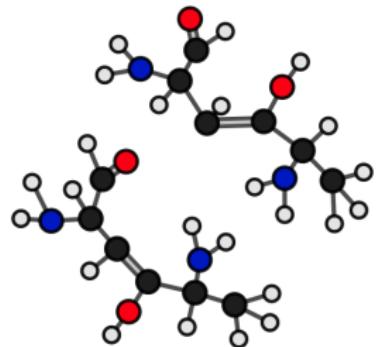
- $\lambda : N \rightarrow \mathcal{L}$, and
- $E \subseteq N \times \mathcal{L} \times N$.

We can further extend this such that E is a multi-set and nodes/edges carry (multi-)sets of (complex) labels.

Why graph data?

Big graph data sets are everywhere

- social networks (e.g., LinkedIn, Facebook)
- scientific networks (e.g., Uniprot, PubChem)
- knowledge graphs (e.g., DBpedia)
- ...



Why graph data?

Big graph data sets are everywhere

Analytics on big graphs increasingly important

- role discovery in social networks
- identifying patterns in biological networks
- finding emerging topics in a research network
- study of human migration patterns over short and long timescales
- ...

What makes a database a graph database?

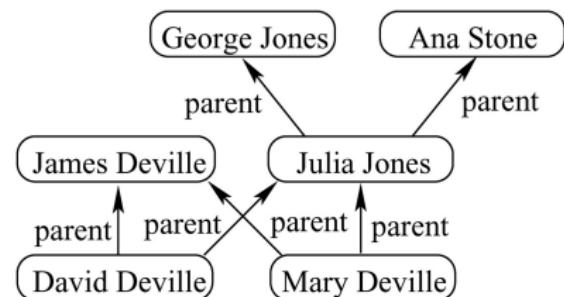
We can loosely identify three characteristics

- ① Data, schemas, queries, and/or updates are represented by graphs, or their generalization.
- ② Data manipulation is by graph transformations or operations on graph features such as paths, neighborhoods, diameter, etc.
- ③ There are possibly facilities for expressing appropriate integrity constraints such as label typing and domain/range restrictions.

A graph

NAME	LASTNAME
George	Jones
Ana	Stone
Julia	Jones
James	Deville
David	Deville
Mary	Deville

PERSON	PARENT
Julia	George
Julia	Ana
David	James
David	Julia
Mary	James
Mary	Julia



Towards triples

A data model for web data?

Some major problems with classical data models

- evolving schemas
- data heterogeneity

Lessons learned in the past decades:

- metadata is data
- it's all about relationships

Design for change

Towards triples

Basic requirements for a data model for the web:

- ① flexibly capture “things” and their relationships
- ② don’t force artificial data/metadata distinctions
- ③ support full spectrum between structured and unstructured data

The RDF data model

Resource description framework (RDF)

- W3C recommendation data model for (semantic) web data
- graph-based data model, embodying lessons learned from previous data models
- uses web identifiers (URIs) to identify resources (“things”)
- uses triples to state relationships between things
- serialization formats: N-triples, N3, RDF/XML

The RDF data model: triples

Relationships: Triples

- basic data structure
- has form (subject, predicate, object)
- “subject *has relationship predicate to object*”
- has no “metadata”, i.e., the predicate is also just a piece of data

An [RDF graph](#) is a finite set of triples.

```
{<Fred, WorksInDept, 1234>,
 <Yoko, WorksInDept, 5678>,
 <Yoko, knows, Fred>,
 <Jane, HasTitle, Safety Officer>,
 <Yoko, HasTitle, Team Leader>,
 <Yoko, HasTitle, Safety Officer>,
 <1234, LocatedInBuilding, B9>, ...,
 <knows, subPropertyOf, socialRelationship>}
```

Sources of RDF data

As part of the Linked Open Data (LOD) vision, a wide variety of data becoming available in RDF:

- Friend of a friend
- Wikipedia
- Uniprot gene database
- Bio2RDF
- BBC, New York Times
- open government data (UK, Australia, US, NL, Japan, ...)
- ...

Richer data modeling

RDF allows us to easily express facts

- John is the father of Mary
 $\{(john, fatherOf, mary)\}$
- Geoffrey knows that John is the father of Mary
 $\{(geoff, knows, S), (S, subject, john), (S, predicate, fatherOf), (S, object, mary)\}$

But, we'd like to be able to express more generic knowledge

- All fathers are male
- If a person has a daughter, then they are a parent

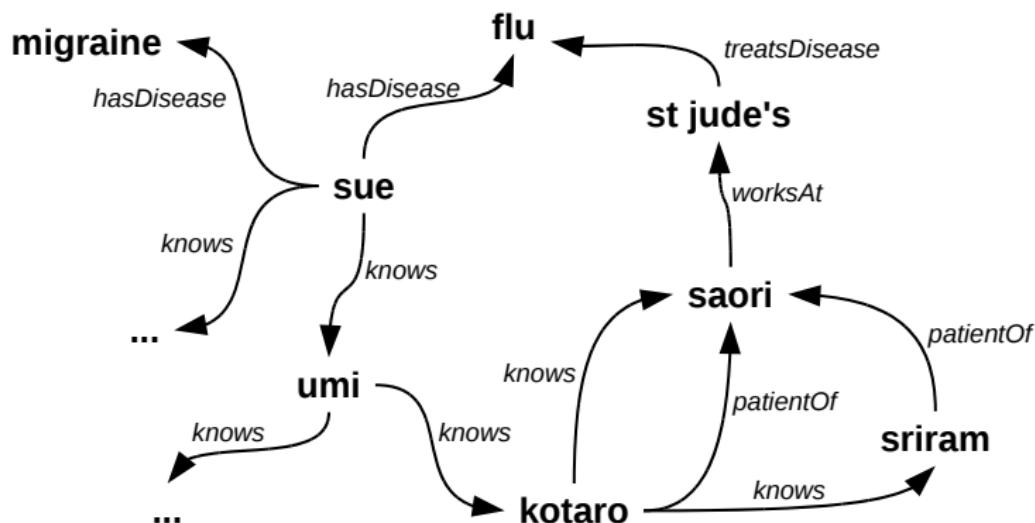
This kind of knowledge is often referred to as *schematic*, *ontological*, or *terminological* knowledge

Graph queries

A small graph

Let's consider directed edge-labeled graphs over the domain of **people**, **hospitals**, and **diseases**, and we have edge labels: knows, worksAt, patientOf, hasDisease, treatsDisease.

An example instance:



Query language capabilities

Graph query languages typically feature one or both of the following basic capabilities

- subgraph matching
- finding nodes connected by paths

and possibly additional advanced features such as approximate matching and comparing paths

Subgraph matching

Subgraph matching is the core basic capability of most graph query languages

Essentially, this consists of conjunctive queries on graphs

- an edge pattern is a triple (s, ℓ, t) where s and t can be either constants in N or variables, and ℓ is an edge label
- a query rule is then a pattern

$$\textit{head} \quad \leftarrow \quad \textit{body}$$

where *head* and *body* are sets of edge patterns such that every variable occurring in *head* occurs in *body*

- ▶ alternatively, *head* is a list of zero or more of the variables (possibly with repetition) appearing in *body*
- a query is then a finite set of rules (of the same arity).

