CHAPTER 5
SEMI-STRUCTURED DATA

Section 1. Introduction

Structured Data

- Data is represented in a strict format.
- Data is organized in semantic units known as tuples or entities.
- Similar entities are grouped together into relations.
- Entities in the same relation have the same attributes.
- The order of the attributes for a relation is important.
- Schema constructs (names of relations, attributes, etc.) exist separate from the data.

With structured data, as found in relational databases, it is common to design the database schema first and then populate the database with data afterwards. The DBMS will check to ensure that all of the data follows the structures and constraints as specified in the schema.

Unstructured Data

- Data can be of any type.
- Not necessarily following any format or sequence.
- Does not follow any rules.
- Is not predictable.
- Examples include text, video, sound, images.

Semi-structured Data

- Data is organized in "semantic entities."
- Similar entities are grouped together.
- Entities in the same group may not have the same attributes.
- The order of attributes is not necessarily important.
- Not all attributes may be required.
- The size of the same attributes in a group may differ.
- The type of the same attributes in a group may differ.

With semi-structured data the information that is normally associated with a schema is contained within the data, which is sometimes called "self-describing." There is no clear separation between the data and the schema, and the degree to which the data is structured depends on the application. In some forms of semi-structured data there is no separate schema. In others the schema exists but only places loose constraints on the data.

- Semi-structured data can be naturally modeled in terms of graphs which contain labels that give semantics to its underlying structure.

Semi-structured data has emerged as an important topic of study for a variety of reasons:

1. There are data sources such as the Web, which we would like to treat as databases but which cannot be constrained by a schema.

2. It may be desirable to have an extremely flexible format for data exchange between disparate databases.

3. Even when dealing with structured data, it may be helpful to view it as semi-structured for the purposes of browsing.

Common Sources of semi-structured Data:

- E-mails
- XML and other markup languages
- Binary executables
- TCP/IP packets
- Zipped files
- Integration of data from different sources
- Web pages

Advantages of Semi-structured Data:

- The data is not constrained by a fixed schema
- Flexible - the Schema can be easily changed.
- Data is portable
- It is possible to view structured data as semi-structured data
- Its supports users who can not express their need in SQL
- It can deal easily with the heterogeneity of sources.

Disadvantages of Semi-structured data

- The lack of a fixed, rigid schema makes it difficult in storage of the data
- Interpreting any relationships between data is difficult as there is no separation of the schema and the data.
- Queries are less efficient as compared to structured data.

Problems faced in storing semi-structured data

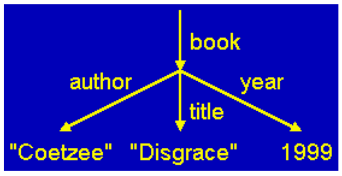
- Data usually has an irregular and partial structure. Some sources have implicit structure of data, which makes it difficult to interpret the relationship between data.
- Schema and data are usually tightly coupled i.e they are not only linked together but are also dependent of each other. Same query may update both schema and data with the schema being updated frequently.
- Distinction between schema and data is very uncertain or unclear. This complicates the designing

Section 2. Data Models for Semi-Structured Data

Object Exchange Model (OEM)

- a rooted, labeled, directed graph
- edge labels map to strings; these constitute the "self-describing data"
- only its leaf nodes have labels which map to data values
- no ordering of edges leaving a node

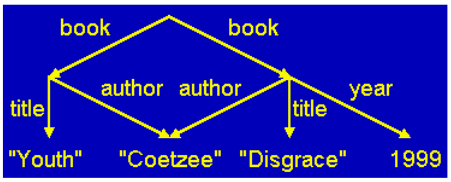
Example:



Here we represent the above graph may be written using a text-based/serialization syntax such as

```
{ book: { author: "Coetzee",
          title: "Disgrace",
          year: 1999
        }
}
```

