DSA5104
Principles of Data Management and Retrieval

Lecture 6: Semi-structured Data Management

Decomposition of a Relation Scheme

- How to normalize a relation?
  - *Decompose* into multiple **normalized** relations

- Suppose R contains attributes $A_1 \ldots A_n$.

- A *decomposition* of R consists of replacing R by two or more relations such that:
  - Each new relation scheme contains a subset of the attributes of R, and
  - Every attribute of R appears as an attribute of at least one of the new relations.

Lossy Decomposition (example)

A	B	C
1	2	3
4	5	6
7	2	8

$A \rightarrow B;\ B \rightarrow C$

⇒

A	B
1	2
4	5
7	2

B	C
2	3
5	6
2	8

A	B
1	2
4	5
7	2

$\bowtie$

B	C
2	3
5	6
2	8

=

A	B	C
1	2	3
4	5	6
7	2	8
1	2	8
7	2	3

Lossy Decomposition (example)

A	B	C
1	2	3
4	5	6
7	2	8

$A \rightarrow B$; $\cancel{B \rightarrow C}$

⇒

A	B
1	2
4	5
7	2

B	C
2	3
5	6
2	8

A	B
1	2
4	5
7	2

$\bowtie$

B	C
2	3
5	6
2	8

=

A	B	C
1	2	3
4	5	6
7	2	8
1	2	8
7	2	3

Lossy Decomposition (example)

A	B	C
1	2	3
4	5	6
7	2	8

$A \to B; C \to B$

A	B
1	2
4	5
7	2

B	C
2	3
5	6
2	8

A	B
1	2
4	5
7	2

$\bowtie$

B	C
2	3
5	6
2	8

$=$

A	B	C
1	2	3
4	5	6
7	2	8
1	2	8
7	2	3

Recap

- Decomposition
  - Lossy decomposition
  - Lossless decomposition (Definition / Test)
  - Dependency Preserving Decomposition
    - Projection of set of FDs F
  - Decomposition into BCNF (algorithm)
    - A lossless decomposition that is guaranteed to terminate
    - Decomposition using FDs
  - BCNF and Dependency Preservation
    - 3NF (3rd Normal Form)

Semi-Structured 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)
  - , **Arrays**
    - Widely used for scientific and monitoring applications

Features of Semi-Structured Data Models (Cont.)

- **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

Basic Normal Forms

- 1st Normal Form – all attributes atomic
  - I.e. relational model
  - Violated by many common data models
    - Including XML, JSON, various OO models
  - Some of these "non-first-normal form" (NFNF) quite useful in various settings
    - Especially in update-never settings – e.g., data tranfer
    - If you never "unnest", then who cares!
      - Basically relational collection of structured objects

