Database Systems

Lecture 9: Complex Data Types

Dr. Momtazi
momtazi@aut.ac.ir

Outline

- **Semi-Structured Data**
  - XML
  - JSON
  - RDF
- Object Orientation
- Textual Data
- Spatial Data

Semi-Structured Data

- Many applications require storage of complex data, whose schema changes often
- The relational model's requirement of atomic data types may be an overkill
  - E.g., storing set of interests as a set-valued attribute of a user profile may be simpler than normalizing it
- Data exchange can benefit greatly from semi-structured data
  - Exchange can be between applications, or between back-end and front-end of an application
  - Web-services are widely used today, with complex data fetched to the front-end and displayed using a mobile app or JavaScript
- JSON and XML are widely used semi-structured data models

Features of Semi-Structured Data Models

- **Flexible schema**
  - **Wide column** representation: allow each tuple to have a different set of attributes, can add new attributes at any time
  - **Sparse column** representation: schema has a fixed but large set of attributes, by each tuple may store only a subset

- **Multivalued data types**
  - **Sets**, **multisets**
    - E.g.,: set of interests {‘basketball, ‘La Liga’, ‘cooking’, ‘anime’, ‘jazz’}
  - **Key-value map** (or just **map** for short)
    - Store a set of key-value pairs
    - E.g., {(brand, Apple), (ID, MacBook Air), (size, 13), (color, silver)}
    - Operations on maps: *put*(key, value), *get*(key), *delete*(key)

Features of Semi-Structured Data Models

- **Arrays**
  - Widely used for scientific and monitoring applications
  - E.g., readings taken at regular intervals can be represented as array of values instead of (time, value) pairs
    - [5, 8, 9, 11] instead of {(1,5), (2, 8), (3, 9), (4, 11)}
- Multi-valued attribute types
  - Modeled using *non first-normal-form* (*NFNF*) data model
  - Supported by most database systems today
- **Array database**: a database that provides specialized support for arrays
  - E.g., compressed storage, query language extensions etc
  - Oracle GeoRaster, PostGIS, SciDB, etc

Nested Data Types

- Hierarchical data is common in many applications
- JSON: JavaScript Object Notation
  - Widely used today
- XML: Extensible Markup Language
  - Earlier generation notation, still used extensively

Motivation

- Each application area has its own set of standards for representing information

- XML/JSON have become the basis for all new generation data interchange formats

- Earlier generation formats were based on plain text with line headers indicating the meaning of fields
  - Similar in concept to email headers
  - Does not allow for nested structures, no standard “type” language
  - Tied too closely to low level document structure (lines, spaces, etc)

Motivation

- Data interchange is critical in today's networked world
  - Examples:
    - Banking: funds transfer
    - Order processing (especially inter-company orders)
    - Scientific data
      - Chemistry: ChemML, …
      - Genetics: BSML (Bio-Sequence Markup Language), …
  - Paper flow of information between organizations is being replaced by electronic flow of information

Example

- Purchase orders are typically generated by one organization and sent to another.
- Traditionally they were printed on paper by the purchaser and sent to the supplier; the data would be manually re-entered into a computer system by the supplier.
- This slow process can be greatly sped up by sending the information electronically between the purchaser and supplier.
- XML/JSON provides a standard way of tagging the data.
- The two organizations must of course agree on what tags appear in the purchase order, and what they mean

Motivation for Nesting

- Each purchase order has a purchaser and a list of items as two of its nested structures.
  - Each item in turn has an item identifier, description and a price nested within it.
  - The purchaser has a name and address nested within it.

- Such information are normally split into multiple relations in a relational schema:
  - Item information
  - Purchaser information
  - Purchase orders
  - The relationship between purchase orders, purchasers, and items

Motivation for Nesting

- The relational representation helps to avoid redundancy.
  - E.g., item descriptions would be stored only once for each item identifier in a normalized relational schema.

- In the XML purchase order, however, the descriptions may be repeated in multiple purchase orders that order the same item.

- Gathering all information related to a purchase order into a single nested structure, even at the cost of redundancy, is attractive when information has to be exchanged with external parties.

Applications

- Storing data with complex structures
  - Applications that need to store data that are structured, but are not easily modeled as relations
    - E.g., user preferences stored by a Web browser

  - Editable document representation as part of office application packages for storing documents
    - E.g., Open Office: Open Document Format (ODF)
    - E.g., Microsoft Office: Office Open XML (OOXML)

Applications

- Standardized data exchange formats
  - E.g., ChemML: a standard for representing/exchanging information about chemicals
    - Molecular structure
    - Boiling and melting points
    - Calorific values
    - Solubility in various solvents
  - E.g., required information for business-to-business (B2B) market
    - Product descriptions
    - Price information
    - Product inventories
    - Quotes for a proposed sale

Outline

- Semi-Structured Data
  - **XML**
    - Structure of XML Data
    - XML Document Schema
    - Querying and Transformation
    - Storage of XML Data
  - JSON
  - RDF
- Object Orientation
- Textual Data
- Spatial Data

Introduction

- XML: Extensible Markup Language
- Defined by the WWW Consortium (W3C)
- Derived from SGML (Standard Generalized Markup Language), but simpler to use than SGML

Introduction

- Markup: anything in a document that is not intended to be part of the printed output.

- Such notes convey extra information about the text.
  - For example, a typeset in a magazine may want to make notes about how the typesetting should be done. These notes should be distinguished from the actual content, so that they do not end up printed in the magazine.

- In electronic document processing, a markup language is a formal description of what part of the document is content, what part is markup, and what the markup means.

Introduction

- Documents have tags giving extra information about sections of the document
  - E.g.

    <title> XML </title>

    <slide> Introduction …</slide>

- **Extensible**, unlike HTML
  - Users can add new tags, and *separately* specify how the tag should be handled for display

Introduction

- The ability to specify new tags, and to create nested tag structures make XML a great way to exchange **data**, not just documents.
  - Much of the use of XML has been in data exchange applications, not as a replacement for HTML
- Tags make data (relatively) self-documenting
  - E.g.

```
<university>
    <department>
      <dept_name> Comp. Sci. </dept_name>
      <building> Taylor </building>
      <budget> 100000 </budget>
    </department>
    <course>
      <course_id> CS-101 </course_id>
      <title> Intro. to Computer Science </title>
      <dept_name> Comp. Sci </dept_name>
      <credits> 4 </credits>
    </course>
</university>
```

Example of XML Data

```
<purchase_order>
    <identifier> P-101 </identifier>
    <purchaser>  .... </purchaser>
    <itemlist>
      <item>
          <identifier> RS1 </identifier>
          <description> Atom powered rocket sled </description>
          <quantity> 2 </quantity>
          <price> 199.95 </price>
      </item>
      <item>
          <identifier> SG2 </identifier>
          <description> Superb glue </description>
          <quantity> 1 </quantity>
          <unit-of-measure> liter </unit-of-measure>
          <price> 29.95 </price>
      </item>
     </itemlist>
    <total cost> 429.85 </total cost>
    <payment terms> Cash-on-delivery </payment terms>
</purchase_order>
```

Motivation

- Each XML based standard defines what are valid elements, using
  - XML type specification languages to specify the syntax
    - DTD (Document Type Descriptors)
    - XML Schema
  - Plus textual descriptions of the semantics

- XML allows new tags to be defined as required
  - However, this may be constrained by DTDs

- A wide variety of tools is available for parsing, browsing and querying XML documents/data

Comparison with Relational Data

- Inefficient: tags, which in effect represent schema information, are repeated

- Better than relational tuples as a data-exchange format
  - Unlike relational tuples, XML data is self-documenting due to presence of tags
  - Non-rigid format: tags can be added or ignored
    - E.g., <unit-of-measure>
  - Allows nested structures
  - Wide acceptance, not only in database systems, but also in browsers, tools, and applications

XML Features

- Data Exchange
- Hierarchical Structure
- Extensibility
- Self-Describing
- Integrating XML with Relational Databases

XML Features

- An XML tree starts at a root element and branches from the root to child elements.
- All elements can have sub elements (child elements)

```
<?xml version="1.0"?>
<shipto>
  <name>John Smith</name>
  <address>123 Main ST</address>
  <city>Boise</city>
  <country>USA</country>
</shipto>
```

Root element

Child elements

XML Features

- XML is extensible, in that element can be easily added as needed.

```
<?xml version="1.0"?>
<shipto>
  <name>John Smith</name>
  <address>123 Main ST</address>
  <city>Boise</city>
  <country>USA</country>
</shipto>
```

```
<?xml version="1.0"?>
<shipto>
  <name>John Smith</name>
  <address>123 Main ST</address>
  <city>Boise</city>
  <state>ID</state>
  <country>USA</country>
</shipto>
```

XML Features

- XML is self-describing (sort of) with the use of element tags
  - Human-readable format
  - Tags describe the content of the element (sort of)

From reading the tags, it's pretty clear that we're talking about a "Ship To" address that contains the name, address, city & country.

```xml
<?xml version="1.0"?>
<shipto>
  <name>John Smith</name>
  <address>123 Main ST</address>
  <city>Boise</city>
  <country>USA</country>
</shipto>
```

But it doesn't provide full metadata, e.g.:

- **What's the data type?**
- **What's the business definition?**
- **Is <name> a required field?**

XML Features

- Similar to DDL, an XML Schema (XSD) defines the structure & format of data

Metadata

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="shiporder">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="orderperson" type="xs:string"/>
      <xs:element name="shipto">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="address" type="xs:string"/>
            <xs:element name="city" type="xs:string"/>
            <xs:element name="country" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
          </xs:sequence>
    <xs:attribute name="orderid" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>
</xs:schema>
```

XSD

Data

```xml
<?xml version="1.0"?>
<shipto>
  <name>John Smith</name>
  <address>123 Main ST</address>
  <city>Boise</city>
  <country>USA</country>
</shipto>
```

XML

Data

Order Shipment

Ship to:
John Smith
123 Main ST
Boise
USA

.....
.....

XML Features

- XML is often used in conjunction with relational databases for permanent storage and integration with other operational, reporting, and reference data.

Outline

- Semi-Structured Data
  - **XML**
    - **Structure of XML Data**
    - XML Document Schema
    - Querying and Transformation
    - Storage of XML Data
  - JSON
  - RDF
- Object Orientation
- Textual Data
- Spatial Data

Structure of XML Data

- **Tag**: label for a section of data
- **Element**: section of data beginning with <*tagname*> and ending with matching </*tagname*>
- Every document must have a single top-level root element
  - That encompasses all other elements in the document
  - E.g., <university> ... </university>

- Elements must be properly nested
  - Proper nesting
    - <course> … <title> …. </title> </course>
  - Improper nesting
    - <course> … <title> …. </course> </title>
  - Formally: every start tag must have a unique matching end tag, that is in the context of the same parent element.

Structure of XML Data

- Mixture of text with sub-elements is legal in XML.
  - Example:

```
<course>
    This course is being offered for the first time in 2009.
    <course id> BIO-399 </course id>
    <title> Computational Biology </title>
    <dept name> Biology </dept name>
    <credits> 3 </credits>
</course>
```

  - Useful for document processing rather than data processing which deals with structured data

Attributes

- Elements can have **attributes**

```
<course course_id= “CS-101”>
    <title> Intro. to Computer Science</title>
    <dept name> Comp. Sci. </dept name>
    <credits> 4 </credits>
</course>
```

- Attributes are specified by *name=value* pairs inside the starting tag of an element
- An element may have several attributes, but each attribute name can only occur once

```
<course  course_id = “CS-101”  credits=“4”>
```

Attributes vs. Subelements

- Distinction between subelement and attribute
  - In the context of documents, attributes are part of markup, while subelement contents are part of the basic document contents
  - In the context of data representation, the difference is unclear and may be confusing
    - Same information can be represented in two ways
      - <course course_id= “CS-101”> … </course>
      - <course>
        <course_id>CS-101</course_id> …
        </course>
  - Suggestion: use attributes for identifiers of elements, and use subelements for other contents

Namespaces

- XML data has to be exchanged between organizations
- Same tag name may have different meaning in different organizations, causing confusion on exchanged documents
- Specifying a unique string as an element name avoids confusion
- Better solution: use unique-name:element-name
- Avoid using long unique names all over document by using XML Namespaces

```
<university xmlns:yale="http://www.yale.edu">
    ...
     <yale:course>
        <yale:course_id> CS-101 </yale:course_id>
        <yale:title> Intro. to Computer Science</yale:title>
        <yale:dept_name> Comp. Sci. </yale:dept_name>
        <yale:credits> 4 </yale:credits>
     </yale:course>
    ...
</university>
```

More on XML Syntax

- Elements without subelements or text content can be abbreviated by ending the start tag with a /> and deleting the end tag
  - <course course_id="CS-101" Title="Intro. To Computer Science" dept_name = "Comp. Sci." credits="4" />
- To store string data that may contain tags, without the tags being interpreted as subelements, use CDATA as below
  - <![CDATA[<course> … </course>]]>

  Here, <course> and </course> are treated as just strings

  CDATA stands for "character data"

Outline

- Semi-Structured Data
  - **XML**
    - Structure of XML Data
    - **XML Document Schema**
    - Querying and Transformation
    - Storage of XML Data
  - JSON
  - RDF
- Object Orientation
- Textual Data
- Spatial Data

XML Document Schema

- Database schemas constrain what information can be stored, and the data types of stored values
- XML documents are not required to have an associated schema
- However, schemas are very important for XML data exchange
  - Otherwise, a site cannot automatically interpret data received from another site
- Two mechanisms for specifying XML schema
  - **Document Type Definition (DTD)**
    - Widely used
  - **XML Schema**
    - Newer, increasing use

Document Type Definition (DTD)

- The type of an XML document can be specified using a DTD
- DTD constraints structure of XML data
  - What elements can occur
  - What attributes can/must an element have
  - What subelements can/must occur inside each element, and how many times.
- DTD does not constrain data types like integer ...
  - All values represented as strings in XML
- DTD syntax
  - <!ELEMENT element (subelements-specification) >
  - <!ATTLIST element (attributes) >

Element Specification in DTD

- Subelements can be specified as
  - names of elements, or
  - #PCDATA (parsed character data), i.e., character strings
  - EMPTY (no subelements) or ANY (anything can be a subelement)

- Example

```
<! ELEMENT department (dept_name  building, budget)>
<! ELEMENT dept_name (#PCDATA)>
<! ELEMENT building(#PCDATA)>
<! ELEMENT budget (#PCDATA)>
```

University DTD

```
<!DOCTYPE  university [
    <!ELEMENT university ( (department|course|instructor|teaches)+)>
    <!ELEMENT department ( dept name, building, budget)>
    <!ELEMENT course ( course id, title, dept name, credits)>
    <!ELEMENT instructor (IID, name, dept name, salary)>
    <!ELEMENT teaches (IID, course id)>
    <!ELEMENT dept name( #PCDATA )>
    <!ELEMENT building( #PCDATA )>
    <!ELEMENT budget( #PCDATA )>
    <!ELEMENT course id ( #PCDATA )>
    <!ELEMENT title ( #PCDATA )>
    <!ELEMENT credits( #PCDATA )>
    <!ELEMENT IID( #PCDATA )>
    <!ELEMENT name( #PCDATA )>
    <!ELEMENT salary( #PCDATA )>
]>
```

Element Specification in DTD

- Subelement specification may have regular expressions

```
<!ELEMENT university ( ( department | course | instructor | teaches )+)>
```

  - Notation:
    - "|" - or
    - "+" - 1 or more occurrences
    - "*" - 0 or more occurrences
    - "?" - 0 or 1 occurrence

Attribute Specification in DTD

- Attribute specification : for each attribute
  - Name
  - Type of attribute
    - CDATA
    - ID (identifier) or IDREF (ID reference) or IDREFS (multiple IDREFs)
  - Whether
    - mandatory (#REQUIRED)
      - has a default value (value),
    - or neither (#IMPLIED)
- Examples
  - <!ATTLIST course course_id CDATA #REQUIRED>, or
  - <!ATTLIST course
    course_id ID #REQUIRED
    dept_name IDREF #REQUIRED
    instructors IDREFS #IMPLIED >

IDs and IDREFs

- An element can have at most one attribute of type ID
- The ID attribute value of each element in an XML document must be distinct
  - Thus the ID attribute value is an object identifier
- An attribute of type IDREF must contain the ID value of another element in the same document
- An attribute of type IDREFS contains a set of (0 or more) ID values. Each ID value must contain the ID value of an element in the same document

University DTD with Attributes

```
<!DOCTYPE university-3 [
    <!ELEMENT university ( (department|course|instructor)+)>
    <!ELEMENT department ( building, budget )>
    <!ATTLIST department
        dept_name ID #REQUIRED >
    <!ELEMENT course (title, credits )>
    <!ATTLIST course
        course_id ID #REQUIRED
        dept_name IDREF #REQUIRED
        instructors IDREFS #IMPLIED >
    <!ELEMENT instructor ( name, salary )>
    <!ATTLIST instructor
        IID ID #REQUIRED
        dept_name IDREF #REQUIRED >
    . . .
]>
```

XML data with ID and IDREF attributes

```
<university-3>
    <department dept name="Comp. Sci.">
        <building> Taylor </building>
        <budget> 100000 </budget>
    </department>
    <department dept name="Biology">
        <building> Watson </building>
        <budget> 90000 </budget>
    </department>
    <course course id="CS-101" dept name="Comp. Sci" instructors="10101 83821">
        <title> Intro. to Computer Science </title>
        <credits> 4 </credits>
    </course>
    ....
    <instructor IID="10101" dept name="Comp. Sci.">
        <name> Srinivasan </name>
        <salary> 65000 </salary>
    </instructor>
    ....
</university-3>
```

Limitations of DTDs

- No typing of text elements and attributes
  - All values are strings, no integers, reals, etc.

- Difficult to specify unordered sets of subelements
  - (A | B)* allows specification of an unordered set
    - Cannot ensure the number of times each of A and B occurs

- IDs and IDREFs are untyped
  - The *instructors* attribute of an course may contain a reference to a department or another course, which is meaningless
    - *instructors* attribute should ideally be constrained to refer to instructor elements

XML Schema

- XML Schema is a more sophisticated schema language which addresses the drawbacks of DTDs.

- Supports typing
  - Built-in types
    - E.g., string, integer, decimal, and boolean
    - Also, constraints on min/max values

  - User-defined types
    - Simple types with added restrictions
    - Complex types constructed using constructors

  - Many more features, including
    - uniqueness and foreign key constraints, inheritance

XML Schema

- XML Schema is itself specified in XML syntax, unlike DTDs
  - More-standard representation, but verbose
- XML Scheme is integrated with namespaces; E.g., “xs:”
- BUT: XML Schema is significantly more complicated than DTDs.

XML Schema Version of Univ. DTD

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="university" type="universityType" />
<xs:element name="department">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="dept name" type="xs:string"/>
            <xs:element name="building" type="xs:string"/>
            <xs:element name="budget" type="xs:decimal"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
....
<xs:element name="instructor">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="IID" type="xs:string"/>
         <xs:element name="name" type="xs:string"/>
         <xs:element name="dept name" type="xs:string"/>
         <xs:element name="salary" type="xs:decimal"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

… Contd.

XML Schema Version of Univ. DTD (Cont.)

```
....
<xs:complexType name="UniversityType">
   <xs:sequence>
      <xs:element ref="department" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="course" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="instructor" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="teaches" minOccurs="0" maxOccurs="unbounded"/>
   </xs:sequence>
</xs:complexType>
</xs:schema>
```

- Choice of "xs:" was ours -- any other namespace prefix could be chosen
- Element "university" has type "universityType", which is defined separately
  - xs:complexType is used later to create the named complex type "UniversityType"

More features of XML Schema

- Attributes specified by xs:attribute tag:
  - <xs:attribute name = “dept_name”/>
  - adding the attribute use = “required” means value must be specified

More features of XML Schema

- Key constraint: “department names form a key for department elements under the root university element:

```
<xs:key name = “deptKey”>
        <xs:selector xpath = “/university/department”/>
        <xs:field xpath = “dept_name”/>
<\xs:key>
```

- Foreign key constraint from course to department:

```
<xs:keyref name = “courseDeptFKey” refer=“deptKey”>
        <xs:selector xpath = “/university/course”/>
        <xs:field xpath = “dept_name”/>
<\xs:keyref>
```

XML Schema vs DTD

- Constraining elements texts to specific types; e.g., numeric types or complex types
- User-defined types
- Uniqueness and foreign-key constraints.
- Integrating with namespaces to allow different parts of a document to conform to different schemas.
- Restricting types to create specialized types, for instance by specifying minimum and maximum values.
- Complex types to be extended by using a form of inheritance.

Outline

- Semi-Structured Data
  - **XML**
    - Structure of XML Data
    - XML Document Schema
    - **Querying and Transformation**
    - Storage of XML Data
  - JSON
  - RDF
- Object Orientation
- Textual Data
- Spatial Data

Querying and Transforming XML Data

- Main applications
  - Extract information from large bodies of XML data
  - Converting data between different representations in XML
- Main requirement: tools for
  - Translation of information from one XML schema to another
  - Querying on XML data
- Above two are closely related, and handled by the same tools

Querying and Transforming XML Data

- Standard XML querying/translation languages
  - XPath
    - Simple language consisting of path expressions
    - a building block for XQuery
  - XQuery
    - An XML query language with a rich set of features
    - It is modeled after SQL but is significantly different, since it has to deal with nested XML data.
    - XQuery also incorporates XPath expressions.
  - XSLT
    - Simple language designed for translation from XML to XML and XML to HTML
    - It is used primarily in document-formatting applications, rather in data management applications

Tree Model of XML Data

- Query and transformation languages are based on a **tree model** of XML data

- An XML document is modeled as a tree, with **nodes** corresponding to elements and attributes
  - Element nodes have child nodes, which can be attributes or subelements
  - Text in an element is modeled as a text node child of the element
  - Children of a node are ordered according to their order in the XML document
  - Element and attribute nodes (except for the root node) have a single parent, which is an element node
  - The root node has a single child, which is the root element of the document

XPath

- XPath is used to address (select) parts of documents using **path expressions**

- A path expression is a sequence of steps separated by "/"
  - Think of file names in a directory hierarchy

- Result of path expression: set of values that along with their containing elements/attributes match the specified path
  - E.g. /university-3/instructor/name returns

    <name>Srinivasan</name>
    <name>Brandt</name>
  - E.g. /university-3/instructor/name/text( ) returns

    Srinivasan
    Brandt

- The nodes returned by each step appear in the same order as their appearance in the document.

XPath

- The initial "/" denotes root of the document (above the top-level tag)
- Path expressions are evaluated left to right
  - Each step operates on the set of instances produced by the previous step
- Attribute values are accessed, using the "@" symbol.
  - E.g. /university-3/course/@course_id returns

    a set of all values of course_id attributes of course elements
- IDREF attributes are not dereferenced automatically (more on this later)

XPath

- Selection predicates may follow any step in a path, in [ ]
  - E.g. /university-3/course[credits] returns
    - 4 course elements containing a credits subelement
  - E.g. /university-3/course[credits >= 4] returns
    - 4 course elements with a credits value greater than or equal to 4,
  - E.g. /university-3/course[credits >= 4]/@course_id returns
    - 4 returns the course identifiers of courses with credits >= 4

Functions in XPath

- XPath provides several functions
- The function count() at the end of a path counts the number of elements in the set generated by the path
  - E.g. /university-2/instructor[count(./teaches/course)> 2] returns
    - instructors teaching more than 2 courses (on university-2 schema)
- Also function for testing position (1, 2, ..) of node w.r.t. siblings
- Boolean connectives and and or and function not() can be used in predicates

Functions in XPath

- The function id("k") returns the node (if any) with an attribute of type ID and value "k".

- IDREFs can be referenced using function id()
  - id() can also be applied to sets of references such as IDREFS and even to strings containing multiple references separated by blanks

  - E.g. /university-3/course/id(@dept_name) returns
    - all department elements referred to from the dept_name attribute of course elements.

  - E.g. /university-3/course/id(@instructors) returns
    - all instructor elements referred to from the instructor attribute of course elements.

More XPath Features

- Operator "|" used to implement union
  - E.g. /university-3/course[@dept name="Comp. Sci"] |
    /university-3/course[@dept name="Biology"]
    - Gives union of Comp. Sci. and Biology courses
    - However, "|" cannot be nested inside other operators.
    - The nodes in the union are returned in the order in which they appear in the document

More XPath Features

- doc(name) returns the root of a named document

- Thus, a path expression can be applied on a specified document, instead of being applied on the current default document.
  - E.g. doc("university.xml")/university/department returns
    4 all departments at the university

- The function collection(name) is similar to doc, but returns a collection of documents identified by name

More XPath Features

- "//" can be used to skip multiple levels of nodes
  - E.g. /university-3//name
    - finds any name element *anywhere* under the /university-3 element, regardless of the element in which it is contained.

- A step in the path can go to parents, siblings, ancestors and descendants of the nodes generated by the previous step, not just to the children
  - "//", described above, is a short form for specifying "all descendants"
  - ".." specifies the parent.

XQuery

- XQuery is a general purpose query language for XML data
- Currently being standardized by the World Wide Web Consortium (W3C)
- XQuery queries are modeled after SQL queries, but differ significantly from SQL .

XQuery

- XQuery is organized into five sections
  for … let … where … order by …return …

- XQuery syntax
  for ⇔ SQL **from**

  let allows temporary variables, and has no equivalent in SQL
  where ⇔ SQL **where**
  order by ⇔ SQL **order by**

  return ⇔ SQL **select** (allows the construction of results in XML)

- It is referred to as " FLWOR " (pronounced "flower") expression

- A FLWOR query need not contain all the clauses

FLWOR Syntax in XQuery

- Simple FLWOR expression in XQuery
  - find all courses with credits > 3, with each result enclosed in an <course_id> .. </course_id> tag

```
for  $x in /university-3/course
let   $courseId := $x/@course_id
where $x/credits > 3
return <course_id> { $courseId } </course id>
```

FLWOR Syntax in XQuery

```
for  $x in /university-3/course
let   $courseId := $x/@course_id
where $x/credits > 3
return <course_id> { $courseId } </course id>
```

- For clause uses XPath expressions, and variable in for clause ranges over values in the set returned by XPath
- The where clause, like the SQL where clause, performs additional tests on the joined tuples from the for clause
- Since the for clause uses XPath expressions, selections may occur within the XPath expression.

```
for $x in /university-3/course[credits > 3]
```

FLWOR Syntax in XQuery

```
for  $x in /university-3/course
let   $courseId := $x/@course_id
where $x/credits > 3
return <course_id> { $courseId } </course id>
```

- The let clause simply allows the results of XPath expressions to be assigned to variable names for simplicity of representation
- Let clause not really needed in this query
- In fact, since this query is simple, we can easily do away with the let clause, and the variable $courseId in the return clause could be replaced with $x/@course id.

```
for $x in /university-3/course[credits > 3]
return <course_id> { $x/@course_id } </course_id>
```

FLWOR Syntax in XQuery

```
for  $x in /university-3/course
let   $courseId := $x/@course_id
where $x/credits > 3
return <course_id> { $courseId } </course id>
```

- Items in the return clause are XML text unless enclosed in {}, in which case they are evaluated as expressions

- The query can be modified to return an element with tag course, with the course identifier as an attribute, by replacing the return clause with the following:

```
return <course course id="{$x/@course id}" />
```

FLWOR Syntax in XQuery

```
for  $x in /university-3/course
let   $courseId := $x/@course_id
where $x/credits > 3
return <course_id> { $courseId } </course id>
```

- XQuery also supports constructing elements using the element and attribute constructors.
  - E.g.,

```
return element course {
        attribute course_id {$x/@course_id},
        attribute dept_name {$x/dept_name},
        element title {$x/title},
        element credits {$x/credits}
}
```

returns course elements with course_id and dept_name as attributes and title and credits as subelements.

Other XQuery Abilities

- Join
- Nester queries
- Grouping and aggregation
- Sorting
- Interface to application programs (Java DOM API)

XSLT

- A **stylesheet** stores formatting options for a document, usually separately from document
  - E.g. an HTML style sheet may specify font colors and sizes for headings, etc.
- The **XML Stylesheet Language (XSL)** was originally designed for generating HTML from XML
- XSLT is a general-purpose transformation language
  - Can translate XML to XML, and XML to HTML
- XSLT transformations are expressed using rules called **templates**
  - Templates combine selection using XPath with construction of results

Outline

- Semi-Structured Data
  - **XML**
    - Structure of XML Data
    - XML Document Schema
    - Querying and Transformation
    - **Storage of XML Data**
  - JSON
  - RDF
- Object Orientation
- Textual Data
- Spatial Data

Storage of XML Data

- XML data can be stored in
  - Non-relational data stores
    - Flat files
      - Natural for storing XML
      - But has all problems discussed in Chapter 1
        - Data isolation
        - Atomicity
        - Concurrent access
        - Security
    - XML database
      - Database built specifically for storing XML data, supporting DOM model and declarative querying
      - Currently no commercial-grade systems

Storage of XML Data

- XML data can be stored in
  - Relational databases
    - Data must be translated into relational form
    - Advantage: mature database systems
    - Disadvantages: overhead of translating data and queries

Storage of XML in Relational Databases

- Approaches:
  - String Representation
  - Tree Representation
  - Map to relations

String Representation

- Store each top level element as a string field of a tuple in a relational database
  - Use a single relation to store all elements
    - Easy to use, but no knowledge about the schema of the stored elements
    - Not possible to query the data directly
  - Use a separate relation for each top-level element type
    - E.g. department, course, instructor, and teaches
      - Each with a string-valued attribute to store the element

- Indexing:
  - Store values of subelements/attributes to be indexed as extra fields of the relation, and build indices on these fields
    - E.g. dept_name, course id, or name

String Representation

- Benefits:
  - Can store any XML data even without DTD
  - As long as there are many top-level elements in a document, strings are small compared to full document
    - Allows fast access to individual elements.
- Drawback**:** Need to parse strings to access values inside the elements
  - Parsing is slow.

Tree Representation

- **Tree representation:** model XML data as tree and store using relations
  nodes(id, parent_id, type, label, value)

- Each element/attribute is given a unique identifier
- Type indicates element/attribute
- Label specifies the tag name of the element/name of attribute
- Value is the text value of the element/attribute
- Can add an extra attribute *position* to record ordering of children

Tree Representation

- Benefit:
  - Can store any XML data, even without DTD

- Drawbacks:
  - Data is broken up into too many pieces, increasing space overheads
  - Even simple queries require a large number of joins, which can be slow

Mapping XML Data to Relations

- Relation created for each element type whose schema is known:
  - An id attribute to store a unique id for each element
  - A relation attribute corresponding to each element attribute
  - A parent_id attribute to keep track of parent element
    - As in the tree representation
    - Position information (i^{th} child) can be store too
- All subelements that occur only once can become relation attributes
  - For text-valued subelements, store the text as attribute value
  - For complex subelements, can store the id of the subelement
- Subelements that can occur multiple times represented in a separate table
  - Similar to handling of multivalued attributes when converting ER diagrams to tables

Storing XML Data in Relational Systems

- Applying above ideas to department elements in university-1 schema, with nested course elements, we get

  department(*id*, *dept_name*, *building*, *budget*)

  course(*parent_id*, *course_id*, *dept_name*, *title*, *credits*)

Publishing and Shredding XML Data

- **Publishing**: process of converting relational data to an XML format for export to other applications.

- **Shredding**: process of converting back an XML document into a set of tuples to be inserted into one or more relations

- *XML-enabled* database systems support automated publishing and shredding
- Otherwise, application code can perform the publishing and shredding operations

Publishing and Shredding XML Data

- An XML-enabled database supports automatic publishing using map to relation mechanism

- Approaches:
  - Simple
    - Creates an XML element for every row of a table
    - Makes each column in that row a subelement of the XML element
  - Complex
    - Allows nested structures to be created
    - Extensions of SQL with nested queries in the select clause have been developed to allow easy creation of nested XML output

SQL/XML

- New standard SQL extension that allows creation of nested XML output
  - Each output tuple is mapped to an XML element *row*
  - Each relation attribute mapped to an XML element of the same name

SQL/XML