Subgraph matching

The semantics $Q(G)$ of evaluating a query Q on a graph G is based on embeddings of the rule *body*'s of Q in G :

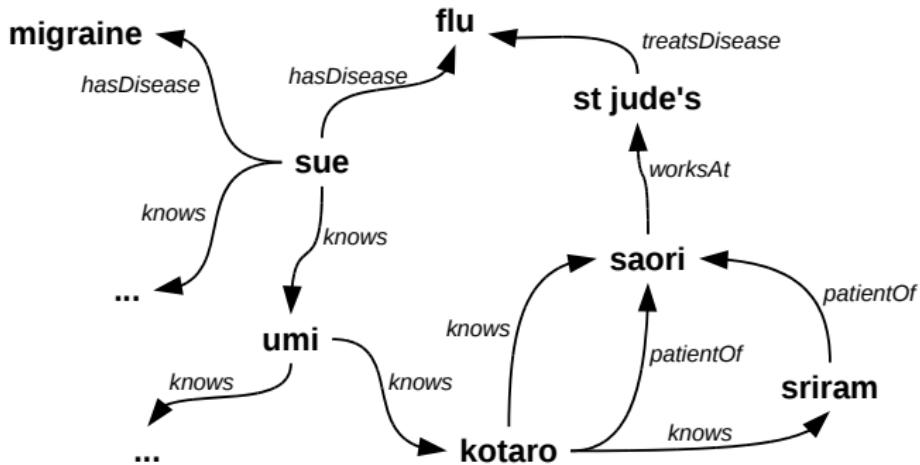
$$Q(G) = \bigcup_{\text{head} \leftarrow \text{body} \in Q} \{h(\text{head}) \mid h(\text{body}) \subseteq G\}$$

where h is a **homomorphism**, i.e., a function with domain $N \cup \text{Variables}$ and range N that is the identity on N .

Alternatively, some graph DBs adopt the stricter **isomorphism** semantics, i.e., homomorphisms that are **injective**.

- In other words, distinct variables in *body* must map to distinct nodes in G .

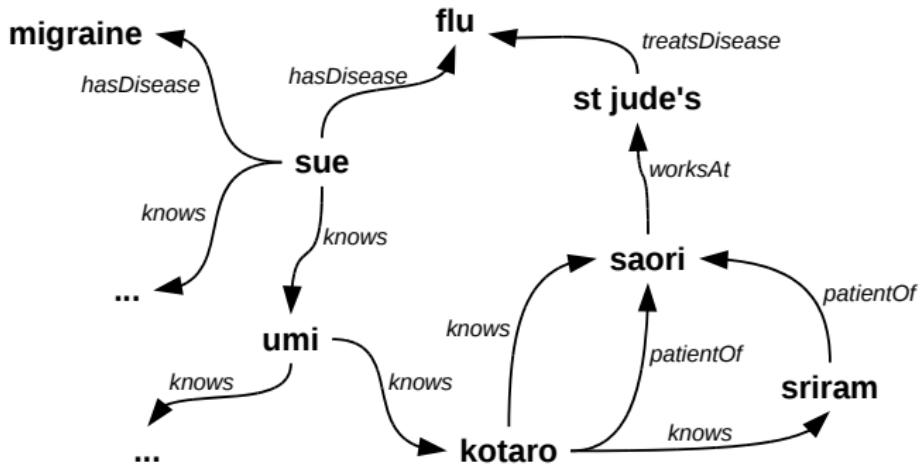
Subgraph matching



Example: People and the doctors of their friends

$$\begin{aligned} Q &= (?p, \text{friendDoctor}, ?d) \leftarrow (?p, \text{knows}, ?f), (?f, \text{patientOf}, ?d) \\ Q(G) &= \{(umi, \text{friendDoctor}, saori), (kotaro, \text{friendDoctor}, saori), \dots\} \end{aligned}$$

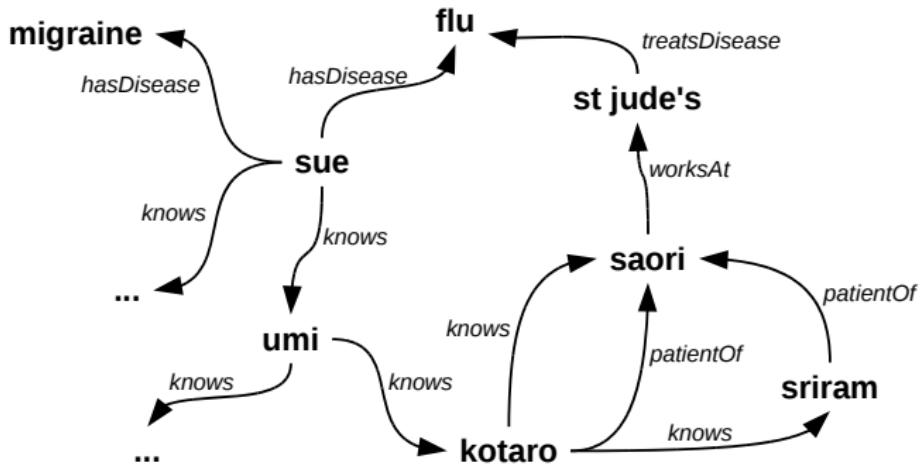
Subgraph matching



Example: People who know someone who knows a doctor.

$$\begin{aligned} Q &= \langle ?p \rangle \leftarrow (?p, \text{knows}, ?f), (?f, \text{knows}, ?d), (?po, \text{patientOf}, ?d) \\ Q(G) &= \{\langle \text{umi} \rangle, \dots\} \end{aligned}$$

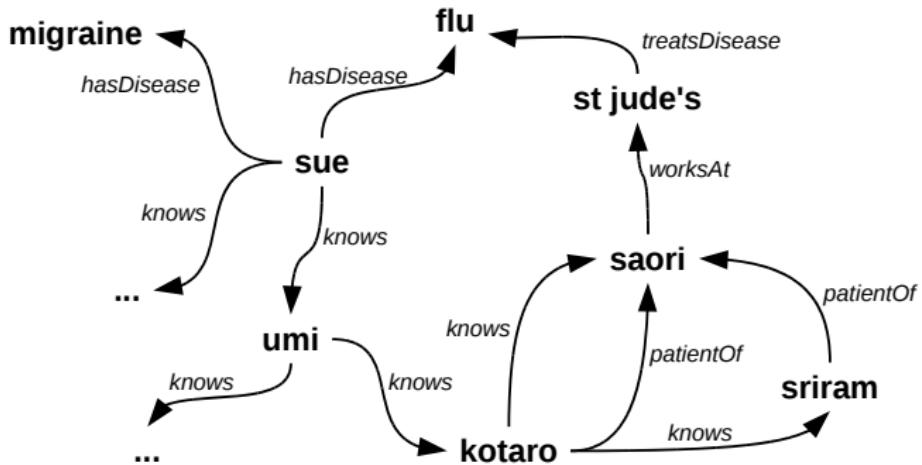
Subgraph matching



Example: Patients and their friends ([homomorphisms](#))

$$\begin{aligned} Q &= \langle ?p, ?f \rangle \leftarrow (?p, \text{knows}, ?f), (?p, \text{patientOf}, ?d) \\ Q(G) &= \{\langle \text{kotaro}, \text{saori} \rangle, \langle \text{kotaro}, \text{sriram} \rangle, \dots\} \end{aligned}$$