- 1st ⊃ 2nd (of historical interest)
  ⊃ 3rd
  ⊃ Boyce-Codd …

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

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" }
        ]
}
```

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

JSON (Cont.)

- JSON is ubiquitous in data exchange today
  - Widely used for web services,
  - *Can represent complex structure and allow flexible structuring*
- SQL extensions for
  - JSON types for storing JSON data
  - Extracting data from JSON objects using path expressions
    - E.g. v-> *ID*, or *v.ID* to access the value of attribute ‘ID’ of v
  - Generating JSON objets from relational data
    - E.g. json_build_object(‘ID’, 12345, ‘name’, ‘Einstein’)
  - Creation of JSON objects from a collection of rows using aggregation
    - E.g. json_agg aggregate function in PostgreSQL
  - *Syntax varies greatly across databases*
- JSON is verbose
  - Compressed representations such as BSON (Binary JSON) used for efficient data storage

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 Framework**
  - RDF - A data representation standard (data model) based on the entity-relationship model
  - Models objects that have attributes, and relationships with other objects by a set of *triples*:

    (*ID, attribute-name, value*)

    (*ID1, relationship-name, ID2*)

    - *ID, ID1, and ID2* are identifiers of entities (resources)
    - Like the ER model, but with a flexible schema (e.g., add new attributes to objects or create new relationships)
  - A triple has the structure (*subject, predicate, object*)
    - E.g., (NBA-2019, *winner*, Raptors)
      (Washington-DC, *capital-of*, USA)
      (Washington-DC, *population*, 6,200,000)
  - *Has a natural graph representation*

Triple View of RDF Data

- Triple - (*subject, predicate, object*)
- Models objects that have attributes, and relationships with other objects
  - (*ID, attribute-name, value*)
  - (*ID1, relationship-name, ID2*)

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 of part of the University database.

Triple View of RDF Data

- Triple - (*subject, predicate, object*)
- Models objects that have attributes, and relationships with other objects
  - (*ID, attribute-name, value*)
  - (*ID1, relationship-name, ID2*)

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 of part of the University database.

Triple View of RDF Data

- Triple - (*subject, predicate, object*)
- Models objects that have attributes, and relationships with other objects
  - (*ID, attribute-name, value*)
  - (*ID1, relationship-name, ID2*)

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 .

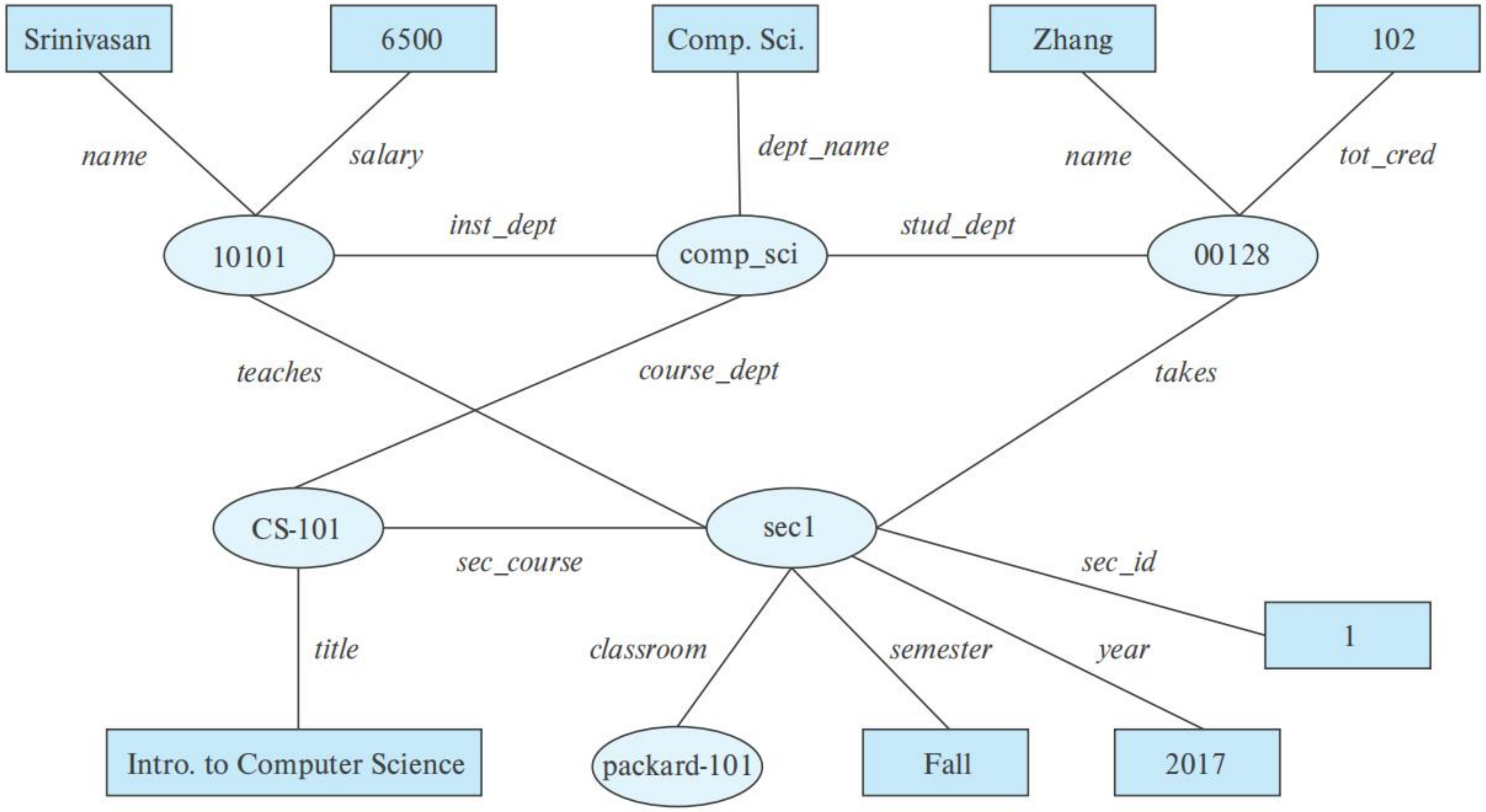
RDF representation of part of the University database.

Graph View of RDF Data

- **Knowledge graph** for part of the University database

- **Objects** - ovals
- **Attribute values** - rectangles
- **Relationships** - edges with associated labels identifying the relationship

- Note: We have omitted the instance-of relationships for brevity.

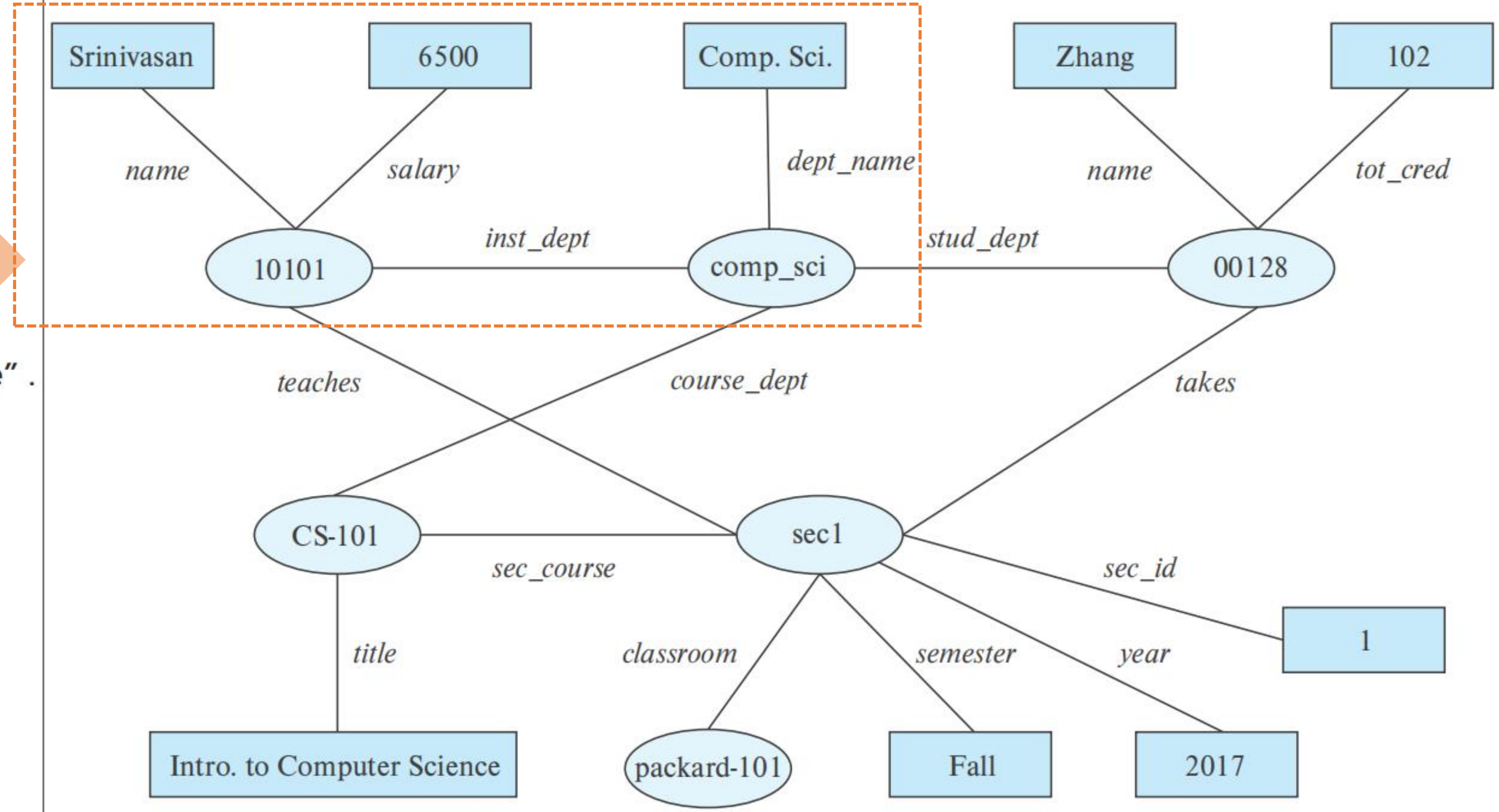


Knowledge Graph

Graph 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 of part of the University database.



Knowledge Graph

Querying RDF - SPARQL

- Triple patterns
  - **?cid** ***title*** **"Intro. to Computer Science"**
  - This triple pattern would match all triples whose predicate is “title” and object is “Intro. to Computer Science”.
  - Note: RDF triple - (*subject, predicate, object*)
  - Note: ?cid is a variable that can match any value

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 of part of the University database.

Querying RDF - SPARQL

- Triple patterns
  - ?cid *title* "Intro. to Computer Science"
  - This triple pattern would match all triples whose predicate is “title” and object is “Intro. to Computer Science”.
  - Note: RDF triple - (*subject, predicate, object*)
  - Note: ?cid is a variable that can match any value

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 of part of the University database.

Querying RDF - SPARQL

- Triple patterns
  - ➢ **?cid *title* "Intro. to Computer Science"**
    - This triple pattern would match all triples whose predicate is “title” and object is “Intro. to Computer Science”.
    - Note: RDF triple - (*subject, predicate, object*)
    - Note: ?cid is a variable that can match any value
  - ➢ **?cid *title* "Intro. to Computer Science"**
    ?sid *course* ?cid
    - The first triple pattern matches the triple (CS-101, title, "Intro. to Computer Science")
    - The second one matches (sec1, course, CS-101).

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 of part of the University database.

Querying RDF - SPARQL

- Triple patterns
  - **?cid** ***title*** **"Intro. to Computer Science"**
  - This triple pattern would match all triples whose predicate is “title” and object is “Intro. to Computer Science”.
  - Note: RDF triple - (*subject, predicate, object*)
  - Note: ?cid is a variable that can match any value
  - **?cid** ***title*** **"Intro. to Computer Science"**
    ?sid ***course*** **?cid**
  - The first triple pattern matches the triple (CS-101, title, "Intro. to Computer Science")
  - The second one matches (sec1, course, CS-101).
  - Note: The shared variable ?cid enforces a *join* condition between the two triple patterns.

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 of part of the University database.

Querying RDF - SPARQL (Cont.)

- A complete SPARQL query
  - **select** ?name
    where {
        ?cid *title* "Intro. to Computer Science" .
        ?sid *course* ?cid .
        ?id *takes* ?sid .
        ?id *name* ?name .
    }
  - *What is the result of this query?*

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 of part of the University database.

Querying RDF - SPARQL (Cont.)

- A complete SPARQL query
  - **select** ?name
    where {
    ?cid *title* "Intro. to Computer Science" .
    ?sid *course* ?cid .
    ?id *takes* ?sid .
    ?id *name* ?name .
    }
  - This query retrieves names of all students who have taken a section whose course is titled “Intro. to Computer Science”.
  - Also supports
    - Aggregation, Optional joins (similar to outerjoins), Subqueries, etc.

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 .

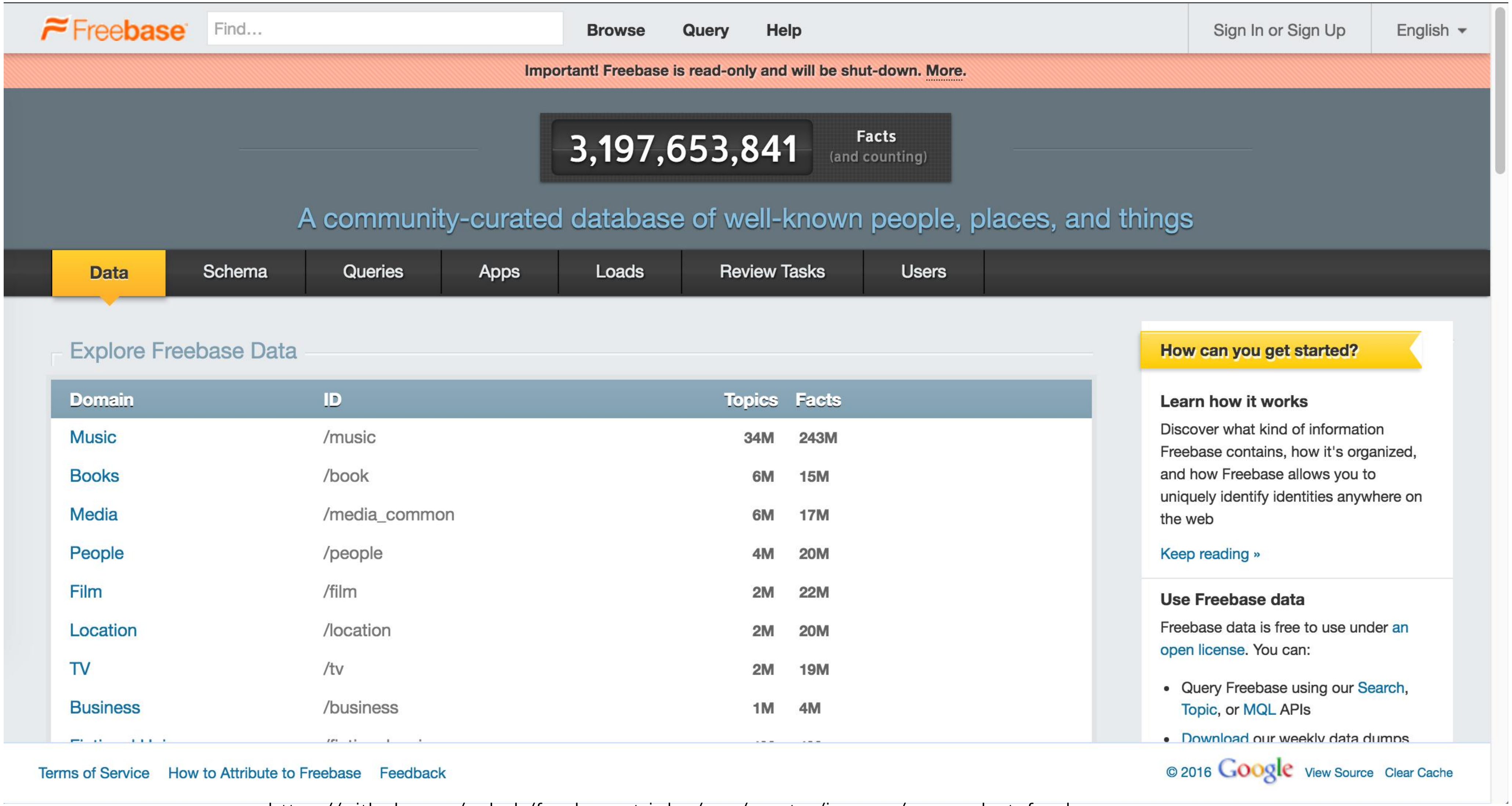
RDF representation of part of the University database.

RDF Representation for N-ary Relationships

- RDF triples represent binary relationships (unlike ER model)
- How to represent n-ary relationships?
  - Approach 1: 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)

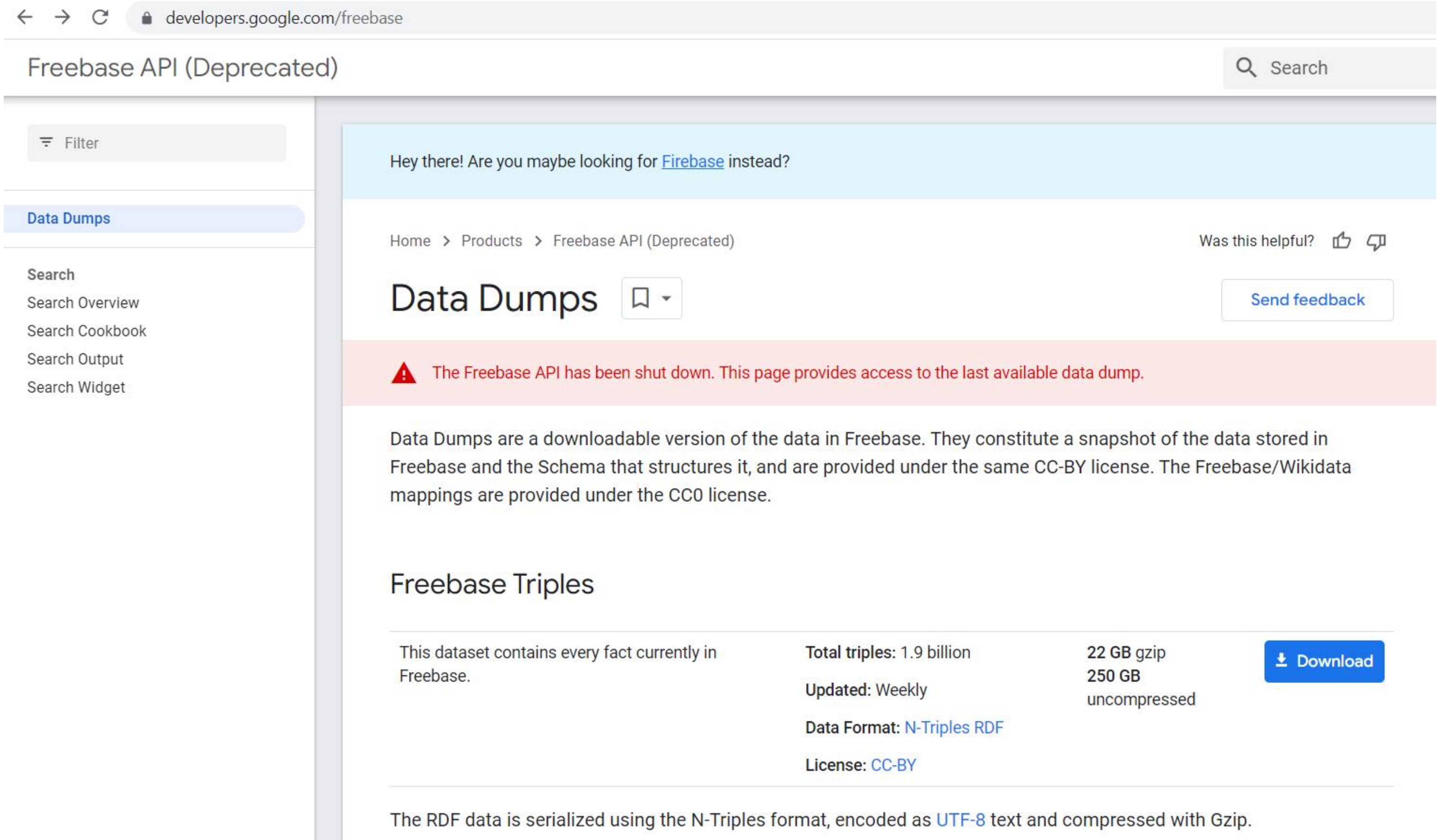
- RDF widely used as knowledge base representation
  - Knowledge Bases: DBPedia, Yago, **Freebase**, WikiData
- **Linked open data** project aims to connect different knowledge graphs to allow queries to span databases

Freebase (2007-2016)



https://github.com/nchah/freebase-triples/raw/master/images/screenshot-freebase-com.png

Freebase



developers.google.com/freebase

Freebase API (Deprecated)

Search

Filter

Data Dumps

Search

Search Overview

Search Cookbook

Search Output

Search Widget

Hey there! Are you maybe looking for Firebase instead?

Home > Products > Freebase API (Deprecated)

Was this helpful?

Data Dumps

Send feedback

The Freebase API has been shut down. This page provides access to the last available data dump.

Data Dumps are a downloadable version of the data in Freebase. They constitute a snapshot of the data stored in Freebase and the Schema that structures it, and are provided under the same CC-BY license. The Freebase/Wikidata mappings are provided under the CC0 license.

Freebase Triples

This dataset contains every fact currently in Freebase.	**Total triples:** 1.9 billion **Updated:** Weekly **Data Format:** N-Triples RDF **License:** CC-BY	22 GB gzip 250 GB uncompressed	Download

The RDF data is serialized using the N-Triples format, encoded as UTF-8 text and compressed with Gzip.

Freebase Schema

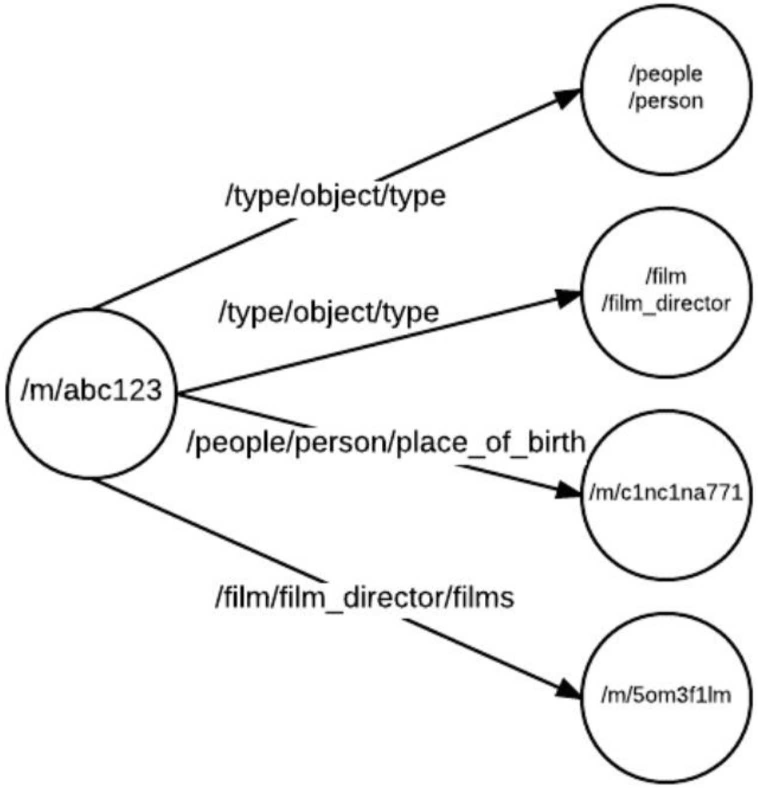
- A distinct entity or object is called a *topic, e.g.,* the notable film director Steven Spielberg
- Each topic is associated with a unique machineId identifier (often abbreviated as mid) that is in the format "/m/ + {alphanumeric}", e.g., /m/abc123.
- The topic is said to be a member of a class by saying it "has certain type(s)".
  - The famous film director will have the Person type (written in a human-readable format as /people/person) and the Film Director type (/film/film_director).
- Under each type, further granular data is represented through *properties*
  - *E.g.,* Place of Birth *(*/people/person/place_of _birth*)* and the films he directed *(*/film/film_director/films*).*
- A property can link a topic to a value or to other topics

Freebase Schema (Cont.)

- RDF triple - (*subject, predicate, object*)
  - A triple links a *subject* through a *predicate* to a *object*.
  - E.g., Johnny Appleseed (subject) likes (predicate) apples (object).