```
<university>
    <department>
        <row>
            <dept name> Comp. Sci. </dept name>
            <building> Taylor </building>
            <budget> 100000 </budget>
        </row>
        <row>
            <dept name> Biology </dept name>
            <building> Watson </building>
            <budget> 90000 </budget>
        </row>
    </department>
    <course>
        <row>
            <course id> CS-101 </course id>
            <title> Intro. to Computer Science </title>
            <dept name> Comp. Sci </dept name>
            <credits> 4 </credits>
        </row>
</university>
```

SQL Extensions

- **xmlelement** creates XML elements
- **xmlattributes** creates attributes

select xmlelement (**name** “course”,
    xmlattributes (*course id* **as** *course id*, *dept name* **as** *dept name*),
    xmlelement (**name** “title”, *title*),
    xmlelement (**name** “credits”, *credits*))
from *course*

SQL Extensions

- Xmlagg creates a forest of XML elements

- Creating an element for each department with a course, containing as subelements all the courses in that department.
- Since the query has a clause group by dept_name, the aggregate function is applied on all courses in each department, creating a sequence of course id elements.

```
select xmlelement (name "department", dept_name,
            xmlagg (xmlforest(course_id)
                        order by course_id))
from course
group by dept_name
```

Outline

- Semi-Structured Data
  - XML
  - **JSON**
  - RDF
- Object Orientation
- Textual Data
- Spatial Data

What is JSON

- **JSON** (JavaScript Object Notation) is a minimal, readable format for structuring data.
- It is used primarily to transmit data between a server and web application, as an alternative to XML.

- JSON is ubiquitous in data exchange today
  - Widely used for web services
  - Most modern applications are architected around on web services

- JSON is verbose
  - Compressed representations such as BSON (Binary JSON) used for efficient data storage

JSON vs. XML

- It is similar to XML in that it is:
  - "self describing" & human readable
  - hierarchical
  - simple & interoperable

- It differs from XML in that it is:
  - can be parsed with standard JavaScript notation
  - uses arrays
  - can be simpler & shorter to read & write
  - parsing is faster

JSON vs. XML

JSON

```
{"employees":[
    {"firstName":“Shannon", "lastName":“Kempe"},
    {"firstName":"Anita", "lastName":“Kress"},
    {"firstName":“Tony", "lastName":“Shaw"}
]}
```

XML

```
<employees>
    <employee>
        <firstName>Shannon</firstName>
        <lastName>Kempe</lastName>
    </employee>
    <employee>
        <firstName>Anita</firstName>
        <lastName>Kress</lastName>
    </employee>
    <employee>
        <firstName>Tony</firstName>
        <lastName>Shaw</lastName>
    </employee>
</employees>
```

JSON Metadata

- The JSON schema offers a richer set of metadata.

Data

```
{
"id": 127849,
"brand": "Super Cooler",
"price": 12.50,
"tags": ["camping", "sports"]
}
```

Example Product in the API

Context Needed (i.e. Metadata)

- Can the ID contain letters?
- What is a brand?
- Is a price required?
- Etc.

Metadata

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "Product",
  "description": "A retail product from Acme's online catalog",
  "type": "object",
  "properties": {
    "id": {
      "description": "The unique identifier for a product",
      "type": "integer"
    },
    "brand": {
      "description": "The brand name of the product as shown in the online catalogue",
      "type": "string"
    },
    "price": {
      "type": "number",
    },
    "tags": {
      "type": "array",
      "items": {
        "type": "string"
      },
      "minItems": 1,
}
  },
  "required": ["id", "brand", "price"]
}
```

JSON Schema

Integrating JSON with Document Databases

- JSON is often used with document databases, such as MongoDB, which usesJSONdocuments in order to store records
- Document databases are popular ways to store unstructured information in a flexible way (e.g. multimedia, social media posts, etc.)
  - Each Collection can contain numerous Documents which could all contain different fields.

{type: “Artifact”,
medium: “Ceramic”
country: “China”,
}

{type: “Book”,
title: “Ancient China”
country: “China”,
}

JSON

- Textual representation widely used for data exchange
- Example of JSON data

```
{
        "ID": "22222",
        "name": {
                "firstname: "Albert",
                "lastname: "Einstein"
        },
        "deptname": "Physics",
        "children": [
                {"firstname": "Hans", "lastname": "Einstein" },
                {"firstname": "Eduard", "lastname": "Einstein" }
        ]
}
```

- Types: integer, real, string, and
  - *Objects: are* key-value maps, i.e. sets of (attribute name, value) pairs
  - Arrays are also key-value maps (from offset to value)

JSON Structure

- JSON uses map data structure rather than tree data structure

- Syntax follows JS syntax

- data is in name/value pairs
- data is separated by commas
- curly braces { } hold objects
- square brackets [ ] hold arrays

JSON

- SQL extensions for
  - JSON types for storing JSON data
  - Extracting data from JSON objects using path expressions
    - E.g. *V*-> *ID*, or *v.ID*
  - Generating JSON from relational data
    - E.g. json.build_object('ID', 12345, 'name', 'Einstein')
  - Creation of JSON collections using aggregation
    - E.g. json_agg aggregate function in PostgreSQL
  - Syntax varies greatly across databases

Outline

- Semi-Structured Data
  - XML
  - JSON
  - **RDF**
    - **Linked Open Data**
    - RDF Structure
    - SPARQL
- Object Orientation
- Textual Data
- Spatial Data

Web of Linked Data

- The Web is evolving from a “Web of linked documents” into a “Web of linked data”
  - The Web started as a collection of documents published online – accessible at a Web location identified by a URL.
  - These documents often contain data about real-world resources which is mainly human-readable and cannot be understood by machines.
  - The Web of Data is about enabling the access to this data, by making it available in machine-readable formats and connecting it using Uniform Resource Identifiers (URIs), thus enabling people and machines to collect the data, and put it together to do all kinds of things with it (permitted by the licence).

Knowledge Graph

Linked Open Data

Linked Open Data

2007

2008

2009

2011

2014

2017

Defining linked data

- The **four design principles** of Linked Data:
  - Use Uniform Resource Identifiers (URIs) as names for things.
  - Use HTTP URIs so that people can look up those names.
  - When someone looks up a URI, provide useful information, using the standards (RDF, SPARQL).
  - Include links to other URIs so that they can discover more things.

Design Principles

1. Use Uniform Resource Identifiers (URIs) as names for things.
2. Use HTTP URIs so that people can look up those names.

E.g. for an organisation: **UNICEF** in EuroVoc.

4 http://eurovoc.europa.eu/1022

Design Principles

3. When someone looks up a URI, provide useful information, using the standards (RDF, SPARQL).
4. Include links to other URIs so that people/machines can discover more things.

```
<rdf:Description rdf:about="http://eurovoc.europa.eu/223339">
    <rdf:type rdf:resource="http://eurovoc.europa.eu/schema#PreferredTerm"/>
    <eu:termReleasedWithVersion>n/a</eu:termReleasedWithVersion>
    <xl:literalForm xml:lang="en">Unicef</xl:literalForm>
    <skos:inScheme rdf:resource="http://eurovoc.europa.eu/100285"/>
    <xl:altLabel rdf:resource="http://eurovoc.europa.eu/223340"/>
    <xl:altLabel rdf:resource="http://eurovoc.europa.eu/223341"/>
</rdf:Description>

<rdf:Description rdf:about="http://eurovoc.europa.eu/1022">
    <rdf:type rdf:resource="http://eurovoc.europa.eu/schema#ThesaurusConcept"/>
    <xl:prefLabel rdf:resource="http://eurovoc.europa.eu/223339"/>
    <skos:broader rdf:resource="http://eurovoc.europa.eu/4365"/>
    <skos:related rdf:resource="http://eurovoc.europa.eu/758"/>
    <skos:exactMatch rdf:resource="http://ec.europa.eu/eclas/euc11/000005663"/>
</rdf:Description>
```

Linked Data vs. Open Data

> "*Open data is data that can be freely used, reused and redistributed by anyone – subject only, at most, to the requirement to attribute and share-alike.*"
> *- OpenDefinition.org*

Open data

Data can be published and be publicly available under an open licence without linking to other data sources.

$\neq$

Linked data

Data can be linked to URIs from other data sources, using open standards such as RDF without being publicly available under an open licence.

5 star-schema of Linked Open Data

★	Make your stuff available on the Web (whatever format) under an open license.
★★	Make it available as structured data (e.g., Excel instead of image scan of a table).
★★★	Use non-proprietary formats (e.g., CSV instead of Excel).
★★★★	Use URIs to denote things, so that people can point at your stuff.
★★★★★	Link your data to other data to provide context.

5 star-schema of Linked Open Data

★ Make your stuff available on the Web under an open licence

ESMA Report on Trends, Risks and Vulnerabilities No. 1, 2013 10

Securities markets

Equity markets

Equities: EU prices close to its 5Y-MA T.01

Price dispersion: Increase due to outliers T.02

Volatility: Significant improvement since July 2012 T.03

Equity markets developed positively in 2012. After a volatile first semester, equity indices increased as from early August 2012, along with an overall improvement of financial markets in the EU, following the announcement of Outright Monetary Transactions (OMT) by the ECB and further progress on the establishment of a Banking Union. The improvement can also be seen in liquidity and volatility indicators, which performed better than their five year averages.

Global Equities: After bottoming out in July, the EU equity index increased by 10% as from then, edging closer to its five-year average. Compared to Japan, EU indices performed slightly better as from July 2012. However, EU indices underperformed compared to the US, due to macroeconomic conditions. US equity markets were around 12% higher than their early 2011 level, while European and Japanese indices were still 3% lower.