Subgraph matching



Example: Patients and their friends (**isomorphisms**)

$$\begin{aligned} Q &= \langle ?p, ?f \rangle \leftarrow (?p, \text{knows}, ?f), (?p, \text{patientOf}, ?d) \\ Q(G) &= \{(\text{kotaro}, \text{saori}), (\text{kotaro}, \text{sriram}), \dots\} \end{aligned}$$

Subgraph matching

Evaluation of subgraph matching queries is NP-complete. This follows from the intractability of the graph homomorphism problem.

Hence, relaxations have been proposed recently based on a *simulation-based* semantics for subgraph matching

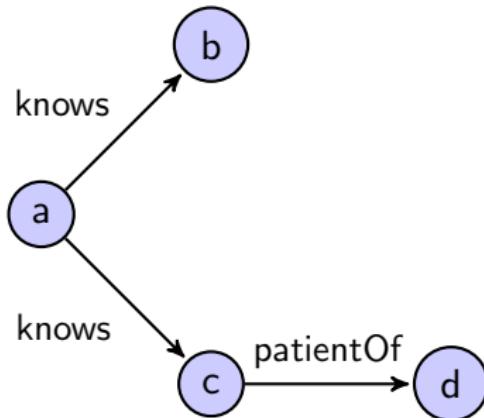
- $\mathcal{O}(|G||Q|)$ time complexity (instead of intractable)

Subgraph matching: simulations

That is, instead of homomorphisms embedding Q in G , we look for a binary relation S between the nodes and variables of (a given) body of Q and the nodes of G such that

- ① for each constant n in the body of Q , n is a node of G and $(n, n) \in S$;
- ② for each variable v in the body of Q there exists a node n of G such that $(v, n) \in S$; and,
- ③ for each $(x, n) \in S$ and each edge pattern $(x, \ell, x') \in Q$, there is an edge $n \xrightarrow{\ell} n' \in G$ such that $(x', n') \in S$.

Subgraph matching: simulations



Example. The following boolean query is simulated in the graph above, but evaluates to “false” under standard query semantics

$$\langle \rangle \leftarrow (?x, \text{knows}, ?y), (?x, \text{knows}, ?z), (?z, \text{patientOf}, ?y)$$

Here a simulation is $S = \{ (?x, a), (?z, c), (?y, b), (?y, d) \}$

Path matching

Regular path queries return all paths (i.e., pairs of nodes) connected by some regular expression over edge labels

- i.e., queries of the form

$$\langle ?x, ?y \rangle \leftarrow (?x, r, ?y)$$

where r is a regular expression over \mathcal{L}

- for example, the “knowing” social network is

$$\langle ?x, ?y \rangle \leftarrow (?x, \text{knows}^+, ?y)$$

and the general social network is

$$\langle ?x, ?y \rangle \leftarrow (?x, (\text{knows} \cup \text{patientOf})^+, ?y)$$

- $\mathcal{O}(|G||r|)$ time complexity
- many variations such as Unions of Conjunctions of RPQs have been intensively studied

Practical syntaxes for graph queries

SPARQL

SPARQL: SPARQL Protocol And RDF Query Language

- Query language for data from RDF documents
- W3C specification since January 2008
- Extremely successful in practice
 - ▶ Key technology in the Linked Data framework
- (Backwards compatible) revision finished in March 2013

Parts of the SPARQL specification:

- Query language: our focus
- Result format: encode results in XML
- Query protocol: transmitting queries and results

SPARQL

```
PREFIX ex: <http://example.org/>
SELECT ?title ?author
FROM <http://example.org/example>
WHERE
{ ?book ex:publishedBy <http://elsevier.com> .
  ?book ex:title      ?title .
  ?book ex:author     ?author . }
```

- Main part is a query pattern (WHERE)
 - ▶ Essentially a subgraph query
 - ▶ **Note:** Variables can be used, even in predicate positions
- Abbreviations for URIs (PREFIX)
- Graph(s) to be queried (FROM)
- Query result based on selected variables (SELECT)
 - ▶ can also CONSTRUCT an output graph

SPARQL: SELECT versus CONSTRUCT

The SELECT query form generates vectors of variable bindings

$$\langle ?title, ?author \rangle \leftarrow (?book, publishedBy, elsevier), \\ (?book, title, ?title), (?book, author, ?author)$$

The CONSTRUCT query form, on the other hand, generates an RDF graph

$$(?book, rdf:type, fullBook) \leftarrow (?book, publishedBy, elsevier), \\ (?book, title, ?title), (?book, author, ?author)$$

SPARQL: SELECT versus CONSTRUCT

```
PREFIX ex: <http://example.org/>
CONSTRUCT { ?book rdf:type ex:fullBook }
FROM <http://example.org/example>
WHERE
{ ?book ex:publishedBy <http://elsevier.com> .
  ?book ex:title      ?title .
  ?book ex:author     ?author . }
```

SPARQL

Further features

- OPTIONAL clauses (introduces null values)
- FILTER clauses, for specifying filter conditions
- UNION of (basic graph patterns) BGP
- Modifiers: postprocess query result set, e.g.: ORDER BY ?age LIMIT 10 OFFSET 5

Further features (SPARQL 1.1)

- aggregates, negation, subqueries, expressions in SELECT, path recursion, federation

SPARQL 1.1

Property paths. Allows us to introduce RPQs into BGPs.

```
PREFIX ex: <http://example.org/>
SELECT ?name
FROM <http://example.org/example>
WHERE
{ ?book    ex:publishedBy <http://elsevier.com> .
  ?book    ex:title        ?title .
  ?book    ex:author        ?author .
  ?author  foaf:knows/foaf:name ?name
}
```

SPARQL 1.1

Property paths. Allows us to introduce RPQs into BGPs.

```
PREFIX ex: <http://example.org/>
SELECT ?name
FROM <http://example.org/example>
WHERE
{ ?book    ex:publishedBy <http://elsevier.com> .
  ?book    ex:title        ?title .
  ?book    ex:author        ?author .
  ?author  foaf:knows/foaf:knows/foaf:name ?name
}
```