```
# /type/object/type indicates an entity's types
/m/abc123, /type/object/type, /people/person
/m/abc123, /type/object/type, /film/film_director

# These triples express facts about /m/abc123
/m/abc123, /people/person/place_of_birth, /m/c1nc1na771
/m/abc123, /film/film_director/films, /m/5om3f1lm
```



"*freebase-triples: A Methodology for Processing the Freebase Data Dumps*", https://arxiv.org/pdf/1712.08707.pdf

XML

XML - Extensible Markup Language

- **Markup** refers to anything in a document that is not intended to be part of the printed output.
  - E.g., a note like “set this word in large size, bold font” does not end up printed in the newspaper

联合早报

2022年9月9日 星期五

zaobaosg

政府机构将采用统一建筑楼面面积定义

李蕙心　报道
hueyshin@sph.com.sg

市区重建局、新加坡土地管理局、建设局和新加坡民防部队将采用统一的建筑楼面面积测量方式和定义，房地产业主日后向不同机构提出申请时将更加便捷。

根据市区重建局网站上的一份通告，从明年6月开始，市建局、土管局、建设局和民防部队将采用相同的建筑楼面面积定义。

通告说，这四家法定机构和制服团体现有的建筑楼面面积定义各有不同。例如，市建局用总楼面面积（gross floor area，简称GFA）来衡量建筑物密度；土管局则以分层面积（strata area）作为业主拥有的面积。土管局用统计总建筑楼面面积（statistical GFA）来计算建筑楼面的总面积；民防部队则使用可通达楼面面积（accessible floor area）来制定防火规定。

此外，它们的面积测量方式也不一样。市建局在计算总楼面面积时会纳入建筑外墙的完整厚度，但不包括开敞空间（void area），例如两层楼空间（double-volume space）或阁楼（loft）。土管局则会把外墙的一半厚度和一些开敞空间算在楼面面积内。

这让土木工程从业员为了按照不同机构的要求准备申请文件时，耗费不少时间和精力。

这也为房地产业主在进行加建与改建（addition and alteration）工程时增添不少麻烦，也可能面对即使分层面积没有增加，但是由于计入开敞空间而导致总楼面面积有所增加的情况。

资料来源 / 市区重建局　　早报制图 / 张进培

定义统一后，测量楼面面积时将包括建筑外墙一半的厚度。所有分层面积，不论是有遮盖还是无遮盖的面积，都会包括在总楼面面积内，开敞空间则不算在内。

另外，建设局和民防部队将采用相同的统计总建筑楼面面积定义。所有在明年6月1日或之后提交给市建局的发展项目申请，以及在今年9月1日或之后推出的政府售地计划地段和政府工业售地计划地段，将根据新定义处理。

ERA产业研究咨询部主管麦俊荣接受《联合早报》访问时说，一些房地产有两层楼空间或者阁楼，这些都算是分层面积，也是可出售楼面（saleable area）。

有些单位的每平方英尺售价把开敞空间计算在内，整体单位尺价会较低，若不把这些空间算在价格内，尺价就会更高。

新定义施行后，他预测一些发展商未来可能会减少这类空间设计，或只用于尺价本来就高的高档私宅。

“一些高档私宅的客厅天花板可能高达6.5公尺，适合吊大型水晶灯，显得豪华气派。发展商能以稀有双层楼空间设计为宣传重点，高价出售这些单位。“

橙易产业研究与咨询部总监孙燕清也有类似看法。

她受访时说：“有开敞空间的住房不多，主要是顶楼私宅单位或阁楼，所以对市场不会有太大的影响。这对买家有一定的帮助，他们会更清楚花钱买下了哪些空间。”

她还说，外墙的测量方式改变后的面积差别不会太明显，因为墙壁并不特别厚。

XML - Extensible Markup Language

- **Markup** refers to anything in a document that is not intended to be part of the printed output.
  - E.g., a note like “set this word in large size, bold font” does not end up printed in the newspaper

- 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.

- Markup languages evolved
  - From specifying instructions for how to print parts of the document to specifying the **function** of the content.
  - E.g., ‘text’ - section headings

联合早报

2022年9月9日 星期五

zaobaosg

政府机构将采用统一建筑楼面面积定义

李蕙心　报道
hueyshin@sph.com.sg

市区重建局、新加坡土地管理局、建设局和新加坡民防部队将采用统一的建筑楼面面积测量方式和定义，房地产业主日后向不同机构提出申请时将更加便捷。

根据市区重建局网站上的一份通告，从明年6月开始，市建局、土管局、建设局和民防部队将采用相同的建筑楼面面积定义。

通告说，这四家法定机构和制服团体现有的建筑楼面面积定义各有不同。例如，市建局用总楼面面积（gross floor area，简称GFA）来衡量建筑物密度；土管局则以分层面积（strata area）作为业主拥有的面积。土管局用统计总建筑楼面面积（statistical GFA）来计算建筑楼面的总面积；民防部队则使用可通达楼面面积（accessible floor area）来制定防火规定。

此外，它们的面积测量方式也不一样。市建局在计算总楼面面积时会纳入建筑外墙的完整厚度，但不包括开敞空间（void area），例如两层楼空间（double-volume space）或阁楼（loft）。土管局则会把外墙的一半厚度和一些开敞空间算在楼面面积内。

这让土木工程从业员为了按照不同机构的要求准备申请文件时，耗费不少时间和精力。

这也为房地产业主在进行加建与改建（addition and alteration）工程时增添不少麻烦，也可能面对即使分层面积没有增加，但是由于计入开敞空间而导致总楼面面积有所增加的情况。

测量楼面面积至外墙的不同方式

资料来源 / 市区重建局　　早报制图 / 张进培

定义统一后，测量楼面面积时将包括建筑外墙一半的厚度。所有分层面积，不论是有遮盖还是无遮盖的面积，都会包括在总楼面面积内，开敞空间则不算在内。

另外，建设局和民防部队将采用相同的统计总建筑楼面面积定义。所有在明年6月1日或之后提交给市建局的发展项目申请，以及在今年9月1日或之后推出的政府售地计划地段和政府工业售地计划地段，将根据新定义处理。

ERA产业研究咨询部主管麦俊荣接受《联合早报》访问时说，一些房地产有两层楼空间或者阁楼，这些都算是分层面积，也是可出售楼面（saleable area）。

有些单位的每平方英尺售价把开敞空间计算在内，整体单位尺价会较低，若不把这些空间算在价格内，尺价就会更高。

新定义施行后，他预测一些发展商未来可能会减少这类空间设计，或只用于尺价本来就高的高档私宅。

“一些高档私宅的客厅天花板可能高达6.5公尺，适合吊大型水晶灯，显得豪华气派。发展商能以稀有双层楼空间设计为宣传重点，高价出售这些单位。“

橙易产业研究与咨询部总监孙燕清也有类似看法。

她受访时说：“有开敞空间的住房不多，主要是顶楼私宅单位或阁楼，所以对市场不会有太大的影响。这对买家有一定的帮助，他们会更清楚花钱买下了哪些空间。”

她还说，外墙的测量方式改变后的面积差别不会太明显，因为墙壁并不特别厚。

XML Introduction

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

- Documents have tags giving extra information about sections of the document
  - The markup takes the form of tags enclosed in angle brackets, <>.
  - Tags are used in pairs, with <tag> and </tag> delimiting the beginning and the end of the portion of the document to which the tag refers.
  - 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
  - Key feature for data representation and exchange
  - HTML with prescribed set of tags - document formatting

XML Representation of (part of) University Information.

University
- department
- course
- instructor
- teaches

- Tags make data (relatively) self-documenting
- Tags provide context for each value and allow the semantics of the value to be identified.