Dispersion: As from August, most of the EU national equity indices experienced increases. More precisely, from then on the top 75% followed an upward trend. However, one European country suffered a very significant decline from January 2011, as indicated by the lowest value of the bottom 25%. The aggregate effect of this sharp decline was mitigated by the fact that the country in question is relatively small in terms of market capitalisation. However, it does indicate European equity markets' increasing differentiation, as price movements in national equity indices for the bottom quartile are decoupled from the generally positive trends in other national indices.

Volatility: Expected volatility, measured by the implied volatility of options on the Eurostoxx 50, decreased from July 2012 and at year-end was around 8.5 percentage points lower than its five-year average. Having been

5 star-schema of Linked Open Data

★ ★ Make it available as structured data

	A	B	C
1	Value	Definition	ortDescriptio
2	AL	Albania	
3	AT	Austria	
4	BA	Bosnia and Herzegovina	
5	BE	Belgium	
6	BG	Bulgaria	
7	CH	Switzerland	
8	CY	Cyprus	
9	CZ	Czech Republic	
10	DE	Germany	
11	DK	Denmark	
12	EE	Estonia	
13	ES	Spain	
14	FI	Finland	
15	FR	France	
16	GB	United Kingdom	
17	GR	Greece	
18	HR	Croatia	
19	HU	Hungary	
20	IE	Ireland	
21	IS	Iceland	
22	IT	Italy	
23	LI	Liechtenstein	
24	LT	Lithuania	
25	LU	Luxembourg	
26	LV	Latvia	
27	ME	Montenegro	
28	MK	Macedonia, the Former Yugoslav Republic of	
29	MT	Malta	
30	NL	Netherlands	
31	NO	Norway	
32	PL	Poland	
33	PT	Portugal	
34	RO	Romania	
35	RS	Serbia	
36	SE	Sweden	
37	SI	Slovenia	

CASnumber | **CountryCode** | Determinand_HazSubs

5 star-schema of Linked Open Data

★ ★ ★ Use non-proprietary formats

- Proprietary: Excel, Word, PDF...
- Non-proprietary: XML, CSV, RDF, JSON, ODF...


```xml
<?xml version="1.0"?>
- <iati-activities generated-datetime="2014-01-09T13:23:20" version="1.0">
  - <iati-activity xml:lang="en" last-updated-datetime="2014-01-09T13:13:09" default-currency="EUR">
      <reporting-org type="15" ref="EC-ELARG">European Commission - Enlargement</reporting-org>
      <iati-identifier>EU-2010/22028/1</iati-identifier>
      <other-identifier owner-name="European Commission - Enlargement" owner-ref="EC-ELARG">2010/22028/1</other-identifier>
      <title>Support for Improvement in Governance and Management (SIGMA) in the Western Balkans and Turkey </title>
      <description>2010 Multi-Beneficiary Programme under the IPA Transition Assistance and Institution Building Component - part implemented by DG ELARG (other parts encoded under 2009/022-
        029 (DG ENTR) and 2009/022-030 (DG TAXUD)). Support for Improvement in Governance and Management (SIGMA) in the Western Balkans and Turkey </description>
      <activity-status code="2">Implementation</activity-status>
      <activity-date type="start-planned" iso-date="2010-12-20">2010-12-20</activity-date>
      <participating-org role="Funding" type="15" ref="EC-ELARG">European Commission - Enlargement</participating-org>
      <participating-org role="Extending" type="15" ref="EU-1">Commission of the European Communities</participating-org>
      <participating-org role="Implementing" ref="50000">OTHER</participating-org>
      <recipient-region code="89">Europe, regional</recipient-region>
      <sector code="15110" vocabulary="DAC">Public sector policy and administrative management</sector>
      <policy-marker code="01" vocabulary="DAC" significance="0">Gender Equality</policy-marker>
      <policy-marker code="02" vocabulary="DAC" significance="0">Aid to Environment</policy-marker>
      <policy-marker code="03" vocabulary="DAC" significance="2">Participatory Development/Good</policy-marker>
      <policy-marker code="04" vocabulary="DAC" significance="0">Trade Development</policy-marker>
      <policy-marker code="05" vocabulary="DAC" significance="0">Aid Targeting the Objectives of the Convention on Biological Diversity</policy-marker>
      <policy-marker code="06" vocabulary="DAC" significance="0">Aid Targeting the Objectives of the Framework Convention on Climate Change - Mitigation</policy-marker>
      <policy-marker code="07" vocabulary="DAC" significance="0">Aid Targeting the Objectives of the Framework Convention on Climate Change - Adaptation</policy-marker>
      <policy-marker code="08" vocabulary="DAC" significance="0">Aid Targeting the Objectives of the Convention to Combat Desertification</policy-marker>
      <collaboration-type code="1">Bilateral</collaboration-type>
      <default-flow-type code="10">ODA (Official development Assistance</default-flow-type>
      <default-finance-type code="110">Grant</default-finance-type>
      <default-aid-type code="C01">Project-type interventions</default-aid-type>
    - <transaction ref="EU-2010/22028/1/0">
        <transaction-type code="C">Commitment</transaction-type>
        <value>10000000</value>
        <transaction-date iso-date="2010-05-06">2010-05-06</transaction-date>
      </transaction>
    - <transaction ref="EU-2010/22028/13/0">
        <transaction-type code="C">Commitment</transaction-type>
        <value>1500000</value>
        <transaction-date iso-date="2010-05-06">2010-05-06</transaction-date>
      </transaction>
    - <transaction ref="EU-2010/22028/15/0">
```

5 star-schema of Linked Open Data

★ ★ ★ ★ Use URIs to denote things

```
<rdf:Description rdf:about="http://ec.europa.eu/semantic_webgate/dataset/additives/resource/additive-300">
  <additives:id>300</additives:id>
  <additives:additiveName>Oxidized polyethylene wax</additives:additiveName>
  <additives:isGroup>No</additives:isGroup>
  <additives:eNumber>E 914</additives:eNumber>
  <rdf:type rdf:resource="http://ec.europa.eu/semantic_webgate/dataset/additives/resource/Additive"/>
</rdf:Description>
<rdf:Description rdf:about="http://ec.europa.eu/semantic_webgate/dataset/additives/resource/additive-301">
  <additives:eNumber>E 920</additives:eNumber>
  <additives:additiveName>L-cysteine</additives:additiveName>
  <additives:isGroup>No</additives:isGroup>
  <rdf:type rdf:resource="http://ec.europa.eu/semantic_webgate/dataset/additives/resource/Additive"/>
  <additives:id>301</additives:id>
</rdf:Description>
```

5 star-schema of Linked Open Data

★ ★ ★ ★ ★ Link your data to other data to provide context

```
<skos:Concept rdf:about="http://publications.europa.eu/resource/authority/corporate-body/UNICEF"
              at:deprecated="false">
  <skos:inScheme rdf:resource="http://publications.europa.eu/resource/authority/corporate-body"/>
  <skos:broader rdf:resource="http://publications.europa.eu/resource/authority/corporate-body/UNO"/>
  <at:authority-code>UNICEF</at:authority-code>
  <at:op-code>UNICEF</at:op-code>
  <atold:op-code>UNICEF</atold:op-code>
  <dc:identifier>UNICEF</dc:identifier>
  <at:start.use>1951-01-01</at:start.use>
  <skos:prefLabel xml:lang="bg">Унисеф - Детски фонд на ООН</skos:prefLabel>
  <skos:prefLabel xml:lang="cs">UNICEF - Dětský fond Organizace spojených národů</skos:prefLabel>
  <skos:prefLabel xml:lang="da">UNICEF - De Forenede Nationers Børnefond</skos:prefLabel>
  <skos:prefLabel xml:lang="de">Unicef - Kinderhilfswerk der Vereinten Nationen</skos:prefLabel>
  <skos:prefLabel xml:lang="el">Unicef - Ταμείο των Ηνωμένων Εθνών για τα Παιδιά</skos:prefLabel>
  <skos:prefLabel xml:lang="en">Unicef - United Nations Children's Fund</skos:prefLabel>
```

```
<skos:Concept rdf:about="http://publications.europa.eu/resource/authority/corporate-body/UNO"
              at:deprecated="false">
  <skos:narrower rdf:resource="http://publications.europa.eu/resource/authority/corporate-body/UNRWA"/>
  <skos:narrower rdf:resource="http://publications.europa.eu/resource/authority/corporate-body/UNICEF"/>
  <skos:narrower rdf:resource="http://publications.europa.eu/resource/authority/corporate-body/ECOSOC"/>
  <skos:narrower rdf:resource="http://publications.europa.eu/resource/authority/corporate-body/UNESCO"/>
  <at:authority-code>UNO</at:authority-code>
  <at:op-code>UNO</at:op-code>
  <atold:op-code>UNO</atold:op-code>
  <dc:identifier>UNO</dc:identifier>
  <at:start.use>1951-01-01</at:start.use>
  <skos:prefLabel xml:lang="bg">Организация на обединените нации</skos:prefLabel>
  <skos:prefLabel xml:lang="cs">Organizace spojených národů</skos:prefLabel>
  <skos:prefLabel xml:lang="da">De Forenede Nationers Organisation</skos:prefLabel>
  <skos:prefLabel xml:lang="de">Vereinte Nationen</skos:prefLabel>
  <skos:prefLabel xml:lang="el">Οργανισμός Ηνωμένων Εθνών</skos:prefLabel>
  <skos:prefLabel xml:lang="en">United Nations Organisation</skos:prefLabel>
```

Linked data foundations

- URIs for naming things
- RDF for describing data
- SPARQL for querying linked data

Uniform Resource Identifier (URI)

- *"A Uniform Resource Identifier (URI) is a compact sequence of characters that identifies an abstract or physical resource."*
  - ISA's 10 Rules for Persistent URIs

- A country, e.g. Belgium
  - http://publications.europa.eu/resource/authority/country/BEL

- An organisation, e.g. the Publications Office
  - http://publications.europa.eu/resource/authority/corporate-body/PUBL

- A dataset, e.g. Countries Named Authority List
  - http://publications.europa.eu/resource/authority/country/

Outline

- Semi-Structured Data
  - XML
  - JSON
  - **RDF**
    - Linked Open Data
    - **RDF Structure**
    - SPARQL
- Object Orientation
- Textual Data
- Spatial Data

Knowledge Graph

RDF

- The Resource Description Framework (RDF) is a syntax for representing data and resources in the Web

- RDF breaks every piece of information down in triples:
  - Subject – a resource, which may be identified with a URI.
  - Predicate – a URI-identified reused specification of the relationship.
  - Object – a resource or literal to which the subject is related.

http://example.org/place/Brussels is the capital of "Belgium".