SPARQL 1.1

Property paths. Allows us to introduce RPQs into BGPs.

```
PREFIX ex: <http://example.org/>
SELECT ?name
FROM <http://example.org/example>
WHERE
{ ?book    ex:publishedBy <http://elsevier.com> .
  ?book    ex:title        ?title .
  ?book    ex:author        ?author .
  ?author  foaf:knows+/foaf:name ?name
}
```

SPARQL

SPARQL: Based on matching simple graph patterns

- simple SQL-like syntax, built from subgraph and regular path queries
- key standard in linked data initiative
 - ▶ try out the DBpedia end point <http://live.dbpedia.org/sparql>

Triple stores

Examples of popular **triple stores** (i.e., RDF graph databases)

- Virtuoso: open source and commercial industrial-strength triple store
 - ▶ <http://virtuoso.openlinksw.com/dataspace/dav/wiki/Main/>
- Sesame/BigData: open source and commercial industrial-strength triple store
 - ▶ <http://rdf4j.org>
 - ▶ <http://www.sysstag.com/bigdata.htm>

Also, **relational DB-based** RDF solutions are available in industrial-strength IBM DB2 and Oracle Database 12c products. See:

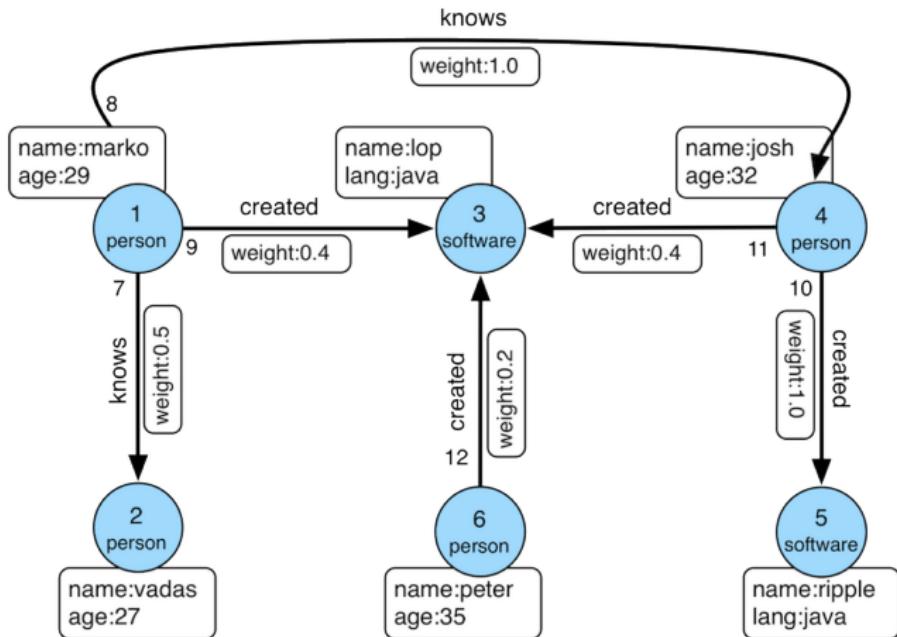
- <http://www.ibm.com/developerworks/data/tutorials/dm-1205rdfdb210/index.html>
- <http://www.oracle.com/technetwork/database/options/spatialandgraph/overview/rdfsemantic-graph-1902016.html>

openCypher

openCypher.

- Declarative graph query language of the popular open-source Neo4j graph database. <http://neo4j.com/developer/cypher/>
- Property graph model
 - ▶ directed node- and edge-labeled graph
 - ▶ nodes and edges have ID
 - ▶ nodes and edges carry sets of property-value pairs

openCypher: property graphs



openCypher: subgraph matching

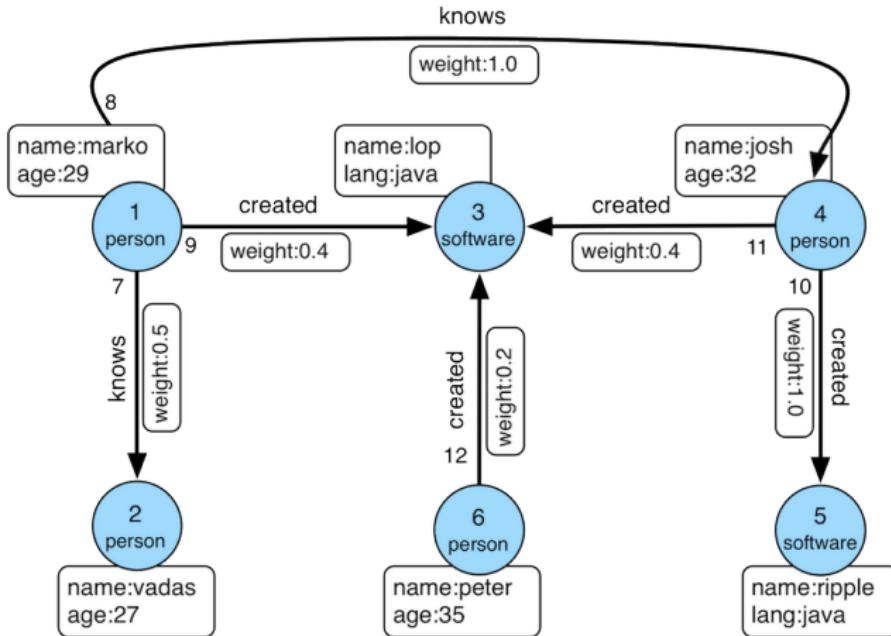
The basic building block of queries is subgraph matching, via a **MATCH** clause, with **isomorphic** matching.

```
MATCH (n:Person)-[:Created]->(m),  
      (m)<-[:Created]-(p)  
WHERE n.age = 29 AND p.age < 35  
RETURN p
```

$$\langle ?p \rangle \leftarrow (?n, created, ?m), (?p, created, ?m), \\ n.age = 29, p.age < 35, n.label = Person$$

Can be further combined using UNION, applying aggregation functions, string functions, etc.

openCypher: subgraph matching



```
MATCH (n:Person)-[:Created]->(m),  
      (m)<-[:Created]-(p)  
WHERE n.age = 29 AND p.age < 35  
RETURN p
```

Result is {⟨4⟩}.

openCypher: path queries

Cypher also provides support for RPQs in the **MATCH** clause.

```
MATCH (n:Person)-[:knows*]->(p)
WHERE n.name = "marko"
RETURN p
```

and with bounded recursion

```
MATCH (n:Person)-[:knows*2..7]->(p)
WHERE n.name = "marko"
RETURN p
```

Can also apply * to a disjunction of symbols

Note to other teachers and users of these slides: We would be delighted if you found this material useful in giving your own lectures. Feel free to use these slides verbatim, or to modify them to fit your own needs. If you make use of a significant portion of these slides in your own lecture, please include this message, or a link to our web site: <http://www.mmds.org>

