1. Information Integration
   - Generally databases in an enterprises have:
     - Several underlying database management systems
       - Oracle, MS SQL Server, DB2, Informix, Sybase (SQL Server), MS Access, etc.
     - Several underlying database schemas
       - Information in an employee table can contain:
         - Employee Name, SSN, DOB, title, hrsPerWeek, modifiedTime, modifiedBy
         - Employee Name, SSN, DOB, title, degree, createTime, createBy
         - Employee Name, SSN, DOB, title, salary, modifiedTime, modifiedBy, createTime, createBy

2. Semi-structured Data
   - A new data model designed to cope with problems of information integration
     - Accommodates of different DBMS
     - Integrates different schemas

3. XML
   - XML: A standard language for describing semi-structured data schemas and representing data.

Framework

1. Information Integration: Making databases from various places work as one.
2. Semi-structured Data: A new data model designed to cope with problems of information integration.
3. XML: A standard language for describing semi-structured data schemas and representing data.

The Information-Integration Problem

- Major bottleneck in enterprise application integration
  - For example,
    - Hewlett Packard split into HP and Agilent
    - HP bought Compaq
  - Need to integrate data from different sources
The Information-Integration Problem

- Related data exists in many places and could, in principle, work together.
- But different databases differ in:
  1. Model (relational, object-oriented?).
  2. Schema (normalized/unnormalized?).
  3. Terminology: are consultants employees? Retirees? Subcontractors?
  4. Conventions (meters versus feet?).

Example

- Consider merger of three stores in a mall
- There is some overlap in the products sold but the databases are different

Two Approaches to Integration

1. Warehousing
   - Makes a copy of the data
   - More developed of the two
2. Mediation
   - Creates a view of the data
   - Newer and less developed

Warehousing

- Make copies of the data sources at a central site and transform it to a common schema.
- Reconstruct data daily/weekly
- Do not try to keep it more up-to-date than that.
- Pro:
  - Very well-developed, and several commercial tools are available
- Con:
  - Data can be old since updates are expensive

Mediation

- Create a view of all sources, as if they were integrated.
- Answer a view query by translating it to terminology of the sources and querying them.
- Pro:
  - Current data
- Con:
  - Can be slow
  - Availability of tools
Semi-structured: Motivation

- Most effective approach to Information Integration:
  - Semi-structured Data Model
  - or Semi-structured Objects

Semi-structured: Motivation

- Main limitation of Object-Oriented Models: Object Models are Strongly Typed
  - Objects of a class have one structure only
- Semi-structured approach solves this problem

Semi-structured Data

- Purpose:
  - Represent data from independent sources more flexibly than
    - either relational
    - or object-oriented models.

Semi-structured Data

- Each object has a class of their own and properties are defined whatever labels are attached to that object
  - Properties mean attributes, relationships, methods, etc.
Semi-structured Data

- Think of objects, but with the type of each object its own business, not that of its “class.”
- Labels to indicate meaning of substructures.

Semi-structured Graphs

- Easy to think of Semi-structured data as Graphs
  - Nodes = objects.
  - Labels on arcs:
    - attributes leading to a leaf node
    - Relationships leading to another node.

Semi-structured Graphs

- Atomic values at leaf nodes
  - nodes with no arcs out.
- Flexibility: no restriction on:
  - Labels out of a node.
  - Number of successors with a given label.

Example: Data Graph

- Root object represents the entire DB. Often look like trees, but are not.
- Notice a new kind of data.

XML

- XML = Extensible Markup Language.
- While HTML uses tags for formatting (e.g., “italic”), XML uses tags for semantics (e.g., “this is an address”).
- Key idea: create tag sets for a domain (e.g., genomics), and translate all data into properly tagged XML documents.

Well-Formed and Valid XML

- Well-Formed XML allows you to invent your own tags.
  - Similar to labels in semi-structured data graph.
- Valid XML involves a DTD (Document Type Definition), which
  - gives a grammar for the use of labels
  - limits the set of labels our of node
  - the order and number of times a label occurs
Well-Formed XML: Header
- Start the document with a declaration, surrounded by `<? ... ?>`.
- Normal declaration for Well-Formed XML is:
  ```xml
  <? XML VERSION = "1.0" STANDALONE = "yes" ?>
  ```
  - Version indicates version number
  - Standalone = "yes" means no DTD provided.