OR

http://example.org/place/Brussels is the capital of http://example.org/place/Belgium.

Subject Predicate Object

RDF Syntax (RDF/XML)

```
<rdf:RDF

   xmlns:dcat="http://www.w3.org/TR/vocab-dcat/"
   xmlns:dct="http://purl.org/dc/terms/"

<dcat:Dataset rdf:about="http://publications.europa.eu/resource/authority/file-type/">
   <dct:title> "File types Name Authority List"< /dct:title>
   <dct:publisher rdf:resource="http://open-data.europa.eu/en/data/publisher/publ"/>
</dcat:Dataset>

<dct:Agent rdf:about="http://open-data.europa.eu/en/data/publisher/publ"/>
   <dct:title>"Publications Office"</dct:title>
</dct:Publisher>

</rdf:RDF>
```

Graph

	Subject
	Predicate
	Object

RDF/XML is currently the only syntax that is standardised by W3C.

Knowledge Representation

- Representation of human knowledge is a long-standing goal of AI
  - Various representations of facts and inference rules proposed over time
- **RDF: Resource Description Format**
  - Simplified representation for facts, represented as triples (*subject, predicate, object*)
    - E.g., (NBA-2019, *winner*, Raptors)
      (Washington-DC, *capital-of*, USA)
      (Washington-DC, *population*, 6,200,000)
  - Models objects that have attributes, and relationships with other objects
    - Like the ER model, but with a flexible schema
    - (*ID, attribute-name, value*)
    - (*ID1, relationship-name, ID2*)
  - Has a natural graph representation

Graph View of RDF Data

- **Knowledge graph**

Triple View of RDF Data

10101	instance-of	instructor .
10101	name	"Srinivasan" .
10101	salary	"6500" .
00128	instance-of	student .
00128	name	"Zhang" .
00128	tot_cred	"102" .
comp_sci	instance-of	department .
comp_sci	dept_name	"Comp. Sci." .
biology	instance-of	department .
CS-101	instance-of	course .
CS-101	title	"Intro. to Computer Science" .
CS-101	course_dept	comp_sci .
sec1	instance-of	section .
sec1	sec_course	CS-101 .
sec1	sec_id	"1" .
sec1	semester	"Fall" .
sec1	year	"2017" .
sec1	classroom	packard-101 .
sec1	time_slot_id	"H" .
10101	inst_dept	comp_sci .
00128	stud_dept	comp_sci .
00128	takes	sec1 .
10101	teaches	sec1 .

RDF Representation

- RDF triples represent binary relationships
- How to represent n-ary relationships?
  - Approach 1 (from Section 6.9.4): Create artificial entity, and link to each of the n entities
    - E.g., (Barack Obama, *president-of*, USA, 2008-2016) can be represented as
      (*e1*, *person*, Barack Obama), (*e1*, *country*, USA),
      (*e1*, *president-from*, 2008) (*e1*, *president-till*, 2016)
  - Approach 2: use **quads** instead of triples, with context entity
    - E.g., (Barack Obama, *president-of*, USA, *c1*)
      (*c1*, *president-from*, 2008) (*c1*, *president-till*, 2016)

RDF Representation

- RDF widely used as knowledge base representation
  - DBPedia, Yago, Freebase, WikiData, ..

- **Linked open data** project aims to connect different knowledge graphs to allow queries to span databases

Outline

- Semi-Structured Data
  - XML
  - JSON
  - **RDF**
    - Linked Open Data
    - RDF Structure
    - **SPARQL**
- Object Orientation
- Textual Data
- Spatial Data

SPARQL

- SPARQL is a standardised language for querying RDF data.

Querying RDF: SPARQL

- Triple patterns
  - ?cid *title* "Intro. to Computer Science"
    ?sid *course* ?cid
- SPARQL queries
  - **select** ?name
    where {
    ?cid *title* "Intro. to Computer Science" .
    ?sid *course* ?cid .
    ?id *takes* ?sid .
    ?id *name* ?name .
    }
- Also supports
  - Aggregation, Optional joins (similar to outerjoins), Subqueries, etc.
  - Transitive closure on paths

Outline

- Semi-Structured Data
- **Object Orientation**
  - Complex Data Types and Object Orientation
  - Structured Data Types and Inheritance in SQL
  - Array and Multiset Types in SQL
  - Comparison of Object-Oriented and Object-Relational Databases
- Textual Data
- Spatial Data

Object Orientation

- **Object-relational data model** provides richer type system
  - with complex data types and object orientation

- Applications are often written in object-oriented programming languages
  - Type system does not match relational type system
  - Switching between imperative language and SQL is troublesome

Object Orientation

- Approaches for integrating object-orientation with databases
  - Build an **object-relational database**, adding object-oriented features to a relational database
  - Automatically convert data between programming language model and relational model; data conversion specified by **object-relational mapping**
  - Build an **object-oriented database** that natively supports object-oriented data and direct access from programming language

Outline

- Semi-Structured Data
- **Object Orientation**
  - **Complex Data Types and Object Orientation**
  - Structured Data Types and Inheritance in SQL
  - Array and Multiset Types in SQL
  - Comparison of Object-Oriented and Object-Relational Databases
- Textual Data
- Spatial Data

Outline

- Semi-Structured Data
- **Object Orientation**
  - **Complex Data Types and Object Orientation**
  - Structured Data Types and Inheritance in SQL
  - Array and Multiset Types in SQL
  - Comparison of Object-Oriented and Object-Relational Databases
- Textual Data
- Spatial Data

Object-Relational Data Models

- Extend the relational data model by including object orientation and constructs to deal with added data types.
- Allow attributes of tuples to have complex types, including non-atomic values such as nested relations.
- Preserve relational foundations, in particular the declarative access to data, while extending modeling power.
- Upward compatibility with existing relational languages.

Complex Data Types

- Motivation:
  - Permit non-atomic domains
  - Example of non-atomic domain: set of integers,or set of tuples
  - Allows more intuitive modeling for applications with complex data

- Intuitive definition:
  - allow relations whenever we allow atomic (scalar) values — relations within relations
  - Retains mathematical foundation of relational model
  - Violates first normal form.

Example of a Nested Relation

- Example: library information system
- Each book has
  - title,
  - a list (array) of authors,
  - Publisher, with subfields *name* and *branch*, and
  - a set of keywords
- Non-1NF relation *books*

title	*author_array*	*publisher*	*keyword_set*
		(*name*, *branch*)	
Compilers	[Smith, Jones]	(McGraw-Hill, NewYork)	{parsing, analysis}
Networks	[Jones, Frick]	(Oxford, London)	{Internet, Web}

4NF Decomposition of Nested Relation

- Suppose for simplicity that title uniquely identifies a book
  - In real world ISBN is a unique identifier
- Decompose *books* into 4NF using the schemas:
  - (*title, author, position*)
  - (*title, keyword*)
  - (*title, pub-name, pub-branch*)
- 4NF design requires users to include joins in their queries.

title	*author*	*position*
Compilers	Smith	1
Compilers	Jones	2
Networks	Jones	1
Networks	Frick	2

authors

title	*keyword*
Compilers	parsing
Compilers	analysis
Networks	Internet
Networks	Web

keywords

title	*pub_name*	*pub_branch*
Compilers	McGraw-Hill	New York
Networks	Oxford	London

books4

Complex Types and SQL

- Extensions introduced in SQL:1999 to support complex types:
  - Collection and large object types
    - Nested relations are an example of collection types
  - Structured types
    - Nested record structures like composite attributes
  - Inheritance
  - Object orientation
    - Including object identifiers and references

- Not fully implemented in any database system currently
  - But some features are present in each of the major commercial database systems
    - Read the manual of your database system to see what it supports

Outline

- Semi-Structured Data
- **Object Orientation**
  - Complex Data Types and Object Orientation
  - **Structured Data Types and Inheritance in SQL**
  - Array and Multiset Types in SQL
  - Comparison of Object-Oriented and Object-Relational Databases
- Textual Data
- Spatial Data

Structured Types and Inheritance in SQL

- **Structured types** (a.k.a. **user-defined types**) can be declared and used in SQL

```
create type Name as
    (firstname      varchar(20),
     lastname       varchar(20))
     final

create type Address as
    (street         varchar(20),
     city           varchar(20),
     zipcode        varchar(20))
     not final
```

- Note: **final** and **not final** indicate whether subtypes can be created

Structured Types and Inheritance in SQL

- Structured types can be used to create tables with composite attributes

```
create table person (
  name        Name,
  address     Address,
  dateOfBirth date)
```

- Dot notation used to reference components: *name.firstname*

Structured Types (cont.)

- **User-defined row types**

  create type *PersonType* **as** (
  name Name,
  address Address,
  dateOfBirth **date**)
  not final

- Can then create a table whose rows are a user-defined type
  create table *person* **of** *PersonType*

- Alternative using **unnamed row types**.

  create table *person_r*(
  name **row(**first*name* **varchar**(20),
  lastname **varchar**(20)),
  address **row(***street* **varchar**(20),
  city **varchar**(20),
  zipcode **varchar**(20)),
  dateOfBirth **date**)

Methods

- Can add a method declaration with a structured type.

  method *ageOnDate* (*onDate* **date**)

  returns interval year

- Method body is given separately.

  create instance method *ageOnDate* (*onDate* **date**)

  returns interval year

  for *PersonType*

  begin

  return *onDate* - **self**.*dateOfBirth*;

  end

- We can now find the age of each person:

  select *name.lastname, ageOnDate* (**current_date**)

  from *person*

Constructor Functions

- **Constructor functions** are used to create values of structured types

- E.g.
  create function *Name*(*firstname* **varchar**(20), *lastname* **varchar**(20))
  returns *Name*
  begin
  set self.*firstname* = *firstname;*
  set self.*lastname* = *lastname;*
  end

- To create a value of type *Name,* we use
  new *Name*(‘John’, ‘Smith’)

- Normally used in insert statements
  insert into *Person* **values**
  (**new** *Name*(‘John’, ‘Smith),
  new *Address*(’20 Main St’, ‘New York’, ‘11001’),
  date ‘1960-8-22’);

Type Inheritance

- Suppose that we have the following type definition for people:

```
create type Person
    (name varchar(20),
    address varchar(20))
```

- Using inheritance to define the student and teacher types

```
create type Student
  under Person
  (degree       varchar(20),
   department   varchar(20))

create type Teacher
  under Person
  (salary       integer,
   department   varchar(20))
```

- Subtypes can redefine methods by using **overriding method** in place of **method** in the method declaration

Multiple Type Inheritance

- SQL:1999 and SQL:2003 do not support multiple inheritance

- If our type system supports multiple inheritance, we can define a type for teaching assistant as follows:

  create type *Teaching Assistant*
  under *Student, Teacher*

- To avoid a conflict between the two occurrences of *department* we can rename them

  create type *Teaching Assistant*
  under
  Student **with** (*department* **as** *student_dept*),
  Teacher **with** (*department* **as** *teacher_dept*)

- Each value must have a **most-specific type**

Table Inheritance

- Tables created from subtypes can further be specified as **subtables**

- E.g.

  create table *people* **of** *Person;*

  create table *students* **of** *Student* **under** *people;*

  create table *teachers* **of** *Teacher* **under** *people;*

- Tuples added to a subtable are automatically visible to queries on the supertable
  - E.g. query on *people* also sees *students* and *teachers*.
  - Similarly updates/deletes on *people* also result in updates/deletes on subtables
  - To override this behaviour, use "**only** *people*" in query

Outline

- Semi-Structured Data
- **Object Orientation**
  - Complex Data Types and Object Orientation
  - Structured Data Types and Inheritance in SQL
  - **Array and Multiset Types in SQL**
  - Comparison of Object-Oriented and Object-Relational Databases
- Textual Data
- Spatial Data

Array and Multiset Types in SQL

- SQL supports two collection types:
  - Arrays
    - array types were added in SQL:1999
  - Multisets
    - multiset types were added in SQL:2003 .