```
<university>
    <department>
        <dept_name> Comp. Sci. </dept_name>
        <building> Taylor </building>
        <budget> 100000 </budget>
    </department>
    <department>
        <dept_name> Biology </dept_name>
        <building> Watson </building>
        <budget> 90000 </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>
    <course>
        <course_id> BIO-301 </course_id>
        <title> Genetics </title>
        <dept_name> Biology </dept_name>
        <credits> 4 </credits>
    </course>
```

continued in the right

```
    <instructor>
        <IID> 10101 </IID>
        <name> Srinivasan </name>
        <dept_name> Comp. Sci. </dept_name>
        <salary> 65000 </salary>
    </instructor>
    <instructor>
        <IID> 83821 </IID>
        <name> Brandt </name>
        <dept_name> Comp. Sci. </dept_name>
        <salary> 92000 </salary>
    </instructor>
    <instructor>
        <IID> 76766 </IID>
        <name> Crick </name>
        <dept_name> Biology </dept_name>
        <salary> 72000 </salary>
    </instructor>
    <teaches>
        <IID> 10101 </IID>
        <course_id> CS-101 </course_id>
    </teaches>
    <teaches>
        <IID> 83821 </IID>
        <course_id> CS-101 </course_id>
    </teaches>
    <teaches>
        <IID> 76766 </IID>
        <course_id> BIO-301 </course_id>
    </teaches>
</university>
```

IID - identifier of the instructor

XML Representation of a Purchase Order

- In the previous university example, the XML data representation does not provide any significant benefit over the traditional relational data representation (*use as example for its simplicity*)

- A more realistic use of XML
  - Traditionally, puchase orders are 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.
  - The two organizations must agree on what tags appear in the purchase order, and what they mean.
  - The *nested representation* allows all information in a purchase order to be represented naturally in a single document.
  - *How to store the data using relational model?*

```
<purchase_order>
    <identifier> P-101 </identifier>
    <purchaser>
        <name> Cray Z. Coyote </name>
        <address> Mesa Flats, Route 66, Arizona 12345, USA  </address>
    </purchaser>
    <supplier>
        <name> Acme Supplies  </name>
        <address> 1 Broadway, New York, NY, USA  </address>
    </supplier>
    <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>
    <shipping_mode> 1-second-delivery </shipping_mode>
</purchaseorder>
```

XML - Motivation

Despite the inefficiency in storing repeated tags, XML has the following significant advantages for data exchange and storing complex structured information in files

- First, the presence of the tags makes the message *self-documenting*
  - I.e., a schema need not be consulted to understand the meaning of the text.

```
<purchase_order>
    <identifier> P-101 </identifier>
    <purchaser>
        <name> Cray Z. Coyote </name>
        <address> Mesa Flats, Route 66, Arizona 12345, USA </address>
    </purchaser>
    <supplier>
        <name> Acme Supplies </name>
        <address> 1 Broadway, New York, NY, USA </address>
    </supplier>
    <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>
    <shipping_mode> 1-second-delivery </shipping_mode>
</purchaseorder>
```

XML - Motivation

Despite the inefficiency in storing repeated tags, XML has the following significant advantages for data exchange and storing complex structured information in files

- First, the presence of the tags makes the message *self-documenting*
  - I.e., a schema need not be consulted to understand the meaning of the text.
- Second, the format of the document is *not rigid*.
  - Allow data to evolve over time - add new tags / ignore unexpected tags
  - Easy to represent multivalued attributes - multiple occurrences of the same tag

```
<purchase_order>
    <identifier> P-101 </identifier>
    <purchaser>
        <name> Cray Z. Coyote </name>
        <address> Mesa Flats, Route 66, Arizona 12345, USA </address>
    </purchaser>
    <supplier>
        <name> Acme Supplies </name>
        <address> 1 Broadway, New York, NY, USA </address>
    </supplier>
    <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>
    <shipping_mode> 1-second-delivery </shipping_mode>
</purchaseorder>
```

XML - Motivation (Cont.)

- Third, XML allows nested structures.
- E.g., Each purchase order has a purchaser and a list of items as two of its ***nested structures***
- *How to store such information in relational model?*

```
<purchase_order>
    <identifier> P-101 </identifier>
    <purchaser>
        <name> Cray Z. Coyote </name>
        <address> Mesa Flats, Route 66, Arizona 12345, USA </address>
    </purchaser>
    <supplier>
        <name> Acme Supplies </name>
        <address> 1 Broadway, New York, NY, USA </address>
    </supplier>
    <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>
    <shipping_mode> 1-second-delivery </shipping_mode>
</purchaseorder>
```

XML - Motivation (Cont.)

- Third, XML allows nested structures.
- E.g., Each purchase order has a purchaser and a list of items as two of its ***nested structures***
- *How to store such information in relational model?*
- Such information would have been split into multiple relations in a relational schema.
  - Item information would have been stored in one relation, purchaser information in a second relation, purchase orders in a third, and the relationship between purchase orders, purchasers, and items would have been stored in a fourth relation.

```
<purchase_order>
    <identifier> P-101 </identifier>
    <purchaser>
        <name> Cray Z. Coyote </name>
        <address> Mesa Flats, Route 66, Arizona 12345, USA </address>
    </purchaser>
    <supplier>
        <name> Acme Supplies </name>
        <address> 1 Broadway, New York, NY, USA </address>
    </supplier>
    <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>
    <shipping_mode> 1-second-delivery </shipping_mode>
</purchaseorder>
```

XML - Motivation (Cont.)

- Third, XML allows nested structures.
- E.g., Each purchase order has a purchaser and a list of items as two of its ***nested structures***
- *How to store such information in relational model?*
- Such information would have been split into multiple relations in a relational schema.
  - The relational representation helps to avoid redundancy; for example, item descriptions would be stored only once for each item identifier in a normalized relational schema.

```
<purchase_order>
    <identifier> P-101 </identifier>
    <purchaser>
        <name> Cray Z. Coyote </name>
        <address> Mesa Flats, Route 66, Arizona 12345, USA </address>
    </purchaser>
    <supplier>
        <name> Acme Supplies </name>
        <address> 1 Broadway, New York, NY, USA </address>
    </supplier>
    <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>
    <shipping_mode> 1-second-delivery </shipping_mode>
</purchaseorder>
```

Motivation for Nesting

- Nesting of data is useful in data transfer
  - Example: elements representing item nested within an itemlist element

- Nesting is not supported, or discouraged, in relational databases
  - With multiple orders, customer name and address are stored redundantly
  - Normalization replaces nested structures in each order by foreign key into table storing customer name and address information
  - Nesting is supported in object-relational databases

- But nesting is appropriate when transferring data
  - External application does not have direct access to data referenced by a foreign key
  - E.g., Gathering all information related to a purchase order into a single nested structure is attractive when information has to be exchanged with external parties

XML - Motivation (Cont.)

- Finally, since the XML format is widely accepted, a wide variety of tools are available to assist in its processing, including programming language APIs to create and to read XML data, browser software, and database tools.

```
<?xml version="1.0" ?>
<person>
  <name>john</name>
  <age>20</age>
</person>
```

XML

```
In [1]: import xmltodict, json

In [2]: xmltodict.parse("""<?xml version="1.0" ?>
                          <person>
                              <name>john</name>
                              <age>20</age>
                          </person>""")

Out[2]: OrderedDict([('person', OrderedDict([('name', 'john'), ('age', '2
        0')]))])

In [3]: print(json.dumps(xmltodict.parse("""<?xml version="1.0" ?>
                                           <person>
                                               <name>john</name>
                                               <age>20</age>
                                           </person>""")))

        {"person": {"name": "john", "age": "20"}}
```

Python Jupyter Notebook

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
  - Allows nested structures
  - Wide acceptance, not only in database systems, but also in browsers, tools, and applications

Structure of XML Data

- **Tag**: label for a section of data
- **Element**: section of data beginning with <*tagname*> and ending with matching </*tagname*>
- Elements must be properly nested
  - Proper nesting
    - <course> … <title> …. </title> </course>
  - Improper nesting
    - <course> … <title> …. </course> </title>
  - Text is said to appear **in the context of** an element if it appears between the start-tag and end-tag of that element.
  - Formally: Tags are *properly nested* if every start-tag has a unique matching end-tag that is in the context of the same parent element.
- Every document must have a single top-level element

Structure of XML Data (Cont.)

- 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 markup, but discouraged for data representation

Nested XML Representation of University Information

```
<university>
    <department>
        <dept_name> Comp. Sci. </dept_name>
        <building> Taylor </building>
        <budget> 100000 </budget>
    </department>
    <department>
        <dept_name> Biology </dept_name>
        <building> Watson </building>
        <budget> 90000 </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>
    <course>
        <course_id> BIO-301 </course_id>
        <title> Genetics </title>
        <dept_name> Biology </dept_name>
        <credits> 4 </credits>
    </course>
```

```
<university-1>
    <department>
        <dept_name> Comp. Sci. </dept_name>
        <building> Taylor </building>
        <budget> 100000 </budget>
        <course>
            <course_id> CS-101 </course_id>
            <title> Intro. to Computer Science </title>
            <credits> 4 </credits>
        </course>
        <course>
            <course_id> CS-347 </course_id>
            <title> Database System Concepts </title>
            <credits> 3 </credits>
        </course>
    </department>
    <department>
        <dept_name> Biology </dept_name>
        <building> Watson </building>
        <budget> 90000 </budget>
        <course>
            <course_id> BIO-301 </course_id>
            <title> Genetics </title>
            <credits> 4 </credits>
        </course>
    </department>
    <instructor>
        <IID> 10101 </IID>
        <name> Srinivasan </name>
        <dept_name> Comp. Sci. </dept_name>
        <salary> 65000. </salary>
        <course_id> CS-101 </course_id>
    </instructor>
</university-1>
```

Nested XML Representation of University Information

```
<university>
    <department>
        <dept_name> Comp. Sci. </dept_name>
        <building> Taylor </building>
        <budget> 100000 </budget>
    </department>
    <department>
        <dept_name> Biology </dept_name>
        <building> Watson </building>
        <budget> 90000 </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>
    <course>
        <course_id> BIO-301 </course_id>
        <title> Genetics </title>
        <dept_name> Biology </dept_name>
        <credits> 4 </credits>
    </course>
```

```
<university-1>
    <department>
        <dept_name> Comp. Sci. </dept_name>
        <building> Taylor </building>
        <budget> 100000 </budget>
        <course>
            <course_id> CS-101 </course_id>
            <title> Intro. to Computer Science </title>
            <credits> 4 </credits>
        </course>
        <course>
            <course_id> CS-347 </course_id>
            <title> Database System Concepts </title>
            <credits> 3 </credits>
        </course>
    </department>
    <department>
        <dept_name> Biology </dept_name>
        <building> Watson </building>
        <budget> 90000 </budget>
        <course>
            <course_id> BIO-301 </course_id>
            <title> Genetics </title>
            <credits> 4 </credits>
        </course>
    </department>
    <instructor>
        <IID> 10101 </IID>
        <name> Srinivasan </name>
        <dept_name> Comp. Sci. </dept_name>
        <salary> 65000. </salary>
        <course_id> CS-101 </course_id>
    </instructor>
</university-1>
```

Nested XML Representation of University Information

```
<university>
    <department>
        <dept_name> Comp. Sci. </dept_name>
        <building> Taylor </building>
        <budget> 100000 </budget>
    </department>
    <department>
        <dept_name> Biology </dept_name>
        <building> Watson </building>
        <budget> 90000 </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>
    <course>
        <course_id> BIO-301 </course_id>
        <title> Genetics </title>
        <dept_name> Biology </dept_name>
        <credits> 4 </credits>
    </course>
```

```
<university-1>
    <department>
        <dept_name> Comp. Sci. </dept_name>
        <building> Taylor </building>
        <budget> 100000 </budget>
        <course>
            <course_id> CS-101 </course_id>
            <title> Intro. to Computer Science </title>
            <credits> 4 </credits>
        </course>
        <course>
            <course_id> CS-347 </course_id>
            <title> Database System Concepts </title>
            <credits> 3 </credits>
        </course>
    </department>
    <department>
        <dept_name> Biology </dept_name>
        <building> Watson </building>
        <budget> 90000 </budget>
        <course>
            <course_id> BIO-301 </course_id>
            <title> Genetics </title>
            <credits> 4 </credits>
        </course>
    </department>
    <instructor>
        <IID> 10101 </IID>
        <name> Srinivasan </name>
        <dept_name> Comp. Sci. </dept_name>
        <salary> 65000. </salary>
        <course_id> CS-101 </course_id>
    </instructor>
</university-1>
```

Nested XML Representation of University Information