Well-Formed XML: Body
- Body of document is a root tag surrounding nested tags.
- Body can include:
  - several properly matching tags (as in html structure)
  - Root tag can have a special meaning such as document type or can be generic

Tags
- Tags, as in HTML, are normally matched pairs, as
  ```xml
  <BLAH> ... </BLAH>.
  ```
- Tags may be nested arbitrarily.
- Some tags requiring no matching ender, such as `<P>` in HTML, are also permitted.
- however, we will not use these in examples

Example: Well-Formed XML
- Root tag RESTS surrounds the entire document
- `<NAME>` tag specifies the REST name
- `<SODA>` tags have names and price for each Soda nested in `<NAME>` and `<PRICE>` tags.

XML and Semi-structured Data
- Well-Formed XML documents with nested tags is exactly the same idea as trees of semi-structured data.
- Tags are the labels on edges
- Nodes represent data between matching tags
- Parent-child relationship is immediate nesting in XML

XML and Semi-structured Data
- Semi-structured approach allows for non-tree structures
- We shall see that XML also enables non-tree structures, as does the semi-structured data model.
Exercise

Convert the following into a Semi-structured representation

```xml
<?xml version = "1.0" standalone = "yes" ?>
<RESTS>
<REST>
  <NAME>Taco Bell</NAME>
  <SODA><NAME>Pepsi</NAME>
  <PRICE>1.00</PRICE></SODA>
  <SODA><NAME>Sobe</NAME>
  <PRICE>2.00</PRICE></SODA>
</REST>
<REST>
<REST> ...
</REST>
</RESTS>
```

Solution

The `<RESTS>` XML document is:

```
<RESTS>
  <REST>
    <NAME>KFC</NAME>
    <SODA><NAME>Pepsi</NAME>
    <PRICE>1.00</PRICE></SODA>
    <SODA><NAME>Sobe</NAME>
    <PRICE>2.00</PRICE></SODA>
  </REST>
</RESTS>
```

Document Type Definitions

- Most interesting use of XML: Valid XML
  - Essentially a context-free grammar for describing XML tags and their nesting.
  - Each domain of interest creates one DTD that describes all the documents this group will share.
  - For example, electronic components, travel industry, etc., will have their own DTDs

DTD Structure

```
<!DOCTYPE <root tag> [
  <!ELEMENT <name> ( <components> )>
  <more elements>
]>
```

Example: DTD

```
<!DOCTYPE Rests [ 
  <!ELEMENT RESTS (REST*)>
  <!ELEMENT REST (NAME, SODA+)> 
  <!ELEMENT NAME (#PCDATA)> 
  <!ELEMENT SODA (NAME, PRICE)> 
  <!ELEMENT PRICE (#PCDATA)> 
]>
```

DTD Elements

- Element definition consists of its name (tag), and a parenthesized description of any nested tags.
  - includes order of subtags
  - and their multiplicity (0, 1, many times).
- Leaves (text elements) have #PCDATA in place of nested tags.
Element Descriptions Rules

- Subtags must appear in order shown.
- A tag may be followed by a symbol to indicate its multiplicity:
  - * = zero or more.
  - + = one or more.
  - ? = zero or one.
- Symbol | can connect alternative sequences of tags.

Example: Element Description

A name is

- Either an optional title (e.g., “Dr.”), a first name, and a last name, in that order,
- or it is an IP address:

```xml
<!ELEMENT NAME (TITLE?, FIRST, LAST) | IPADDR )>
```

Use of DTDs

- In order to specify a document follows a particular DTD
  1. Set STANDALONE = “no”.
  2. Either:
     a) Include the DTD as a preamble of the XML document, or
     b) Follow DOCTYPE and the <root tag> by SYSTEM and a path to the file where the DTD is stored.