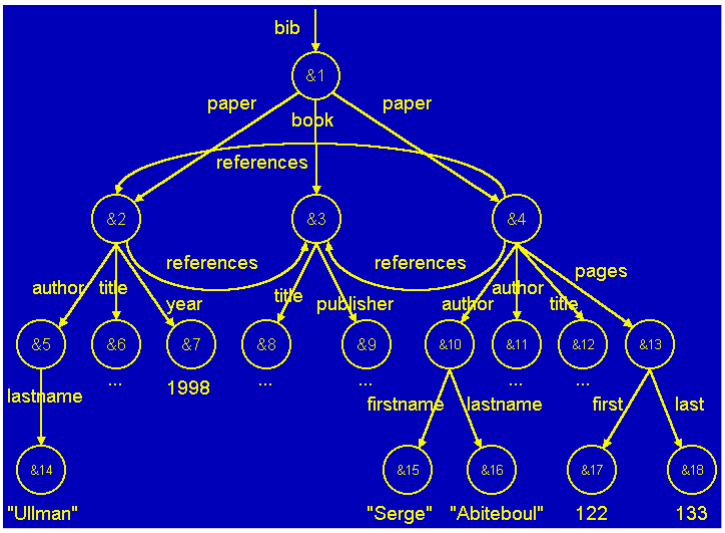
Note, labels can be repeated, e.g., for multiple books




```
{ book: { author: "Coetzee",
          title: "Disgrace",
          year: 1999
        }
  book: { author: "Coetzee",
          title: "Youth",
        }
}
```

(here, we need a "root" label, for example "wks" -- for "works," but do not show it. This will appear in our next example)

The graphs in the above structure are really a tree; but they don't have to be. To handle graphs that are not trees we introduce *object identifiers* (*oid*s) for nodes.




```
{bib: &1
   { paper: &2 { ... },
     book:  &3 { ... },
     paper: &4
          { author: &10
                  { firstname: &15 "Serge",
                    lastname: &16 "Abiteboul" },
            author: &11 { ... }
            title: &12 { ... }
            pages: &13
                 { first: &17 122,
                   last: &18 133 },
            references: &2,
            references: &3
          }
   }
}
```

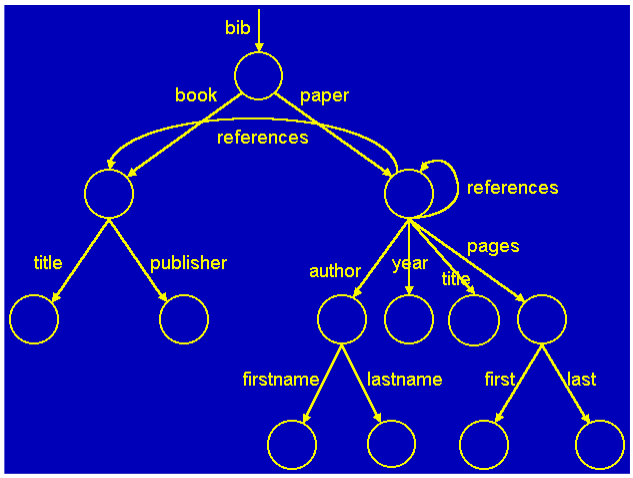
Schema Graphs

Given some semi-structured data, we might want to extract a schema that describes it. Such a schema would be useful for

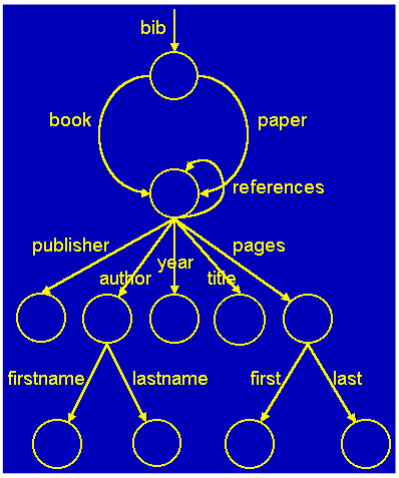
- browsing the data by types
- optimizing queries by reducing the number of paths searched
- improving storage of data

A *schema graph* specifies what edges are *permitted* in a data graph

- every path in the data graph occurs in the schema graph



A less specific schema graph



XML

XML (eXtensible Markup Language), as defined by the World Wide Web Consortium in 1998, was originally developed to provide a means of marking up a document or character stream to identify structural or other units within the data. It was quickly realized, however, that XML can be used to represent semi-structured data. Examining XML will be the major emphasis in this chapter.