```xml
<university-2>
    <instructor>
        <ID> 10101 </ID>
        <name> Srinivasan </name>
        <dept_name> Comp. Sci.</dept_name>
        <salary> 65000 </salary>
        <teaches>
            <course>
                <course_id> CS-101 </course_id>
                <title> Intro. to Computer Science </title>
                <dept_name> Comp. Sci. </dept_name>
                <credits> 4 </credits>
            </course>
        </teaches>
    </instructor>

    <instructor>
        <ID> 83821 </ID>
        <name> Brandt </name>
        <dept_name> Comp. Sci.</dept_name>
        <salary> 92000 </salary>
        <teaches>
            <course>
                <course_id> CS-101 </course_id>
                <title> Intro. to Computer Science </title>
                <dept_name> Comp. Sci. </dept_name>
                <credits> 4 </credits>
            </course>
        </teaches>
    </instructor>
</university-2>
```

Nested XML Representation of University Information

- Details of courses taught by an instructor are stored nested within the instructor element

```
<university-2>
    <instructor>
        <ID> 10101 </ID>
        <name> Srinivasan </name>
        <dept_name> Comp. Sci.</dept_name>
        <salary> 65000 </salary>
        <teaches>
            <course>
                <course_id> CS-101 </course_id>
                <title> Intro. to Computer Science </title>
                <dept_name> Comp. Sci. </dept_name>
                <credits> 4 </credits>
            </course>
        </teaches>
    </instructor>

    <instructor>
        <ID> 83821 </ID>
        <name> Brandt </name>
        <dept_name> Comp. Sci.</dept_name>
        <salary> 92000 </salary>
        <teaches>
            <course>
                <course_id> CS-101 </course_id>
                <title> Intro. to Computer Science </title>
                <dept_name> Comp. Sci. </dept_name>
                <credits> 4 </credits>
            </course>
        </teaches>
    </instructor>
</university-2>
```

Nested XML Representation of University Information

- Details of courses taught by an instructor are stored nested within the instructor element
- Redundancy occurs if a course is taught by more than one instructor
    - Similar to items in purchase order
- Nested representations are widely used to avoid joins.
    - Different from a normalized representation

```
<university-2>
    <instructor>
        <ID> 10101 </ID>
        <name> Srinivasan </name>
        <dept_name> Comp. Sci.</dept_name>
        <salary> 65000 </salary>
        <teaches>
            <course>
                <course_id> CS-101 </course_id>
                <title> Intro. to Computer Science </title>
                <dept_name> Comp. Sci. </dept_name>
                <credits> 4 </credits>
            </course>
        </teaches>
    </instructor>

    <instructor>
        <ID> 83821 </ID>
        <name> Brandt </name>
        <dept_name> Comp. Sci.</dept_name>
        <salary> 92000 </salary>
        <teaches>
            <course>
                <course_id> CS-101 </course_id>
                <title> Intro. to Computer Science </title>
                <dept_name> Comp. Sci. </dept_name>
                <credits> 4 </credits>
            </course>
        </teaches>
    </instructor>
</university-2>
```

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>
```

```
<course>
    <course_id> CS-101 </course_id>
    <title> Intro. to Computer Science </title>
    <dept_name> Comp. Sci. </dept_name>
    <credits> 4 </credits>
</course>
```

- Attributes of an element are specified by *name=value* pairs inside the starting tag of an element, before the closing “>” of the tag.
  - Attribute values must always be quoted (either single or double quotes) - strings without markup

- 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 construction, 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 contents

Elements Containing No Subelements

- An element of the form <element></element> that contains no subelements or text content can be abbreviated as <element/>;

- Abbreviated elements may, however, contain attributes.

```
<?xml version="1.0" encoding="ISO-8859-15"?>
<package destination="SU" origin="ASR" version="1.0">
  <recognized_sentence>
    <information>
      I would like the train fares from Valencia to Madrid
    </information>
    <confidences>
        <word confidence="0.47" value="I" />
        <word confidence="0.68" value="would" />
        <word confidence="0.53" value="like" />
        <word confidence="0.75" value="the" />
        <word confidence="0.64" value="train" />
        <word confidence="0.56" value="fares" />
        <word confidence="0.84" value="from" />
        <word confidence="0.93" value="Valencia" />
        <word confidence="0.78" value="to" />
        <word confidence="0.93" value="Madrid" />
    </confidences>
  </recognized_sentence>
  <grammar name="dihana.jsgf">
</package>
```

Namespaces

- XML data has to be exchanged between organizations
- *Problem:* Same tag name may have different meaning in different organizations, causing confusion on exchanged documents
- *Possible solution:* 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**

Namespaces (Cont.)

- The idea of a namespace is to prepend each tag or attribute with a universal resource identifier (e.g., a web address).

- For example, Yale University wanted to ensure that XML documents it created would not duplicate tags used by any business partner's XML documents

```
<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>
```

Attribute xmlns:yale is defined as the ***abbreviation*** *for the URL*

- The university uses a web URL such as *http://www.yale.edu* as a unique identifier, and prepends it with a colon to each tag name (too long...)

More on XML Syntax

- 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"

  - A CDATA section starts with

    <![CDATA[

    and ends with

    ]]>

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 (*optional*)
- 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
  - All values represented as strings in XML
- DTD syntax
  - <!ELEMENT element (subelements-specification) >
  - <!ATTLIST element (attributes) >

Element Specification in DTD

<!ELEMENT element (subelements-specification) >

- Subelements can be specified as
  - Names of elements, or
  - #PCDATA (parsed character data), i.e., text data
  - EMPTY (no subelements/content) or ANY (anything can be a subelement)
- Example

  <!ELEMENT department (dept_name building, budget)>
  <!ELEMENT dept_name (#PCDATA)>
  <!ELEMENT budget (#PCDATA)>
- Subelement specification may have *regular expressions*

  E.g., <!ELEMENT university ((department | course | instructor | teaches)+)>
  - Notation:
    - "|" - alternatives, i.e., 'or'
    - "+" - 1 or more occurrences
    - "*" - 0 or more occurrences

REGEX Cheat Sheet

REGEX SYNTAX	MEANING	EXAMPLE	MATCHES	DOES NOT MATCH
.	Any single character	go.gle	google, goggle	gogle
[abc]	Any of these character	analy[zs]e	analyse, analyze	analyxe
[a-z]	Any character in this range	demo[2-4]	demo2, demo3	demo1, demo5
[^abc]	None of these characters	analy[^zs]e	analyxe	analyse, analyze
[^a-z]	Not a character in this range	demo[^2-4]	demo1, demo5	demo2, demo3
|	Or	demo|example	demo, demos, example	test
^	Starts with	^demo	demos, demonstration	my demo
$	Ends with	demo$	my demo	demonstration
?	Zero or one times (greedy)	demos?123	demo123, demos123	demoA123
??	Zero or one times (lazy)			
*	Zero or more times (greedy)	goo*gle	gogle, goooogle	goggle
*?	Zero or more times (lazy)			
+	One or more times (greedy)	goo+gle	google, gooogle	gogle, goggle
+?	One or more times (lazy)			
{n}	n times exactly	w{3}	www	w, ww
{n,m}	from n to m times	a{4, 7}	aaaa, aaaaa, aaaaaa, aaaaaaa	aaaaaaaa, aaa, a
{n,}	at least n times	go{2,}gle	google, gooogle, goooogle	ggle, gogle
()	Group	^(demo|example)[0-9]+	demo1, example4	demoexample2
(?:)	Passive group (Useful for filters)			
\	Escape	AU\$10	AU$10, AU$100	AU10, 10
\s	White space			
\S	Non-white space			
\d	Digit character			
\D	Non-digit character			
\w	Word			
\W	Non-word (e.g. punctuation, spaces)			

https://www.optimisation-conversion.com/wp-content/uploads/2015/05/regex-cheat-sheet.gif

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 )>
]>
```

Attribute Specification in DTD

<!ATTLIST element (attributes) > - Attributes must have a type declaration and a default declaration.

- Attribute specification : for each attribute
  - Name
  - Type of attribute (*type declaration*)
    - CDATA - character data
    - ID (identifier) or IDREF (ID reference) or IDREFS (multiple IDREFs)   *(more on this later)*
  - Whether (*default declaration*)
    - mandatory (#REQUIRED) - a value must be specified for the attribute in each element
    - has a default value (value),
    - or neither (#IMPLIED) - the document may omit this attribute
- 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 attribute of type ID provides a unique identifier for the element
- 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
    - *(We renamed the attribute ID of the instructor relation to IID to avoid confusion with the type ID)*

- An attribute of type IDREF is a reference to an element
- An attribute of type IDREF must contain the ID value of an element in the same document
- An attribute of type IDREFS contains a set of (0 or more) ID values, separated by space
    - Each ID value must contain the ID value of an element in the same document

University DTD with ID and IDREFS Attribute Types

```
<!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 >
    ··· declarations for title, credits, building,
        budget, name and salary ···
] >
```

- The course elements use **course_id** as their **identifier attribute**
- course_id has been made an attribute of course instead of a subelement.
- Additionally, each course element also contains an IDREF of the department and an IDREFS attribute instructors

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>
```

DTD

```
<!ELEMENT course (title, credits )>
<!ATTLIST course
    course_id ID #REQUIRED
    dept_name IDREF #REQUIRED
    instructors IDREFS #IMPLIED >
```

Limitations of DTDs

- DTDs are strongly connected to the *document formatting* heritage of XML (*unsuitable for data-processing applications*)
- No typing of text elements and attributes
  - All values are strings, no integers, reals, etc.
  - E.g., the element *balance* cannot be constrained to be a positive number.
- Difficult to specify unordered sets of subelements
  - Order is usually irrelevant in databases (unlike in the document-layout environment from which XML evolved)
  - (A | B)* allows specification of an unordered set, but
    - Cannot ensure that each of A and B occurs only once
- IDs and IDREFs are untyped
  - The *instructors* attribute of an course may contain a reference to 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. It supports
  - Typing of values
    - E.g., integer, string, boolean, etc
    - Also, constraints on min/max values
  - User-defined, comlex types (using constructors such as **complexType** and **sequence**)
  - Many more features, including
    - uniqueness and foreign key constraints, inheritance

- XML Schema is itself specified in XML syntax, unlike DTDs
  - More-standard representation, but verbose
- XML Scheme is integrated with namespaces
- **BUT**: XML Schema is significantly more complicated than DTDs.
- *The XML Schema language is also referred to as XML Schema Definition (XSD)*

XML Schema Version of University DTD

We prefix the XML Schema tag with the namespace prefix "xs:"

```
<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="course">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="course_id" type="xs:string"/>
            <xs:element name="title" type="xs:string"/>
            <xs:element name="dept_name" type="xs:string"/>
            <xs:element name="credits" type="xs:decimal"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
  ...
```

```
<!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 )>
]>
```

XML Schema Version of University 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>
```

```
<!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 )>
]>
```

- Defines the type UniversityType as containing zero or more occurrences of each of department, course, instructor, and teaches
- Note the use of **ref** to specify the occurrence of an element defined earlier.

Defining Attributes in XML Schema

- Define dept_name as an attribute:

```
<xs:attribute name = “dept name” type="xs:string"/>
<xs:attribute name = “dept name” type="xs:string" use="optional"/>
```

Defining Attributes in XML Schema

- Define dept_name as an attribute:

```
<xs:attribute name = “dept name” type="xs:string"/>
<xs:attribute name = “dept name” type="xs:string" use="optional"/>
```

Attributes by default are optional

Defining Attributes in XML Schema

- Define dept_name as an attribute:

```
<xs:attribute name = “dept name” type="xs:string"/>
<xs:attribute name = “dept name” type="xs:string" use="optional"/>
```

Attributes by default are optional

- *Sample XML Schema*
- *Sample XML*