- Multiset is an unordered collection, where an element may occur multiple times.

- Multisets are like sets, except that a set allows each element to occur at most once.

- Unlike elements in a multiset, the elements of an array are ordered
  - So we can distinguish the first item from the second item, and so on.

Array and Multiset Types in SQL

- Example of array and multiset declaration:

```
create type Publisher as
    (name          varchar(20),
     branch        varchar(20));

create type Book as
    (title         varchar(20),
     author_array  varchar(20) array [10],
     pub_date      date,
     publisher     Publisher,
     keyword-set   varchar(20) multiset);

create table books of Book;
```

Creation of Collection Values

- Array construction

  array ['Silberschatz',` Korth',` Sudarshan']

- Multisets

  multiset ['computer', 'database', 'SQL']

- To create a tuple of the type defined by the books relation:

  ('Compilers', **array**[` Smith',` Jones'],
  new *Publisher* (` McGraw-Hill',` New York'),
  multiset [` parsing',` analysis'])

- To insert the preceding tuple into the relation books

  insert into *books*
  values
  ('Compilers', **array**[` Smith',` Jones'],
  new *Publisher* (` McGraw-Hill',` New York'),
  multiset [` parsing',` analysis']);

Querying Collection-Valued Attributes

- To find all books that have the word “database” as a keyword,

  select *title*
  from *books*
  where ‘database’ **in** (**unnest**(*keyword-set*))

- We can access individual elements of an array by using indices
  - E.g.: If we know that a particular book has three authors, we could write:

  select *author_array*[1], *author_array*[2], *author_array*[3]
  from *books*
  where *title* = ` Database System Concepts’

- To get a relation containing pairs of the form “title, author_name” for each book and each author of the book

  select *B.title, A.author*
  from *books* **as** *B*,
  unnest (*B.author_array*) **as** *A* (*author*)

Unnesting

- The transformation of a nested relation into a form with fewer (or no) relation-valued attributes is called **unnesting**.
- E.g.

  select *title*, *A* **as** *author*, *publisher.name* **as** *pub_name*,
  publisher.branch **as** *pub_branch*, *K.keyword*
  from *books* **as** *B*,
  unnest(*B.author_array*) **as** *A* (*author*),
  unnest (*B.keyword_set*) **as** *K* (*keyword*)
- Result relation *flat_books*

title	*author*	*pub_name*	*pub_branch*	*keyword*
Compilers	Smith	McGraw-Hill	New York	parsing
Compilers	Jones	McGraw-Hill	New York	parsing
Compilers	Smith	McGraw-Hill	New York	analysis
Compilers	Jones	McGraw-Hill	New York	analysis
Networks	Jones	Oxford	London	Internet
Networks	Frick	Oxford	London	Internet
Networks	Jones	Oxford	London	Web
Networks	Frick	Oxford	London	Web

Nesting

- **Nesting** is the opposite of unnesting, creating a collection-valued attribute

- Nesting can be done in a manner similar to aggregation, but using the function **colect**() in place of an aggregation operation, to create a multiset

- To nest the *flat_books* relation on the attribute *keyword*:

select *title*, *author*, *Publisher* (*pub_name, pub_branch*) **as** *publisher*,
    collect (*keyword*) **as** *keyword_set*
from *flat_books*
groupby *title, author, publisher*

title	*author*	*publisher*	*keyword_set*
		(*pub_name,pub_branch*)	
Compilers	Smith	(McGraw-Hill, New York)	{parsing, analysis}
Compilers	Jones	(McGraw-Hill, New York)	{parsing, analysis}
Networks	Jones	(Oxford, London)	{Internet, Web}
Networks	Frick	(Oxford, London)	{Internet, Web}

Nesting

- To nest on both authors and keywords:

```
select title,
        collect (author ) as author_set,
        Publisher (pub_name, pub_branch) as publisher,
        collect  (keyword ) as keyword_set
from    flat_books
group by title, publisher
```

Nesting

- To nest on both authors and keywords using set():

```
select title,
        set (author ) as author_set,
        Publisher (pub_name, pub_branch) as publisher,
        set (keyword ) as keyword_set
from    flat_books
group by title, publisher
```

Nesting

- To nest on both authors and keywords using subqueries:

```
select  title,
        ( select  author
          from  flat-books  as  M
          where  M.title=O.title)  as  author-set,
        Publisher (pub-name, pub-branch)  as  publisher,
        ( select  keyword
          from  flat-books  as N
          where  N.title = O.title )  as  keyword-set
from  flat-books  as  O
```

Nesting

- To nest on both authors and keywords using subqueries:

```
select title,
        array( select author
          from authors as A
          where A.title = B.title
          order by A.position) as author_array,
        Publisher(pub_name, pub_branch) as publisher,
        multiset( select keyword
          from keywords as K
          where K.title = B.title) as keyword_set,
from books4 as B;
```

Outline

- Semi-Structured Data
- **Object Orientation**
  - Complex Data Types and Object Orientation
  - Structured Data Types and Inheritance in SQL
  - Array and Multiset Types in SQL
  - **Comparison of Object-Oriented and Object-Relational Databases**
- Textual Data
- Spatial Data

Object-Relational Mapping

- **Object-Relational Mapping (ORM)** systems built on top of traditional relational databases
- Implementer provides a mapping from objects to relations
- Objects can be retrieved from database
  - System uses mapping to fetch relevant data from relations and construct objects
  - Updated objects are stored back in database by generating corresponding update/insert/delete statements

Object-Relational Mapping

- Object-relational mapping (ORM) systems allow
  - Specification of mapping between programming language objects and database tuples
  - Automatic creation of database tuples upon creation of objects
  - Automatic update/delete of database tuples when objects are update/deleted
  - Interface to retrieve objects satisfying specified conditions
    - Tuples in database are queried, and object created from the tuples
- Details in Section 9.6.2
  - Hibernate ORM for Java
  - Django ORM for Python

Comparison of O-R Databases

- Object-relational databases are object-oriented data-bases built on top of the relation model
- Object-relational mapping systems build an object layer on top of a traditional relational database
- Another type:
  - Object-oriented databases
- Each of these approaches targets a different market.

Comparison of O-R Databases

- Object-relational systems aim at making data modeling and querying easier by using complex data types.
  - Typical applications include storage and querying of complex data, including multimedia data.
  - SQL , however, imposes a significant performance penalty for certain kinds of applications that run primarily in main memory, and that perform a large number of accesses to the database.

Comparison of O-R Databases

- Object-relational mapping systems allow programmers to build applications using an object model, while using a traditional database system to store the data.
  - They combine the robustness of widely used relational database systems, with the power of object models for writing applications.
  - However, they suffer from overheads of data conversion between the object model and the relational model used to store data.

Comparison of O-R Databases

- **Relational systems**
  - simple data types, powerful query languages, high protection.
- **Object-relational systems**
  - complex data types, powerful query languages, high protection.
- **Object-relational mapping systems**
  - complex data types integrated with programming language, but built as a layer on top of a relational database system

Outline

- Semi-Structured Data
- Object Orientation
- **Textual Data**
  - **TF-IDF**
  - Link Analysis
  - Evaluation
- Spatial Data

Textual Data

- **Information retrieval**: querying of unstructured data
  - Simple model of keyword queries: given query keywords, retrieve documents containing all the keywords
  - Today, keyword queries return many types of information as answers
    - E.g., a query “cricket” typically returns scores from ongoing or recent cricket matches, rather than just top-ranked documents related to cricket

- More advanced models rank relevance of documents

- Relevance ranking is essential since there are usually many documents matching keywords

Document Selection vs Ranking

Document Selection Problem

- The classifier is unlikely accurate
  - “Over-constrained” query no relevant documents to return
  - “Under-constrained” query over delivery
  - Hard to find the right position between these two extremes

- Even if it is accurate, all relevant documents are not equally relevant (relevance is a matter of degree!)
  - Prioritization is needed

- Thus, ranking is generally preferred => Vector Space Model

Ranked Retrieval

- Providing a relevance ranking of the documents with respect to a query
  - Assign a score to each query-document pair, say in [0, 1].
  - This score measures how well document and query “match”.
  - Sort documents according to scores

Vector Space Model

- Represent a doc/query by a term vector
  - Term: basic concept, e.g., word
  - Each term defines one dimension
  - N terms define an N-dimensional space
  - Query vector: $q=(x_1, ..., x_N)$, x_i is query term weight
  - Doc vector: $d=(y_1, ..., y_N)$, y_j is doc term weight

- relevance(q,d) ≈ similarity(q,d) = f(q,d)

Vector Space Model

- Main items in calculating scores
  - The importance of the term in query and document:
    - how many times does a query term occur in q and d?

    => Term Frequency (TF)
  - The general importance of the term in the collection:
    - is it a frequent or rare term? how often do we see the query term in the entire collection?

    => Document Frequency (DF)
  - Normalization of the scores based on the length of the document:
    - How long is d?