Map-Reduce and the New Software Stack

Mining of Massive Datasets

Jure Leskovec, Anand Rajaraman, Jeff Ullman
Stanford University

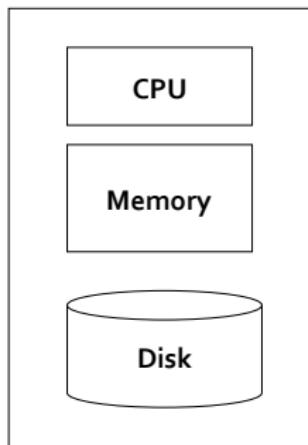
<http://www.mmds.org>



MapReduce

- Much of the course will be devoted to
large scale computing for data mining
- **Challenges:**
 - How to distribute computation?
 - Distributed/parallel programming is hard
- **Map-reduce** addresses all of the above
 - Google's computational/data manipulation model
 - Elegant way to work with big data

Single Node Architecture



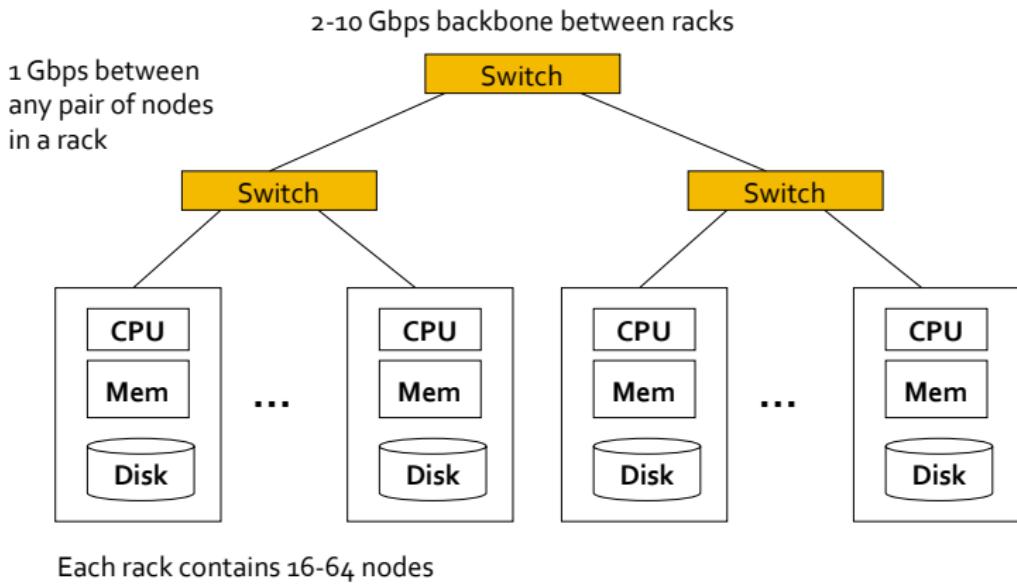
Machine Learning, Statistics

“Classical” Data Mining

Motivation: Google Example

- 20+ billion web pages x 20KB = 400+ TB
- 1 computer reads 30-35 MB/sec from disk
 - ~4 months to read the web
- ~1,000 hard drives to store the web
- Takes even more to do something useful with the data!
- Today, a standard architecture for such problems is emerging:
 - Cluster of commodity Linux nodes
 - Commodity network (ethernet) to connect them

Cluster Architecture



In 2011 it was guestimated that Google had 1M machines, <http://bit.ly/Shh0RO>



Large-scale Computing

- Large-scale computing for data mining problems on commodity hardware
- Challenges:
 - How do you distribute computation?
 - How can we make it easy to write distributed programs?
 - Machines fail:
 - One server may stay up 3 years (1,000 days)
 - If you have 1,000 servers, expect to loose 1/day
 - People estimated Google had ~1M machines in 2011
 - 1,000 machines fail every day!

Idea and Solution

- **Issue:** Copying data over a network takes time
- **Idea:**
 - Bring computation close to the data
 - Store files multiple times for reliability
- **Map-reduce addresses these problems**
 - Google's computational/data manipulation model
 - Elegant way to work with big data
 - **Storage Infrastructure – File system**
 - Google: GFS. Hadoop: HDFS
 - **Programming model**
 - Map-Reduce

Storage Infrastructure

- **Problem:**

- If nodes fail, how to store data persistently?

- **Answer:**

- **Distributed File System:**

- Provides global file namespace
 - Google GFS; Hadoop HDFS;

- **Typical usage pattern**

- Huge files (100s of GB to TB)
 - Data is rarely updated in place
 - Reads and appends are common

Distributed File System

■ Chunk servers

- File is split into contiguous chunks
- Typically each chunk is 16-64MB
- Each chunk replicated (usually 2x or 3x)
- Try to keep replicas in different racks

■ Master node

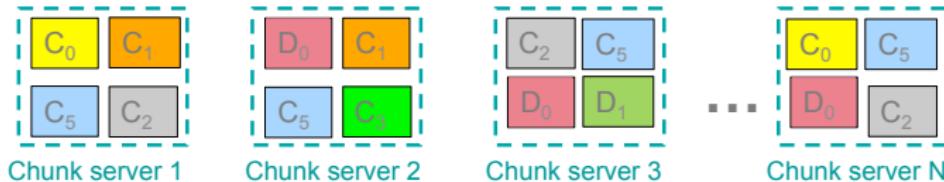
- a.k.a. Name Node in Hadoop's HDFS
- Stores metadata about where files are stored
- Might be replicated

■ Client library for file access

- Talks to master to find chunk servers
- Connects directly to chunk servers to access data

Distributed File System

- Reliable distributed file system
- Data kept in “chunks” spread across machines
- Each chunk **replicated** on different machines
 - Seamless recovery from disk or machine failure



Bring computation directly to the data!

Chunk servers also serve as compute servers

Programming Model: MapReduce

Warm-up task:

- We have a huge text document
- Count the number of times each distinct word appears in the file
- **Sample application:**
 - Analyze web server logs to find popular URLs

Task: Word Count

Case 1:

- File too large for memory, but all <word, count> pairs fit in memory

Case 2:

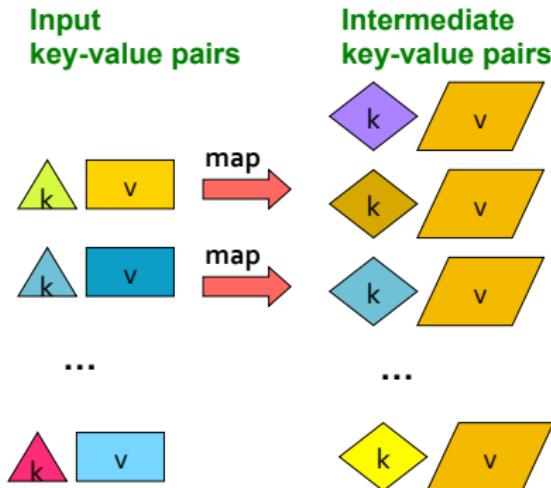
- Count occurrences of words:
 - `words (doc.txt) | sort | uniq -c`
 - where `words` takes a file and outputs the words in it, one per a line
- Case 2 captures the essence of **MapReduce**
 - Great thing is that it is naturally parallelizable