```
<xs:element name="Order">
    <xs:complexType>
        <xs:attribute name="OrderID"
                      type="xs:int" />
    </xs:complexType>
</xs:element>
```

```
<Order OrderID="6" />
```

or

```
<Order />
```

```
<xs:element name="Order">
    <xs:complexType>
        <xs:attribute name="OrderID"
                      type="xs:int"
                      use="required" />
    </xs:complexType>
</xs:element>
```

```
<Order OrderID="6" />
```

More Features of XML Schema

- XML Schema allows the specification of keys and key references, corresponding to the primary-key and foreign-key definition in SQL.

- 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>
```

  - The **selector** is a path expression that defines the scopefor the constraint, and **field** declarations specify the elements or attributes that form the 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>
```

  - The **refer** attribute specifies the name of the key declaration that is being referenced

Benefits of XML Schema over DTDs

- It allows the text that appears in elements to be constrained to specific types, such as numeric types in specific formats or complex types such as sequences of ele ments of other types.
- It allows user-defined types to be created.
- It allows uniqueness and foreign-key constraints.
- It is integrated with namespaces to allow different parts of a document to conform to different schemas.

Querying and Transforming XML Data

- Translation of information from one XML schema to another
- Querying on XML data
  - Just as the output of a relational query is a relation, the output of an XML query can be an XML document.
  - Above two are closely related, and handled by the same tools

- Standard XML querying/translation languages
  - **XPath**
    - Simple language consisting of path expressions, a building block for XQuery
  - **XQuery**
    - The standard XML query language with a rich set of features
    - Modeled after SQL but is significantly different (nested XML data)
  - XSLT (primarily for document-formatting applications)
    - Simple language designed for translation from XML to XML, and XML to HTML

Tree Model of XML Data

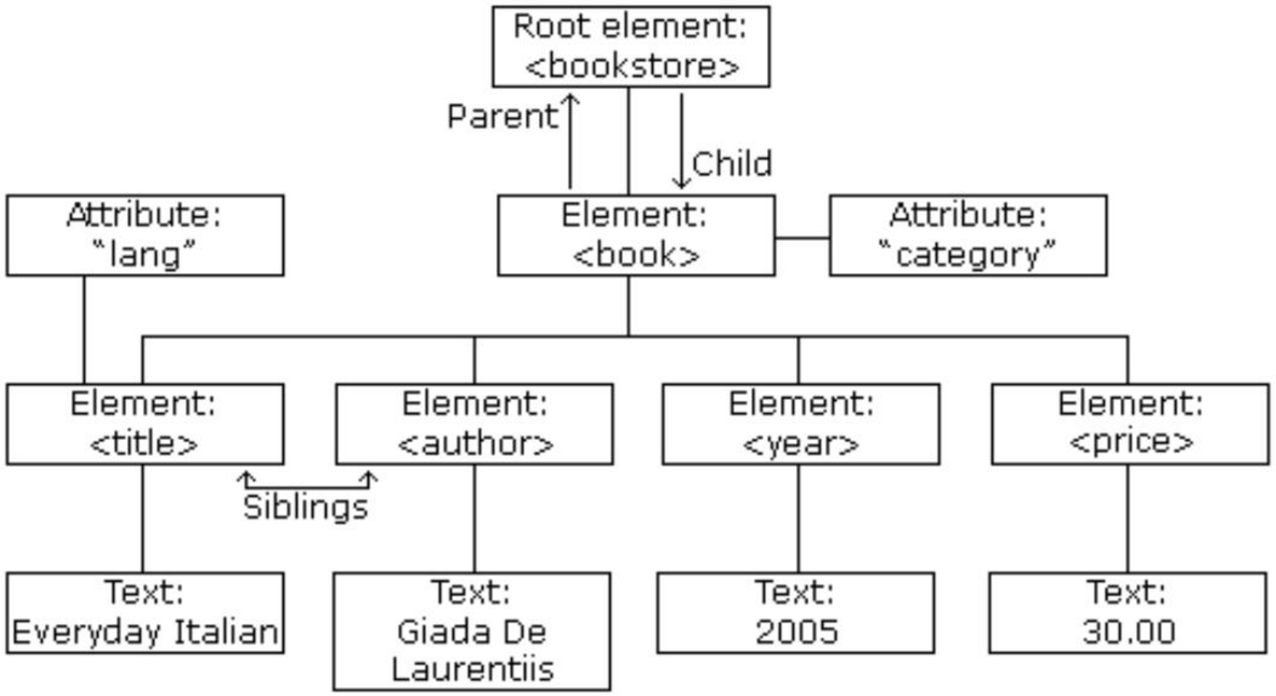
- Query and transformation languages are based on a **tree model** of XML data

- 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 content of an element is modeled as a text node child of the element
  - Element and attribute nodes (except for the root node) have a single parent, which is an element node
  - Children of a node are ordered according to their order in the XML document

  - The terms *parent, child, ancestor, descendant*, and *siblings* are used in the tree model of XML data.

Tree Model of XML Data - Example

```xml
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
  <book category="cooking">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
  <book category="children">
    <title lang="en">Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
  <book category="web">
    <title lang="en">Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
  </book>
</bookstore>
```



https://www.w3schools.com/xml/xml_tree.asp

XPath

- XPath is used to address (select) parts of an XML document using **path expressions**
  - XPath standard - XPath 2.0

- A path expression is a sequence of location steps separated by “/”
  - Unlike “.” operator that separates location steps in SQL
  - Think of file names in a directory hierarchy
  - E.g., /university-3/instructor/name

```
<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>
```

XPath

- XPath is used to address (select) parts of an XML document using **path expressions**
  - XPath standard - XPath 2.0
- A path expression is a sequence of location steps separated by “/”
  - Unlike “.” operator that separates location steps in SQL
  - Think of file names in a directory hierarchy
- Result of path expression: a set of nodes that along with their containing elements/attributes match the specified path

```
<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>
```

XPath

- XPath is used to address (select) parts of an XML document using **path expressions**
  - XPath standard - XPath 2.0
- A path expression is a sequence of location steps separated by “/”
  - Unlike “.” operator that separates location steps in SQL
  - Think of file names in a directory hierarchy
- Result of path expression: a set of nodes that along with their containing elements/attributes match the specified path
- E.g., /university-3/instructor/name evaluated on the university-3 data we saw earlier returns two elements

<name>Srinivasan</name>
<name>Brandt</name>

```
<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>
```

XPath

- XPath is used to address (select) parts of an XML document using **path expressions**
  - XPath standard - XPath 2.0
- A path expression is a sequence of location steps separated by “/”
  - Unlike “.” operator that separates location steps in SQL
  - Think of file names in a directory hierarchy
- Result of path expression: a set of nodes that along with their containing elements/attributes match the specified path
- E.g., /university-3/instructor/name evaluated on the university-3 data we saw earlier returns two elements

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

  returns the same names, but without the enclosing tags

```
<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>
```

XPath (Cont.)

- The initial "/" denotes root of the document (abstract root above the top-level tag)
- Path expressions are evaluated left to right
  - Each step operates on the set of nodes produced by the previous step

- For example, the expression /university-3 returns a single node corresponding to the

  <university-3>

  tag, while /university-3/instructor returns the ***two nodes*** corresponding to the

  instructor

  elements that are children of the **university-3** node

```
<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>
```

XPath (Cont.)

- Attribute values may also be accessed, using the “@” symbol.
  - E.g., /university-3/course/@course id
    - returns a set of all values of course_id attributes of course elements.
- Selection predicates may follow any step in a path, in [ ]
  - E.g., /university-3/course[credits >= 4]
    - returns course elements with a credits value greater than or equal to 4
    - /university-3/course[credits] returns course elements containing a credits subelement
  - E.g., /university-3/course[credits >= 4]/@course_id
    - returns the course identifiers of courses with credits >= 4

```
<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>
```

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
    - E.g., /university-2/instructor[position()<3]
      - Returns the first two instructors
- Boolean connectives and and or and function not() can be used in predicates

```
<university-2>
    <instructor>
        <ID> 10101 </ID>
        <name> Srinivasan </name>
        <dept_name> Comp. Sci.</dept_name>
        <salary> 65000 </salary>
        <teaches>
            <course>
                <course_id> CS-101 </course_id>
                <title> Intro. to Computer Science </title>
                <dept_name> Comp. Sci. </dept_name>
                <credits> 4 </credits>
            </course>
        </teaches>
    </instructor>

    <instructor>
        <ID> 83821 </ID>
        <name> Brandt </name>
        <dept_name> Comp. Sci.</dept_name>
        <salary> 92000 </salary>
        <teaches>
            <course>
                <course_id> CS-101 </course_id>
                <title> Intro. to Computer Science </title>
                <dept_name> Comp. Sci. </dept_name>
                <credits> 4 </credits>
            </course>
        </teaches>
    </instructor>
</university-2>
```

Functions in XPath (Cont.)

- The function id("foo") returns the node (if any) with an attribute of type ID and value "foo".
  - 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 the instructor elements referred to in the instructors attribute of course elements.

```
<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>
```

DTD

```
<!ELEMENT course (title, credits )>
<!ATTLIST course
    course_id ID #REQUIRED
    dept_name IDREF #REQUIRED
    instructors IDREFS #IMPLIED >
```

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.
- "//" 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.
    - This example illustrates the ability to find required data *without full knowledge of the schema*
- 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 from for specifying "all descendants"
  - ".." specifies the parent.
- doc(name) returns the root of a named document, e.g., doc("university.xml")/university/department

XQuery

- The World Wide Web Consortium (W3C) has developed XQuery as the standard query language for XML.
  - XQuery 1.0, which was released as a W3C recommendation on 23 January 2007

- XQuery queries are modeled after SQL queries but differ significantly from SQL. XQuery uses a
  for … let … where … order by … result …
  syntax
  for ⇔ SQL **from**
  where ⇔ SQL **where**
  order by ⇔ SQL **order by**
  result ⇔ SQL **select**
  let allows temporary variables, and has no equivalent in SQL

- They are referred to as “FLWOR” (pronounced “flower”) expressions
  - A FLWOR query need not contain all the clauses

FLWOR Syntax in XQuery

- Simple FLWOR expression in XQuery
  - Find all courses with credits > 3

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

  - Returns course identifiers with each result enclosed in an <course_id> .. </course_id> tag
  - **for** clause uses *XPath expressions*, and variable in **for** clause ranges over values in the set returned by XPath

```
<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>
```

FLWOR Syntax in XQuery

- Simple FLWOR expression in XQuery
  - Find all courses with credits > 3

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

  - Returns course identifiers with each result enclosed in an <course_id> .. </course_id> tag
  - **for** clause uses *XPath expressions*, and variable in **for** clause ranges over values in the set returned by XPath

- **let** clause not really needed in this query, and selection can be done in XPath. Query can be written as:

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

```xml
<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>
```

FLWOR Syntax in XQuery

- Simple FLWOR expression in XQuery
  - Find all courses with credits > 3

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

- The use of curly brackets (“{}”) in the return clause:
  - Items in the return clause are XML text unless enclosed in {}, in which case they are evaluated as expressions
  - E.g., **return** <course_id> $x/@course id </course_id>

    returns several copies of the string

    <course_id> $x/@course_id </course_id>
  - E.g., **return** <course course_id=“{$x/@course_id}” />

```
<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>
```

FLWOR Syntax in XQuery (Cont.)

- Alternative notation for constructing elements using the **element** and **attribute** constructors

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

- If the **return** clause in the previous query is replaced by the above **return** clause, the query would return course elements with
  - course_id and dept_name as attributes, and
  - title and credits as subelements

Joins

- Joins are specified in a manner very similar to SQL

```
for $c in /university/course,
     $i  in /university/instructor,
     $t  in /university/teaches
where $c/course_id= $t/course_id and $t/IID = $i/IID
return <course_instructor> { $c $i } </course_instructor>
```