Section 3. Introduction to XML

XML (eXtensible Markup Language) provides a general approach for representing all types of information such as data sets containing numerical and categorical variables; spreadsheets; visual graphical displays; descriptions of user interfaces, social network structures, text documents such as word processing documents and presentation documents, *RSS* (Rich Site Summary or alternatively Real Simple Syndication) feeds, data sent to and fromWeb services, settings and preferences on computers*, etc.*

Over the last twenty years, *XML* has grown from a proposed simplification of *SGML (Standard Generalized Markup Language)* to a widely adopted and used technology in a multitude of areas. This ubiquity and broad set of applications wherein it is used makes a compelling case for why anyone working with data needs to have some familiarity with *XML*.

A markup language is simply a mechanism for identifying structures in a document. The XML specification defines a standard way to add markup to documents.

XML documents are made up of storage units called *entities*, which contain either parsed or unparsed data. Parsed data is made up of characters, some of which form character data, and some of which form markup. Markup encodes a description of the document's storage layout and logical structure. XML provides a mechanism to impose constraints on the storage layout and logical structure.

A software module called an ***XML processor*** is used to read XML documents and provide access to their content and structure.

XML Syntax

Syntactically, XML documents look like HTML documents. A *well-formed* XML document—one that conforms to the XML syntax—starts with a *prolog* and contains exactly one *element*. A simple prolog is the following (and even here the "`encoding="UTF-8"` part can be omitted):

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

The single element can be viewed as the root of the document. Additional elements are nested inside the root element, and attributes can be attached to them. Attribute values must be in quotes, and tags must be balanced. Empty element tags must either end with a /> or be explicitly closed.

Elements

The basic unit in *XML* is the *element*, which is also refer to as a *node* when emphasizing the hierarchical or treelike structure of the *XML* document.

Each element begins with a *start-tag* which has the form <*tagname*> and concludes with a matching *end-tag*, whose structure is the same as the opening tag except with a forward slash (/) preceding the name. For example a start tag <*title*>. would have an end tag <*/title*> and a start tag<*article*>,would have an end tag <*/article*>.

- Thus, tags are paired and they delimit an element and its contents.

- In many respects, pairs of opening and ending tags are like parentheses, but with names that make it easier to identify the pairs when the elements are nested hierarchically.

Child Elements

XML elements can have content made up of other *XML* elements that are treated as *child elements*.

It is this nested/recursive structure that allows us to represent different, complex data structures using

XML.

Example

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

<BookList>
  <Book>
    <Author>Coetzee</Author>
    <Title>Disgrace</Title>
    <Year>1999</Year>
  </Book>
</Booklist>
```

Example

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

<BookList>
  <Book>
    <Author>Coetzee</Author>
    <Title>Disgrace</Title>
    <Year>1999</Year>
  </Book>
  <Book>
    <Author>Coetzee</Author>
    <Title>Youth</Title>
  </Book>
</Booklist>
```

Fundamental XML Rules

- All XML elements must have an opening and a closing tag
- XML tags are case sensitive
- XML documents must have a root element
- XML elements must be properly nested
- Comments are allowed in XML

  <!-- This is a comment -->

An element can have element content, simple content (text) mixed content (element and simple content), and even empty content. An element can also have attributes, which we will show in pur example below, and then describe shortly).

Example:

```
<?xml version="1.0" ?>

<BookList>
      <book>
         <title>My First XML</title>
         <prod id="33-657" media="paper"></prod>
         <chapter>Introduction to XML
            <para>What is HTML</para>
            <para>What is XML</para>
         </chapter>

         <chapter>XML Syntax
            <para>Elements must have a closing tag</para>
            <para>Elements must be properly nested</para>
         </chapter>
      </book>
</BookList>
```

In the example above, the element ***book*** has *element content*, because it contains other elements. The element ***chapter*** has *mixed conten*t because it contains both text and other elements. The element ***para*** has *simple content* (or text content) because it contains only text. The element ***prod*** has *empty content*, because it carries no information.

Also, in the example above only the ***prod*** element has *attributes*.

Attributes

Attributes generally provide additional information about an element. Attribute values must always be enclosed in quotes, but either single or double quotes can be used.

In our example above, the attribute named ***id*** has the *value* "33-657". The attribute named ***media*** has the value "paper".

Should One Use Child Elements or Attributes to Store Data?

There is sometimes disagreement over whether to use attributes or put the values as children within an element. For example, we could have represented the *prod* element in the above example in an equivalent form as

```
<prod>
   <id>=33-657</id>
   <media>paper</media>
</prod>
```

We typically use an *XML* attribute when giving data about the data (that is, metadata) within the element. The attributes keep this information separate from the content.