MapReduce: Overview

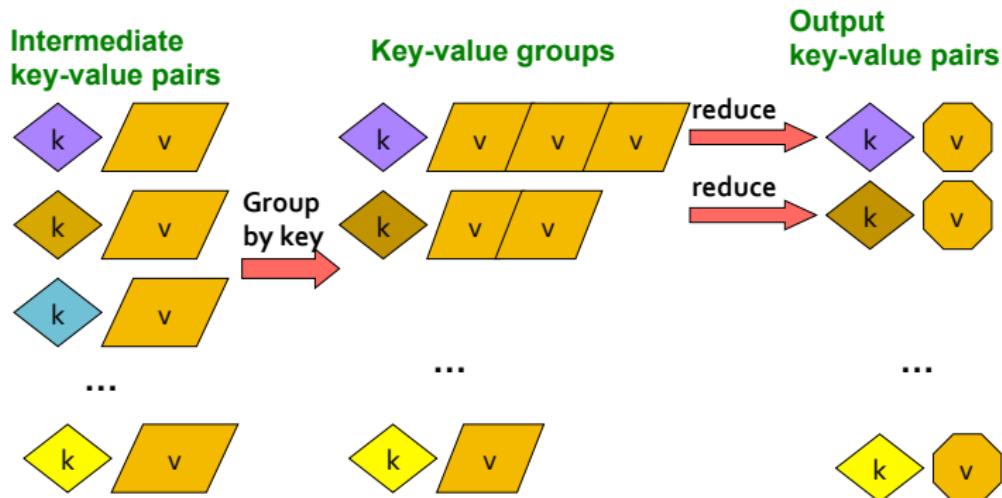
- Sequentially read a lot of data
- **Map:**
 - Extract something you care about
- **Group by key:** Sort and Shuffle
- **Reduce:**
 - Aggregate, summarize, filter or transform
- Write the result

Outline stays the same, **Map** and **Reduce** change to fit the problem

MapReduce: The Map Step



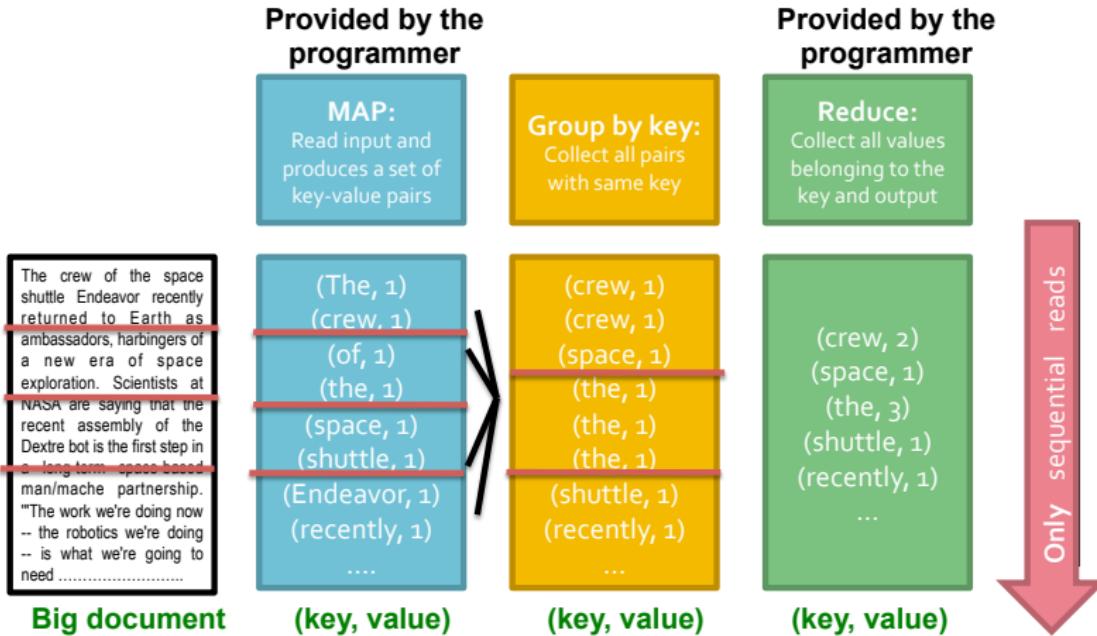
MapReduce: The Reduce Step



More Specifically

- **Input:** a set of key-value pairs
- Programmer specifies two methods:
 - **Map(k, v) $\rightarrow <k', v'>^*$**
 - Takes a key-value pair and outputs a set of key-value pairs
 - E.g., key is the filename, value is a single line in the file
 - There is one Map call for every (k, v) pair
 - **Reduce($k', <v'>^*$) $\rightarrow <k', v''>^*$**
 - All values v' with same key k' are reduced together and processed in v' order
 - There is one Reduce function call per unique key k'

MapReduce: Word Counting



Word Count Using MapReduce

```
map(key, value):
// key: document name; value: text of the document
for each word w in value:
    emit(w, 1)

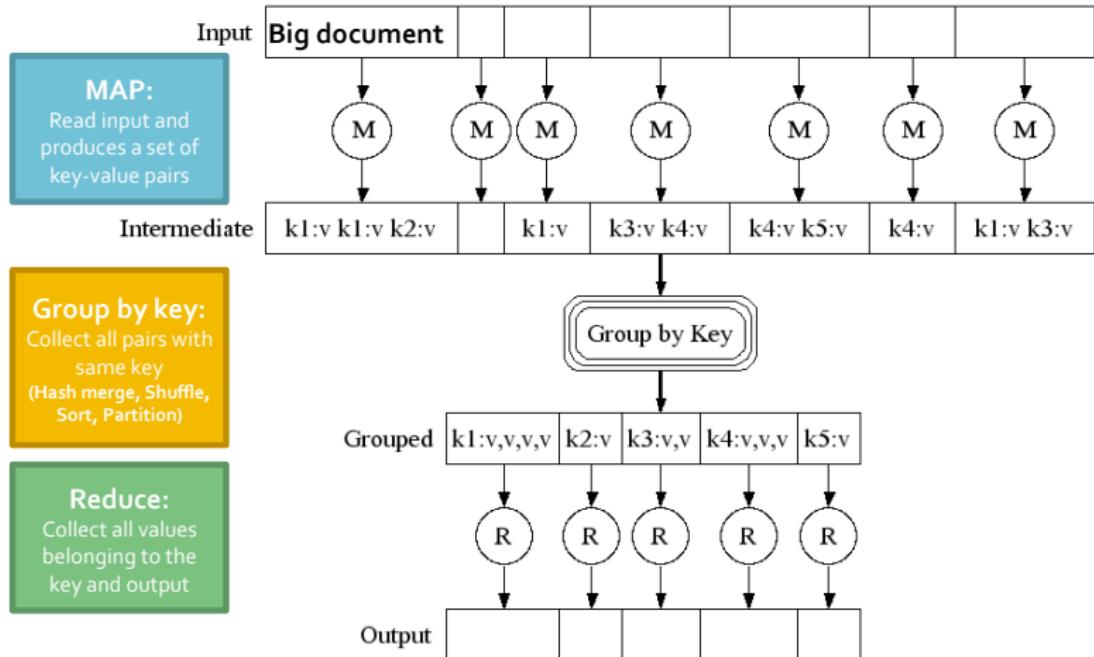
reduce(key, values):
// key: a word; value: an iterator over counts
    result = 0
    for each count v in values:
        result += v
    emit(key, result)
```