```
<university>
    <department>
        <dept_name> Comp. Sci. </dept_name>
        <building> Taylor </building>
        <budget> 100000 </budget>
    </department>
    <department>
        <dept_name> Biology </dept_name>
        <building> Watson </building>
        <budget> 90000 </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>
    <course>
        <course_id> BIO-301 </course_id>
        <title> Genetics </title>
        <dept_name> Biology </dept_name>
        <credits> 4 </credits>
    </course>
     ...
    <instructor>
        <IID> 76766 </IID>
        <name> Crick </name>
        <dept_name> Biology </dept_name>
        <salary> 72000 </salary>
    </instructor>
    <teaches>
        <IID> 10101 </IID>
        <course_id> CS-101 </course_id>
    </teaches>
     ...
```

Joins

- Joins are specified in a manner very similar to SQL

```
for $c in /university/course,
     $i  in /university/instructor,
     $t  in /university/teaches
where $c/course_id= $t/course_id and $t/IID = $i/IID
return <course_instructor> { $c $i } </course_instructor>
```

- The same query can be expressed with the selections specified as XPath selections:

```
for $c in /university/course,
     $i in /university/instructor,
     $t in /university/teaches[ $c/course_id= $t/course_id
                                              and $t/IID = $i/IID]
return <course_instructor> { $c $i } </course_instructor>
```

```
<university>
    <department>
        <dept_name> Comp. Sci. </dept_name>
        <building> Taylor </building>
        <budget> 100000 </budget>
    </department>
    <department>
        <dept_name> Biology </dept_name>
        <building> Watson </building>
        <budget> 90000 </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>
    <course>
        <course_id> BIO-301 </course_id>
        <title> Genetics </title>
        <dept_name> Biology </dept_name>
        <credits> 4 </credits>
    </course>
     ...
    <instructor>
        <IID> 76766 </IID>
        <name> Crick </name>
        <dept_name> Biology </dept_name>
        <salary> 72000 </salary>
    </instructor>
    <teaches>
        <IID> 10101 </IID>
        <course_id> CS-101 </course_id>
    </teaches>
     ...
```

Comparison Operations on Sequences

- Path expressions may return a single value or element, or a sequence of values or elements.
  - Path expressions in XQuery are the same as path expressions in XPath 2.0.
  - In the absence of schema information, it may not be possible to infer whether a path expression returns a single value or a sequence of values.
  - Such path expressions may participate in comparison operations such as =,<, and >=

- XQuery has an interesting definition of comparison operations on sequences
  - For example, $x/credits > 3
    - If the result is a sequence containing multiple values, the expression evaluates to true if at least one of the values is greater than 3
  - For example, $x/credits = $y/credits
    - It evaluates to true if any one of the values returned by the first expression is equal to any one of the values returned by the second expression
  - If the above behavior is not appropriate, use operators **eq, ne, lt, gt, le, ge**
    - These raise an error if either of their inputs is a sequence with multiple values.

Nested Queries

- XQuery FLWOR expressions can be nested in the **return** clause in order to generate element nestings that do not appear in the source document.

```
<university>
    <department>
        <dept_name> Comp. Sci. </dept_name>
        <building> Taylor </building>
        <budget> 100000 </budget>
    </department>
    <department>
        <dept_name> Biology </dept_name>
        <building> Watson </building>
        <budget> 90000 </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>
    <course>
        <course_id> BIO-301 </course_id>
        <title> Genetics </title>
        <dept_name> Biology </dept_name>
        <credits> 4 </credits>
    </course>
     ...
    <instructor>
        <IID> 76766 </IID>
        <name> Crick </name>
        <dept_name> Biology </dept_name>
        <salary> 72000 </salary>
    </instructor>
    <teaches>
        <IID> 10101 </IID>
        <course_id> CS-101 </course_id>
    </teaches>
     ...
```

Nested Queries

- The following query converts data from the flat structure for university information into the nested structure used in university-1

```
<university-1>
{
    for $d in /university/department
    return
        <department>
            { $d/* }
            { for $c in /university/course[dept_name = $d/dept_name]
                return $c }
        </department>
}
{
    for $i in /university/instructor
    return
        <instructor>
            { $i/* }
            { for $c in /university/teaches[IID = $i/IID]
                return $c/course_id }
        </instructor>
}
</university-1>
```

  - $d/* denotes all the children of the node to which $d is bound, without the enclosing top-level tag

```
<university>
    <department>
        <dept_name> Comp. Sci. </dept_name>
        <building> Taylor </building>
        <budget> 100000 </budget>
    </department>
    <department>
        <dept_name> Biology </dept_name>
        <building> Watson </building>
        <budget> 90000 </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>
    <course>
        <course_id> BIO-301 </course_id>
        <title> Genetics </title>
        <dept_name> Biology </dept_name>
        <credits> 4 </credits>
    </course>
    ...
    <instructor>
        <IID> 76766 </IID>
        <name> Crick </name>
        <dept_name> Biology </dept_name>
        <salary> 72000 </salary>
    </instructor>
    <teaches>
        <IID> 10101 </IID>
        <course_id> CS-101 </course_id>
    </teaches>
    ...
```

Nested Queries

- The following query converts data from the flat structure for university information into the nested structure used in university-1

```
<university-1>
{
    for $d in /university/department
    return
        <department>
            { $d/* }
            { for $c in /university/course[dept_name = $d/dept_name]
                return $c }
        </department>
}
{
    for $i in /university/instructor
    return
        <instructor>
            { $i/* }
            { for $c in /university/teaches[IID = $i/IID]
                return $c/course_id }
        </instructor>
}
</university-1>
```

  - $d/* denotes all the children of the node to which $d is bound, without the enclosing top-level tag

```
<university-1>
    <department>
        <dept_name> Comp. Sci. </dept_name>
        <building> Taylor </building>
        <budget> 100000 </budget>
        <course>
            <course_id> CS-101 </course_id>
            <title> Intro. to Computer Science </title>
            <credits> 4 </credits>
        </course>
        <course>
            <course_id> CS-347 </course_id>
            <title> Database System Concepts </title>
            <credits> 3 </credits>
        </course>
    </department>
    <department>
        <dept_name> Biology </dept_name>
        <building> Watson </building>
        <budget> 90000 </budget>
        <course>
            <course_id> BIO-301 </course_id>
            <title> Genetics </title>
            <credits> 4 </credits>
        </course>
    </department>
    <instructor>
        <IID> 10101 </IID>
        <name> Srinivasan </name>
        <dept_name> Comp. Sci. </dept_name>
        <salary> 65000. </salary>
        <course_id> CS-101 </course_id>
    </instructor>
</university-1>
```

Aggregate Functions

- XQuery provides a variety of aggregate functions such as **sum()** and **count()** that can be applied on sequences of elements or values.

- The function distinct-values() applied on a sequence returns a sequence without duplication.

- Aggregation functions can be used in any XPath path expression.

- To avoid namespace conflicts, functions are associated with a namespace:

  http://www.w3.org/2005/xpath-functions

  which has a default namespace prefix of **fn**.

  - E.g., **fn:sum** or **fn:count**

Grouping and Aggregation

- Nested queries are used for grouping (XQuery does not provide a **group by** construct)

```
for $d in /university/department
return
      <department-total-salary>
          <dept_name> { $d/dept_name } </dept_name>
          <total_salary> { fn:sum(
               for $i in /university/instructor[dept_name = $d/dept_name]
               return $i/salary
             ) }
          </total_salary>
       </department-total-salary>
```

- The above query on the university XML schema finds the total salary of all instructors in each department

Sorting in XQuery

- Results can be sorted in XQuery by using the **order by** clause
- E.g., to return instructors sorted by name

```
for $i in /university/instructor
order by $i/name
return <instructor> { $i/* } </instructor>
```

- Use **order by** $i/name **descending** to sort in descending order

Sorting in XQuery (Cont.)

- Sorting can be done at multiple levels of nesting.
- Sort departments by dept_name, and with courses sorted by course_id within each department

```
<university-1> {
      for $d in /university/department
      order by $d/dept_name
      return
          <department>
             { $d/* }
             { for $c in /university/course[dept_name = $d/dept_name]
               order by $c/course_id
               return <course> { $c/* } </course> }
          </department>
} </university-1>
```

Functions and Types

- User defined functions using the type system of XML Schema

```
declare function local:dept_courses($iid as xs:string) as element(course)*
{
   for $i  in /university/instructor[IID = $iid],
        $c in /university/courses[dept_name = $i/dept_name]
   return $c
}
```

- The above user-defined function takes as input an instructor identifier and returns a list of all courses offered by the department to which the instructor belongs

- Namespace **xs:** is predefined by XQuery to be associated with the XML Schema namespace
- Namespace **local:** is predefined to be associated with XQuery local functions

Functions and Types

- User defined functions using the type system of XML Schema

```
declare function local:dept_courses($iid as xs:string) as element(course)*
{
   for $i  in /university/instructor[IID = $iid],
        $c in /university/courses[dept_name = $i/dept_name]
   return $c
}
```

- The type specifications are optional for function parameters and return values
- XQuery uses the type system of XML Schema
  - **element(course)** allows elements with the tag course
  - The type* indicates a sequence of values of that type
  - The above definition of function dept_courses specifies the return value as a sequence of course elements.

Functions and Types (Cont.)

- Function invocation

```
for $i in /university/instructor[name = "Srinivasan"],
return local:dept_courses($i/IID)
```

```
declare function local:dept_courses($iid as xs:string) as element(course)*
{
   for $i  in /university/instructor[IID = $iid],
        $c in /university/courses[dept_name = $i/dept_name]
   return $c
}
```

- Returns the department courses for the instructor(s) named Srinivasan

Type Conversion

- If a numeric value represented by a string is compared to a numeric type, type conversion from string to the numeric type is done *automatically*.
- When an element is passed to a function that expects a string value, type conversion to a string is done by concatenating all the text values contained (nested) within the element.
- E.g., function contains(a,b) checks if string a contains string b
  - If input **a** is an element, it will check if the element **a** contains the string **b** nested anywhere inside it
- XQuery also provides functions to convert between types
  - For instance, number(x) converts a string to a number.

Other XQuery Features

- XQuery supports **If-then-else** constructs that can be used within **return** clauses
- Universal and existential quantification in **where** clause predicates
  - **some** $e **in** *path* **satisfies** *P*
  - **every** $e **in** *path* **satisfies** *P*
  - E.g., "*find departments where every instructor has a salary greater than $50,000*"

    for $d **in** /university/department
    where every $i **in** /university/instructor[dept name=$d/dept name]
        satisfies $i/salary > 50000
    return $d
  - Note that if a department has no instructor, it will trivially satisfy the above condition.
  - Add **and fn:exists**(/university/instructor[dept name=$d/dept name]) to to ensure that there is at least one instructor in the department

XSLT

- A **stylesheet** stores formatting options for a document, usually separated from document
  - E.g. an HTML style sheet may specify font colors and sizes for headings, etc.

- The **XSL (eXtensible Stylesheet Language)** was originally designed for generating HTML from XML

- **XSLT (XSL Transformations)** 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

Application Program Interfaces to XML

- There are two standard application program interfaces to XML data
    - Both these APIs can be used to parse an XML document and create an in-memory representation of the document.
    - **SAX** (Simple API for XML)
        - An event model, user provides event handlers for parsing events
            - E.g., start of element, end of element
    - **DOM** (Document Object Model)
        - **XML** data is parsed into a tree representation, with each element represented by a node, called a DOMNode.
        - Variety of functions provided for traversing the DOM tree
        - E.g.: DOM API provides Node class with methods
          getParentNode( ), getFirstChild( ), getNextSibling( )
          getAttribute( ), getData( ) (for text node)
          getElementsByTagName( ), …
        - Also provides functions for updating DOM tree

Application Program Interfaces to XML

- There are two standard application program interfaces to XML data
  - **SAX** (Simple API for XML)
  - **DOM** (Document Object Model)
  - They are used for applications that deal with individual XML documents.

  - However, they are not suitable for querying large collections of XML data
  - Declarative querying mechanisms such as XPath and XQuery are better suited to this task.

Storage of XML Data

- XML data can be stored in
  - **Non-relational data stores**
    - Flat files
      - Natural for storing XML (file format)
      - But has all problems discussed in Chapter 1 (no concurrency, no recovery, …)
    - XML database
      - Database built specifically for storing XML data, supporting DOM model and declarative querying
      - Currently no commercial-grade systems
  - **Relational databases**
    - Data must be translated into relational form
    - Advantage: mature database systems
    - Disadvantages: overhead of translating data and queries (nested elements & recurring elements)

Storage of XML in Relational Databases

- **Alternatives**
  - String Representation
  - Tree Representation
  - Map to Relations

String Representation

- Small XML documents can be stored as string (**clob**) values
  - **clob -** Character Large OBject, part of the SQL:1999 standard data types

- For large XML documents, store each top level element as a string field of a tuple in a relational database
  - Use a single relation to store all elements, or
  - Use a separate relation for each top-level element type
    - E.g., department_elements, course_elements, instructor_elements, and teaches_elements
      - Each with a string-valued attribute to store the element
  - Store values of subelements/attributes as extra fields of the relation, and build indices on these fields
    - E.g., dept_name or course_id