    => Document length (|d|)

Ranking using TF-IDF

- Term: keyword occurring in a document/query
- **Term Frequency:** $TF(d, t)$, the relevance of a term t to a document d
  - One definition: $TF(d, t) = log(1 + n(d,t)/n(d))$
    where
    - $n(d,t)$ = number of occurrences of term t in document d
    - $n(d)$ = number of terms in document d

	Anthony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth	...
ANTHONY	157	73	0	0	0	1	
BRUTUS	4	157	0	2	0	0	
CAESAR	232	227	0	2	1	0	
CALPURNIA	0	10	0	0	0	0	
CLEOPATRA	57	0	0	0	0	0	
MERCY	2	0	3	8	5	8	
WORSER	2	0	1	1	1	5	
...							

Ranking using TF-IDF

- **Inverse document frequency**: IDF(t)
  - One definition: $IDF(t) = 1/n(t)$
- **Relevance** of a document d to a set of terms Q
  - *One definition:* $r(d, Q) = \sum_{t \in Q} TF(d, t) * IDF(t)$
  - Other definitions
    - take **proximity** of words into account
    - **Stop words** are often ignored

term	df_t	idf_t
calpurnia	1	6
animal	100	4
sunday	1000	3
fly	10,000	2
under	100,000	1
the	1,000,000	0

Document Normalization

- Long doc has a better chance to match any query
- Penalize a long documents with a doc length normalizer

Cosine Similarity

$$\cos(\vec{q},\vec{d}) = \text{SIM}(\vec{q},\vec{d}) = \frac{\vec{q}\cdot\vec{d}}{|\vec{q}||\vec{d}|} = \frac{\sum_{i=1}^{|V|} q_i d_i}{\sqrt{\sum_{i=1}^{|V|} q_i^2}\sqrt{\sum_{i=1}^{|V|} d_i^2}}$$

- cosine similarity of $\vec{q}$ and $\vec{d}$
  - q_i is the *tf-idf* weight of term *i* in the query.
  - d_i is the *tf-idf* weight of term *i* in the document.
  - $|\vec{q}|$ and $|\vec{d}|$ are the lengths of $\vec{q}$ and $\vec{d}$
- Also includes doc length normalization

Outline

- Semi-Structured Data
- Object Orientation
- **Textual Data**
  - TF-IDF
  - **Link Analysis**
  - Evaluation
- Spatial Data

Link Analysis

- Makes use of links and anchor text in web pages.
  - Stored and indexed separately
  - <a href = http://www.aut.ac.ir> Amirkabir University of Technology</a>
- Link analysis identifies popularity and community information
  - e.g., PageRank
- Anchor text can significantly enhance the representation of pages pointed to by links
- Significant impact on web search
  - Less importance in other applications

Ranking Using Hyperlinks

- Hyperlinks provide very important clues to importance
- Google introduced PageRank, a measure of popularity/importance based on hyperlinks to pages
  - Pages hyperlinked from many pages should have higher PageRank
  - Pages hyperlinked from pages with higher PageRank should have higher PageRank
  - Formalized by **random walk** model

Ranking Using Hyperlinks

- Let $T[i, j]$ be the probability that a random walker who is on page i will click on the link to page j
  - Assuming all links are equal, $T[i, j] = 1/Ni$

- Then PageRank[j] for each page j can be defined as
  - $P[j] = \delta/N + (1 - \delta) * \sum_{i=1}^{N} (T[i, j] * P[i])$
    - N = total number of pages,
    - δ a constant usually set to 0.15

Ranking Using Hyperlinks

- Definition of PageRank is circular, but can be solved as a set of linear equations
  - Simple iterative technique works well
  - Initialize all P[i] = 1/N
  - In each iteration use equation $P[j] = \delta/N + (1 - \delta) * \sum_{i=1}^{N} (T[i, j] * P[i])$ to update P
  - Stop iteration when changes are small, or some limit (say 30 iterations) is reached.

Other Aspects

- Other measures of relevance are also important. For example:
  - Keywords in anchor text
  - Number of times who ask a query click on a link if it is returned as an answer

Outline

- Semi-Structured Data
- Object Orientation
- **Textual Data**
  - TF-IDF
  - Link Analysis
  - **Evaluation**
- Spatial Data

Precision and Recall

- Required data
  - The set of relevant documents (but we may not find all)
  - The set of retrieved documents (but not all of them are relevant)

- Precision (P) is the fraction of retrieved documents that are relevant

Precision = #(relevant items retrieved) / #(retrieved items) = P(relevant|retrieved)

- Recall (R) is the fraction of relevant documents that are retrieved

Recall = #(relevant items retrieved) / #(relevant items) = P(retrieved|relevant)

Precision and Recall

Precision = $\frac{\text{True positives}}{\text{Retrieved documents}}$

Recall = $\frac{\text{True positives}}{\text{Relevant documents}}$

Precision/Recall Tradeoff

- Find algorithm that maximizes precision.
  - Or minimizes classification errors (false positives)
  - Return nothing!
- Find algorithm that maximizes recall.
  - Return everything!
- Solution:
  - Considering both measures at the same time by F-Measure

F-Measure

- General formula

$$F = \frac{1}{\alpha\frac{1}{P} + (1-\alpha)\frac{1}{R}} = \frac{(\beta^2+1)PR}{\beta^2 P + R} \quad \text{where} \quad \beta^2 = \frac{1-\alpha}{\alpha}$$

α ∈ [0, 1] and thus β2 ∈ [0, ∞]

- Harmonic mean of recall and precision (β2 =1)

$$F = \frac{1}{\frac{1}{2}(\frac{1}{R}+\frac{1}{P})} = \frac{2RP}{R+P}$$

$z = \frac{1}{2}(x + y)$

$z = 2(x \cdot y) / (x + y)$

Retrieval Effectiveness

- Measures of effectiveness
  - **Precision**: what percentage of returned results are actually relevant
  - **Recall**: what percentage of relevant results were returned
  - At some number of answers, e.g. precision@10, recall@10

Retrieval Effectiveness

- Measures of effectiveness
  - **Precision**: what percentage of returned results are actually relevant
  - **Recall**: what percentage of relevant results were returned
  - At some number of answers, e.g. precision@10, recall@10

Outline

- Semi-Structured Data
- Object Orientation
- Textual Data
- **Spatial Data**

Spatial Data

- Spatial databases store information related to spatial locations, and support efficient storage, indexing and querying of spatial data.
  - **Geographic data** -- road maps, land-usage maps, topographic elevation maps, political maps showing boundaries, land-ownership maps, and so on.
    - **Geographic information systems** are special-purpose databases tailored for storing geographic data.
    - Round-earth coordinate system may be used
      - (Latitude, longitude, elevation)
  - **Geometric data:** design information about how objects are constructed .
    - For example, designs of buildings, aircraft, layouts of integrated-circuits.
    - 2 or 3 dimensional Euclidean space with (X, Y, Z) coordinates

Represented of Geometric Information

Various geometric constructs can be represented in a database in a normalized fashio

- A **line segment** can be represented by the coordinates of its endpoints.
- A **polyline** or **linestring** consists of a connected sequence of line segments and can be represented by a list containing the coordinates of the endpoints of the segments, in sequence.
  - Approximate a curve by partitioning it into a sequence of segments
    - Useful for two-dimensional features such as roads.
    - Some systems also support *circular arcs* as primitives, allowing curves to be represented as sequences of arc
- **Polygons** is represented by a list of vertices in order.
  - The list of vertices specifies the boundary of a polygonal region.
  - Can also be represented as a set of triangles (**triangulation**)

Representation of Geometric Constructs

line segment

{(x1,y1), (x2,y2)}

triangle

{(x1,y1), (x2,y2), (x3,y3)}

polygon

{(x1,y1), (x2,y2), (x3,y3), (x4,y4), (x5,y5)}

polygon

{(x1,y1), (x2,y2), (x3,y3), ID1}
{(x1,y1), (x3,y3), (x4,y4), ID1}
{(x1,y1), (x4,y4), (x5,y5), ID1}

object **representation**

Representation of Geometric Information (Cont.)

- Representation of points and line segment in 3-D similar to 2-D, except that points have an extra z component
- Represent arbitrary polyhedra by dividing them into tetrahedrons, like triangulating polygons.
- Alternative: List their faces, each of which is a polygon, along with an indication of which side of the face is inside the polyhedron.
- Geometry and geography data types supported by many databases
  - E.g. SQL Server and PostGIS
  - point, linestring, curve, polygons
  - Collections: multipoint, multilinestring, multicurve, multipolygon
  - LINESTRING(1 1, 2 3, 4 4)
  - POLYGON((1 1, 2 3, 4 4, 1 1))
  - Type conversions:
    - *ST GeometryFromText*() and *ST GeographyFromText*()
  - *Operations:*
    - *ST Union*(), *ST Intersection*(), …

Design Databases

- Represent design components as objects (generally geometric objects); the connections between the objects indicate how the design is structured.
- Simple two-dimensional objects: points, lines, triangles, rectangles, polygons.
- Complex two-dimensional objects: formed from simple objects via union, intersection, and difference operations.
- Complex three-dimensional objects: formed from simpler objects such as spheres, cylinders, and cuboids, by union, intersection, and difference operations.
- Wireframe models represent three-dimensional surfaces as a set of simpler objects.

Representation of Geometric Constructs

- Design databases also store non-spatial information about objects (e.g., construction material, color, etc.)
- Spatial integrity constraints are important.
  - E.g., pipes should not intersect, wires should not be too close to each other, etc.

(a) Difference of cylinders (b) Union of cylinders

Geographic Data

- **Raster data** consist of bit maps or pixel maps, in two or more dimensions.
  - Example 2-D raster image: satellite image of cloud cover, where each pixel stores the cloud visibility in a particular area.
  - Additional dimensions might include the temperature at different altitudes at different regions, or measurements taken at different points in time.
- Design databases generally do not store raster data.

Geographic Data (Cont.)

- **Vector data** are constructed from basic geometric objects: points, line segments, triangles, and other polygons in two dimensions, and cylinders, spheres, cuboids, and other polyhedrons in three dimensions.
- Vector format often used to represent map data.
  - Roads can be considered as two-dimensional and represented by lines and curves.
  - Some features, such as rivers, may be represented either as complex curves or as complex polygons, depending on whether their width is relevant.
  - Features such as regions and lakes can be depicted as polygons.

Spatial Queries

- **Region queries** deal with spatial regions. e.g., ask for objects that lie partially or fully inside a specified region
  - E.g., PostGIS *ST_Contains*(), *ST_Overlaps*(), …
- **Nearness queries** request objects that lie near a specified location.
- **Nearest neighbor queries**, given a point or an object, find the nearest object that satisfies given conditions.
- **Spatial graph queries** request information based on spatial graphs
  - E.g., shortest path between two points via a road network
- **Spatial join** of two spatial relations with the location playing the role of join attribute.
- Queries that compute intersections or **unions** of regions

Questions?