Map-Reduce: Environment

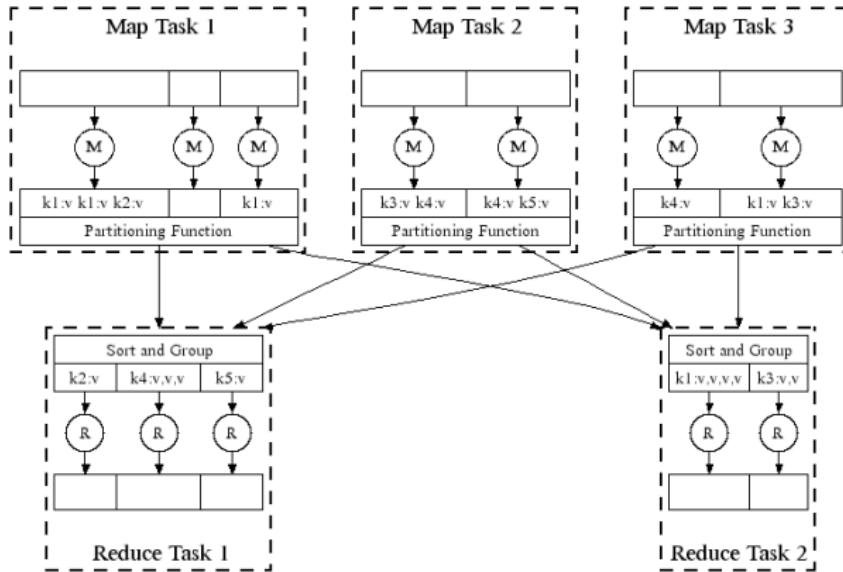
Map-Reduce environment takes care of:

- Partitioning the input data
- Scheduling the program's execution across a set of machines
- Performing the **group by key** step
- Handling machine failures
- Managing required inter-machine communication

Map-Reduce: A diagram



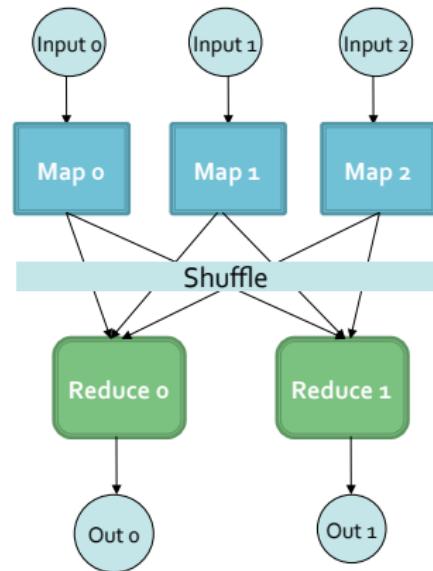
Map-Reduce: In Parallel



All phases are distributed with many tasks doing the work

Map-Reduce

- **Programmer specifies:**
 - Map and Reduce and input files
- **Workflow:**
 - Read inputs as a set of key-value-pairs
 - **Map** transforms input kv-pairs into a new set of k'v'-pairs
 - Sorts & Shuffles the k'v'-pairs to output nodes
 - All k'v'-pairs with a given k' are sent to the same **reduce**
 - **Reduce** processes all k'v'-pairs grouped by key into new k"v"-pairs
 - Write the resulting pairs to files
- **All phases are distributed with many tasks doing the work**



Data Flow

- **Input and final output are stored on a distributed file system (FS):**
 - Scheduler tries to schedule map tasks “close” to physical storage location of input data
- **Intermediate results are stored on local FS of Map and Reduce workers**
- **Output is often input to another MapReduce task**

Coordination: Master

- **Master node takes care of coordination:**
 - **Task status:** (idle, in-progress, completed)
 - **Idle tasks** get scheduled as workers become available
 - When a map task completes, it sends the master the location and sizes of its R intermediate files, one for each reducer
 - Master pushes this info to reducers
- Master pings workers periodically to detect failures

Dealing with Failures

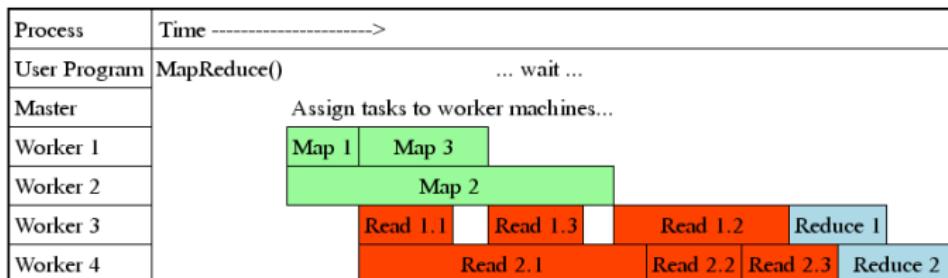
- **Map worker failure**
 - Map tasks completed or in-progress at worker are reset to idle
 - Reduce workers are notified when task is rescheduled on another worker
- **Reduce worker failure**
 - Only in-progress tasks are reset to idle
 - Reduce task is restarted
- **Master failure**
 - MapReduce task is aborted and client is notified

How many Map and Reduce jobs?

- M map tasks, R reduce tasks
- **Rule of a thumb:**
 - Make M much larger than the number of nodes in the cluster
 - One DFS chunk per map is common
 - Improves dynamic load balancing and speeds up recovery from worker failures
- **Usually R is smaller than M**
 - Because output is spread across R files

Task Granularity & Pipelining

- **Fine granularity tasks:** map tasks >> machines
 - Minimizes time for fault recovery
 - Can do pipeline shuffling with map execution
 - Better dynamic load balancing



Refinements: Backup Tasks

■ Problem

- Slow workers significantly lengthen the job completion time:
 - Other jobs on the machine
 - Bad disks
 - Weird things

■ Solution

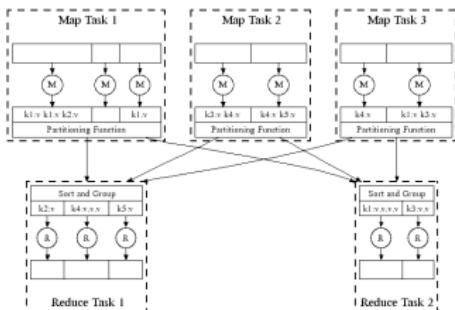
- Near end of phase, spawn backup copies of tasks
 - Whichever one finishes first “wins”