```
<university>
    <department>
        <dept_name> Comp. Sci. </dept_name>
        <building> Taylor </building>
        <budget> 100000 </budget>
    </department>
    <department>
        <dept_name> Biology </dept_name>
        <building> Watson </building>
        <budget> 90000 </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>
    <course>
        <course_id> BIO-301 </course_id>
        <title> Genetics </title>
        <dept_name> Biology </dept_name>
        <credits> 4 </credits>
    </course>
     ...
    <instructor>
        <IID> 76766 </IID>
        <name> Crick </name>
        <dept_name> Biology </dept_name>
        <salary> 72000 </salary>
    </instructor>
    <teaches>
        <IID> 10101 </IID>
        <course_id> CS-101 </course_id>
    </teaches>
     ...
```

String Representation (Cont.)

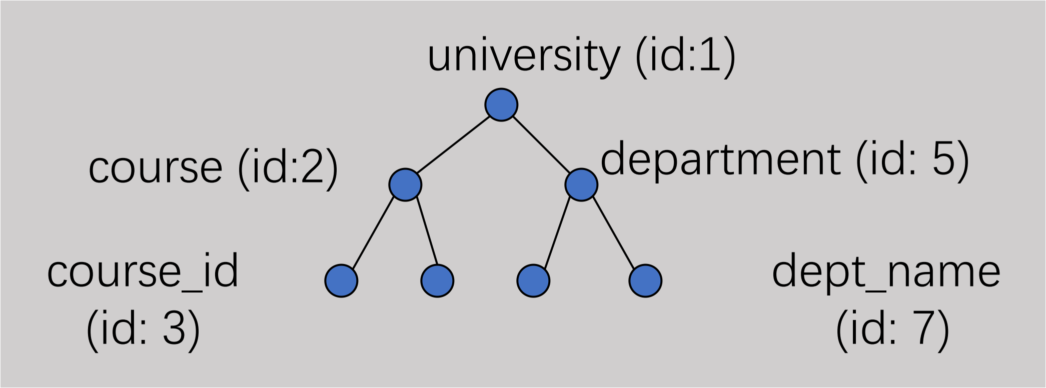
- **Benefits**:
  - Can store any XML data even without DTD
  - As long as there are many top-level elements in a document, strings are small compared to full document
    - Allows fast access to individual elements.

- **Drawback:** Need to parse strings to access values inside the elements
  - Parsing is slow.

Tree Representation

- **Tree representation:** model XML data as a tree and store using a relation

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 (Cont.)

- **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
  - A relation attribute corresponding to each subelement of simple type (i.e., cannot have attributes or subelements)
- A relation is created for complex subelements
  - Store the id of the subelement in the relation of the elment

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
  - A relation attribute corresponding to each subelement of simple type (i.e., cannot have attributes or subelements)
- A relation is created for complex subelements
  - Store the id of the subelement in the relation of the elment

Relations:
department(id, dept_name, building, budget)
course(parent_id, course_id, dept_name, title, credits)

```
<university-1>
    <department>
        <dept_name> Comp. Sci. </dept_name>
        <building> Taylor </building>
        <budget> 100000 </budget>
        <course>
            <course_id> CS-101 </course_id>
            <title> Intro. to Computer Science </title>
            <credits> 4 </credits>
        </course>
        <course>
            <course_id> CS-347 </course_id>
            <title> Database System Concepts </title>
            <credits> 3 </credits>
        </course>
    </department>
    <department>
        <dept_name> Biology </dept_name>
        <building> Watson </building>
        <budget> 90000 </budget>
        <course>
            <course_id> BIO-301 </course_id>
            <title> Genetics </title>
            <credits> 4 </credits>
        </course>
    </department>
    <instructor>
        <IID> 10101 </IID>
        <name> Srinivasan </name>
        <dept_name> Comp. Sci. </dept_name>
        <salary> 65000. </salary>
        <course_id> CS-101 </course_id>
    </instructor>
</university-1>
```

Publishing and Shredding XML Data

- When XML is used to exchange data originated in relational databases between business applications.

- **Publishing**: process of converting relational data to an XML format for export to other applications
- **Shredding**: process of converting back from XML to normalized relation form and stored in a relational database

- XML-enabled database systems support automated publishing and shredding
  - *Publishing* - A simple relation to XML mapping might create an XML element for every row of a table and make each column in that row a subelement of the XML element.
  - *Shredding* - A straightforward inverse of the mapping used to publish the data
    - Or, a mapping can be generated as outlined in the previous slide

Native Storage within a Relational Database

- Many systems offer *native storage* of XML data using the new **xml** data type.

- XML query languages such as XPath and XQuery are supported to query XML data.

  - Allow XQuery queries to be embedded within SQL queries

  - An XQuery query can be executed on a single XML document and can be embedded within an SQL query to allow it to execute on each of a collection of documents, with each document stored in a separate tuple.

SQL/XML

- New standard SQL extension that allows creation of nested XML output (*publishing*)
  - Each output tuple is mapped to an XML element *row*

```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>
        <row>
            <course_id> BIO-301 </course_id>
            <title> Genetics </title>
            <dept_name> Biology </dept_name>
            <credits> 4 </credits>
        </row>
    </course>
</university>
```

SQL/XML representation

Containing the relations department and course

```xml
<university>
    <department>
        <dept_name> Comp. Sci. </dept_name>
        <building> Taylor </building>
        <budget> 100000 </budget>
    </department>
    <department>
        <dept_name> Biology </dept_name>
        <building> Watson </building>
        <budget> 90000 </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>
    <course>
        <course_id> BIO-301 </course_id>
        <title> Genetics </title>
        <dept_name> Biology </dept_name>
        <credits> 4 </credits>
    </course>
     ...
    <instructor>
        <IID> 76766 </IID>
        <name> Crick </name>
        <dept_name> Biology </dept_name>
        <salary> 72000 </salary>
    </instructor>
    <teaches>
        <IID> 10101 </IID>
        <course_id> CS-101 </course_id>
    </teaches>
     ...
```

SQL Extensions

SQL/XML adds several operators and aggregate operations to SQL to allow the construction of XML output directly from the extended SQL.

- **xmlelement** creates XML elements
- **xmlattributes** creates attributes
- E.g., creates an XML element for each course, with the course identifier and department name represented as attributes, and title and credits as subelements.

```
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 (Cont.)

- **xmlforest** creates a forest (collection) of subelements

```
SELECT XMLELEMENT("employee",
         XMLFOREST(
           e.empno AS "works_number",
           e.ename AS "name",
           e.job AS "job")
       ) AS employee
FROM   emp e
WHERE  e.empno = 7782;
```

SQL

```
<employee>
    <works_number>7782</works_number>
    <name>CLARK</name>
    <job>MANAGER</job>
</employee>
```

XML

https://oracle-base.com/articles/misc/sqlxml-sqlx-generating-xml-content-using-sql#xmlforest

SQL Extensions (Cont.)

- **xmlagg** creates a forest of XML elements

```
SELECT XMLELEMENT("employee",
         XMLFOREST(
           e.empno AS "works_number",
           e.ename AS "name")
       ) AS employees
FROM   emp e
WHERE  e.deptno = 10;
```

```
<employee><works_number>7782</works_number><name>CLARK</name></employee>
<employee><works_number>7839</works_number><name>KING</name></employee>
<employee><works_number>7934</works_number><name>MILLER</name></employe>
```

```
SELECT XMLAGG(
         XMLELEMENT("employee",
           XMLFOREST(
             e.empno AS "works_number",
             e.ename AS "name")
         )
       ) AS employees
FROM   emp e
WHERE  e.deptno = 10;
```

```
<employee><works_number>7782</works_number><name>CLARK</name></employee>
<employee><works_number>7839</works_number><name>KING</name></employee><
employee><works_number>7934</works_number><name>MILLER</name></employee>
```

https://oracle-base.com/articles/misc/sqlxml-sqlx-generating-xml-content-using-sql#xmlagg

SQL Extensions (Cont.)

- **xmlagg** creates a forest of XML elements

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

- The above query creates 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

XML Applications

- Storing and exchanging data with complex structures
  - Store data that are structured, but are not easily modeled as relations
    - E.g., user preferences of a browser, with a large number of fields and some are multivalued
  - Storing documents, spreadsheet data, and other data that are part of **office** application packages
    - E.g., Open Document Format (ODF) format standard for storing Open Office and Office Open XML (OOXML) format standard for storing Microsoft Office documents
  - Numerous other standards for a variety of applications
    - **ChemML** - a standard for representing information about chemicals, such as their molecular structure, and a variety of important properties, such as boiling and melting points, calorific values, and solubility in various solvents.
    - **RosettaNet** - a standard for **e-business** applications, defining XML schemas and semantics for representing data as well as standards for message exchange

Standardized Data Exchange Formats

- Using normalized relational schemas to model such complex data requirements would result in a large number of relations that do not correspond directly to the objects that are being modeled.
- Explicit representation of attribute/element names along with values in XML helps avoid confusion between attributes.
- Nested element representations help reduce the number of relations that must be represented, as well as the number of joins required to get required information
  - At the possible cost of redundancy.
  - More natural for humans to read.

```xml
<university-1>
    <department>
        <dept_name> Comp. Sci. </dept_name>
        <building> Taylor </building>
        <budget> 100000 </budget>
        <course>
            <course_id> CS-101 </course_id>
            <title> Intro. to Computer Science </title>
            <credits> 4 </credits>
        </course>
        <course>
            <course_id> CS-347 </course_id>
            <title> Database System Concepts </title>
            <credits> 3 </credits>
        </course>
    </department>
    <department>
        <dept_name> Biology </dept_name>
        <building> Watson </building>
        <budget> 90000 </budget>
        <course>
            <course_id> BIO-301 </course_id>
            <title> Genetics </title>
            <credits> 4 </credits>
        </course>
    </department>
    <instructor>
        <IID> 10101 </IID>
        <name> Srinivasan </name>
        <dept_name> Comp. Sci. </dept_name>
        <salary> 65000. </salary>
        <course_id> CS-101 </course_id>
    </instructor>
</university-1>
```

Web Services

- The information provider defines procedures whose input and output are both in XML format.
  - For information accessed by software programs rather than by end users
- The Simple Object Access Protocol (SOAP) standard:
  - For invocation of procedures across applications with distinct databases
  - XML used to represent procedure input and output
  - For example, Amazon and Google provide SOAP-based procedures to carry out search and other activities.
- A *Web service* is a site providing a collection of SOAP procedures
  - Described using the Web Services Description Language (WSDL)
  - Directories of Web services are described using the Universal Description, Discovery, and Integration (UDDI) standard
  - To invoke a web service, a client must prepare an appropriate SOAP XML message and send it to the service; when it gets the result encoded in XML, the client must then extract information from the XML result.