Example (a)

```xml
<? XML VERSION = "1.0" STANDALONE = "no" ?>
<!DOCTYPE Rests [...
<!ELEMENT REST (NAME, SODA+)>
<!ELEMENT NAME (#PCDATA)>
<!ELEMENT SODA (NAME, PRICE)>
<!ELEMENT PRICE (#PCDATA)>
]>
<RESTS>
<REST><NAME>Taco Bell</NAME>
<SODA><NAME>Pepsi</NAME> <PRICE>1.00</PRICE></SODA>
<SODA><NAME>Sobe</NAME> <PRICE>2.00</PRICE></SODA>
</REST>
<REST> ...
</RESTS>
```

Example (b)

- Assume the RESTS DTD is in file rest.dtd

```xml
<? XML VERSION = "1.0" STANDALONE = "no" ?>
<!DOCTYPE Rests SYSTEM "rest.dtd">
<RESTS>
<REST><NAME>Taco Bell</NAME>
<SODA><NAME>Pepsi</NAME> <PRICE>1.00</PRICE></SODA>
<SODA><NAME>Sobe</NAME> <PRICE>2.00</PRICE></SODA>
</REST>
<REST> ... ...
</RESTS>
```

Attributes

- Attributes are another important component of DTD and XML docs
- Opening tags in XML can have attributes, like `<A HREF = "...">` in HTML.
- In a DTD,

```xml
<!ATTLIST <element name>...
```

gives a list of attributes and their data types for this element.
Example: Attributes
- Rests can have an attribute `kind`, which is either `qsr`, family, or other.
- The element definition is unchanged.
- However, we add an ATTLIST.
  ```xml
  <!ELEMENT REST (NAME SODA*)>
  <!ATTLIST REST kind = "qsr" | "family" | "other">
  ```

Example: Attribute Use
- In a document that allows REST tags, we might see:
  ```xml
  <REST kind = "qsr">
    <NAME>KFC</NAME>
    <SODA><NAME>Sierra Mist</NAME>
    <PRICE>1.00</PRICE></SODA>
  ...
  </REST>
  ```

ID’s and IDREFs
- Introduce links from one object to another.
- Allows the structure of an XML document to be a general graph, rather than just a tree.
- These are pointers from one object to another, in analogy to HTML’s `NAME = “blah”` and `HREF = “#blah”`.

Creating ID’s
- We give an element `Elephant` an attribute `Attention` of type ID in the DTD.
- When using tag `<Elephant>` in an XML document, give its attribute `Attention` a unique value. For example,
  ```xml
  <Elephant  Attention = “213”>
  ```

Creating IDREFs
- IDREFs are similar to IDs:
  - To allow objects of type `Fig` to refer to another object with an ID attribute, give `Fig` an attribute of type IDREF (single string of type ID).
  - Or, let the attribute have type IDREFS, so the `Fig`-object can refer to any number of other objects (any number strings of type ID).

Example: ID’s and IDREFs
- Let’s redesign our RESTS DTD to include both REST and SODA sub-elements.
- Both rests and sodas will have ID attributes called `name`.
- Rests have `PRICE` sub-objects, consisting of a number (the price of one soda) and an IDREF `theSoda` leading to that soda.
- Sodas have attribute `soldBy`, which is an IDREFS leading to all the rests that sell it.
**The DTD**

```xml
<!DOCTYPE Rests [
  <!ELEMENT RESTS (REST*, SODA*)>
  <!ELEMENT REST (PRICE*)>
  <!ATTLIST REST name = ID>
  <!ELEMENT PRICE (#PCDATA)>
  <!ATTLIST PRICE theSoda = IDREF>
  <!ELEMENT SODA ()>
  <!ATTLIST SODA name = ID, soldBy = IDREFS>
]>```

REST objects have name as an ID attribute and have one or more PRICE sub-objects.

PRICE objects have a number (the price) and one reference to a soda.

Soda objects have an ID attribute called name, and a soldBy attribute that is a set of Rest names.

**Example Document**

```xml
<RESTS>
  <REST name = "KFC">
    <PRICE theSoda = "Pepsi">1.00</PRICE>
    <PRICE theSoda = "Sobe">2.00</PRICE>
  </REST> …
  <SODA name = "Pepsi", soldBy = "KFC, TacoBell,..."/>
</RESTS>```