■ Effect

- Dramatically shortens job completion time

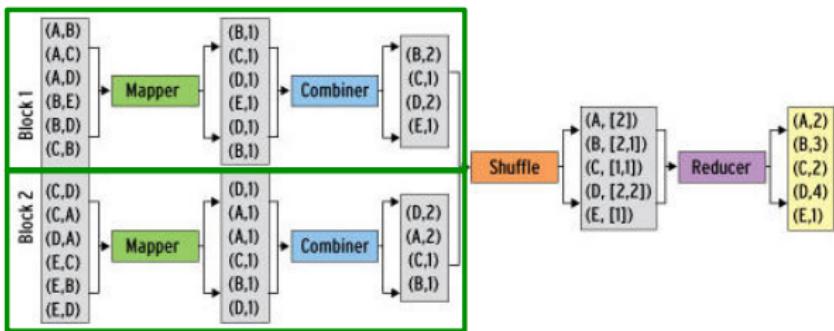
Refinement: Combiners

- Often a Map task will produce many pairs of the form $(k, v_1), (k, v_2), \dots$ for the same key k
 - E.g., popular words in the word count example
- Can save network time by pre-aggregating values in the mapper:**
 - $\text{combine}(k, \text{list}(v_1)) \rightarrow v_2$
 - Combiner is usually same as the reduce function
- Works only if reduce function is commutative and associative



Refinement: Combiners

- Back to our word counting example:
 - Combiner combines the values of all keys of a single mapper (single machine):



- Much less data needs to be copied and shuffled!

Refinement: Partition Function

- Want to control how keys get partitioned
 - Inputs to map tasks are created by contiguous splits of input file
 - Reduce needs to ensure that records with the same intermediate key end up at the same worker
- System uses a default partition function:
 - $\text{hash}(\text{key}) \bmod R$
- Sometimes useful to override the hash function:
 - E.g., $\text{hash}(\text{hostname(URL)}) \bmod R$ ensures URLs from a host end up in the same output file

Problems Suited for Map-Reduce

Example: Host size

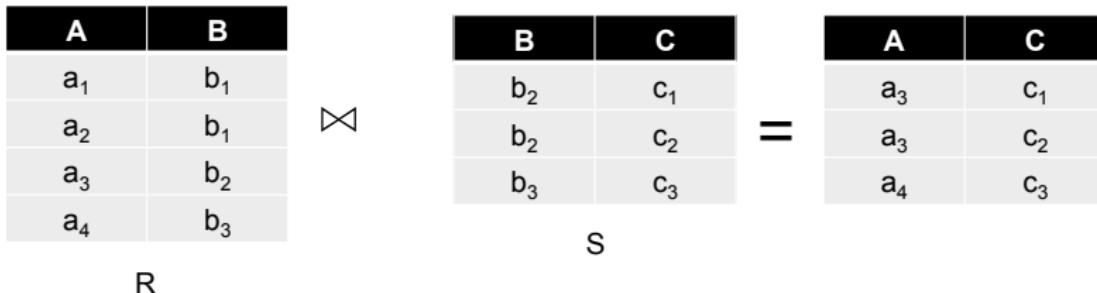
- Suppose we have a large web corpus
- Look at the metadata file
 - Lines of the form: (URL, size, date, ...)
- For each host, find the total number of bytes
 - That is, the sum of the page sizes for all URLs from that particular host
- Other examples:
 - Link analysis and graph processing
 - Machine Learning algorithms

Example: Language Model

- **Statistical machine translation:**
 - Need to count number of times every 5-word sequence occurs in a large corpus of documents
- **Very easy with MapReduce:**
 - **Map:**
 - Extract (5-word sequence, count) from document
 - **Reduce:**
 - Combine the counts

Example: Join By Map-Reduce

- Compute the natural join $R(A,B) \bowtie S(B,C)$
- R and S are each stored in files
- Tuples are pairs (a,b) or (b,c)



Map-Reduce Join

- Use a hash function h from B-values to $1 \dots k$
- A Map process turns:
 - Each input tuple $R(a,b)$ into key-value pair $(b,(a,R))$
 - Each input tuple $S(b,c)$ into $(b,(c,S))$
- Map processes send each key-value pair with key b to Reduce process $h(b)$
 - Hadoop does this automatically; just tell it what k is.
- Each Reduce process matches all the pairs $(b, (a,R))$ with all $(b, (c,S))$ and outputs (a,b,c) .

Cost Measures for Algorithms

- In MapReduce we quantify the cost of an algorithm using
 1. *Communication cost* = total I/O of all processes
 2. *Elapsed communication cost* = max of I/O along any path
 3. (*Elapsed*) *computation cost* analogous, but count only running time of processes

Note that here the big-O notation is not the most useful
(adding more machines is always an option)

Example: Cost Measures

- **For a map-reduce algorithm:**
 - **Communication cost** = input file size + $2 \times$ (sum of the sizes of all files passed from Map processes to Reduce processes) + the sum of the output sizes of the Reduce processes.
 - **Elapsed communication cost** is the sum of the largest input + output for any map process, plus the same for any reduce process

What Cost Measures Mean

- Either the I/O (communication) or processing (computation) cost dominates
 - Ignore one or the other
- Total cost tells what you pay in rent from your friendly neighborhood cloud
- Elapsed cost is wall-clock time using parallelism

Cost of Map-Reduce Join

- **Total communication cost**
 $= O(|R| + |S| + |R \bowtie S|)$
- **Elapsed communication cost** = $O(s)$
 - We're going to pick k and the number of Map processes so that the I/O limit s is respected
 - We put a limit s on the amount of input or output that any one process can have. **s could be:**
 - What fits in main memory
 - What fits on local disk
- With proper indexes, computation cost is linear in the input + output size
 - So computation cost is like comm. cost

Implementations

- Google
 - Not available outside Google
- Hadoop
 - An open-source implementation in Java
 - Uses HDFS for stable storage
 - Download: <http://lucene.apache.org/hadoop/>
- Aster Data
 - Cluster-optimized SQL Database that also implements MapReduce

Data-Flow Systems

- MapReduce uses two ranks of tasks: one for Map the second for Reduce.
 - Data flows from the first rank to the second.
- Generalize in two ways:
 1. Allow any number of ranks.
 2. Allow functions other than Map and Reduce.
- As long as data flow is in one direction only, we can have the blocking property and allow recovery of tasks rather than whole jobs.

Spark

- The most popular implementation of a dataflow system.
- Data passed from one rank of processes to the next forms a *Resilient Distributed Dataset* (RDD).
 - Elements of an RDD are like tuples of a relation.
 - Generalizes (key-value) pairs.
- Built-in operations include Map, Reduce, and also group-aggregate using any components of the RDD type.
 - The thing Hadoop does behind-the-scenes.

Evolution of Big Data Platforms

