



SEVENTH EDITION

# Database System Concepts

## Advance Database -Lecture 1

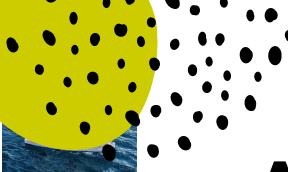
By Dr. Taghinezhad

1

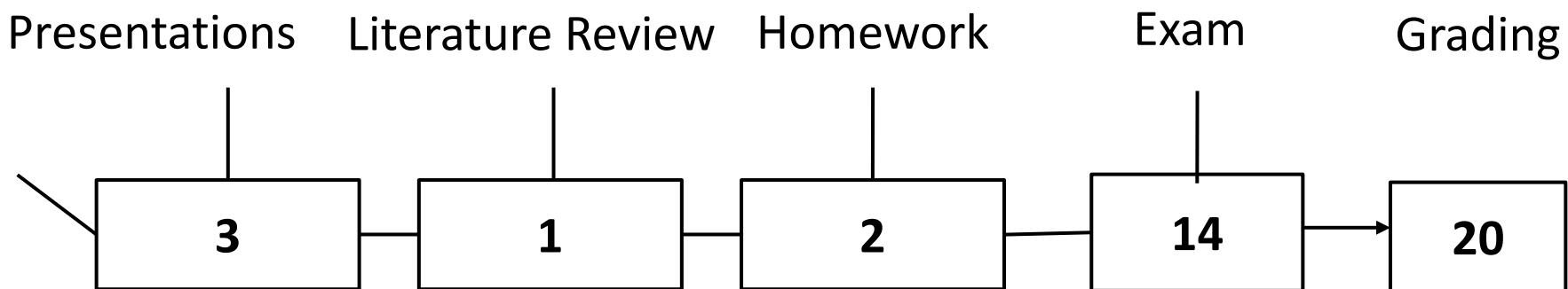
Mail:

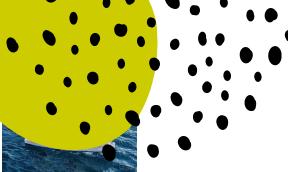
a0taghinezhad@gmail.com





# About This Course





# Projects include:

Each student needs to **review at least three papers** and write a **literature review** on these subjects. The subjects are as follows, but not limited to them:

- 1. immutable databases, decentralized databases, and smart contracts**
- 2. Real-time Data Processing with In-memory Databases:** This is a hot topic as it offers a glimpse into the exciting future of database research<sup>1</sup>.
- 3. Cloud-Inspired Operating Models:** Cloud-inspired operating models, cybersecurity, and data insights are among the top enterprise storage trends of 2023-4.
- 4. Artificial Intelligence and Database Technology**
- 5. Object Storage as Primary Storage, LLM**



# Outlines of this course

- Complex Data Types
  - Semi-structured data
  - Textual data
  - Spatial Data
- Transaction
- Concurrency
- Recovery
- Parallel and Distributed Database



# Chapter 8: Complex Data Types

Database System Concepts, 7<sup>th</sup> Ed.

©Silberschatz, Korth and Sudarshan

See [www.db-book.com](http://www.db-book.com) for conditions on re-use



# Outline

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



# Relational Model and Non-Atomic Data Types

- The relational model is widely used for data representation across various application domains.
- A key requirement of **the relational model** is **atomic data values**, **disallowing multivalued, composite, and other complex data types**.
- **the relational model's constraints** on data types can **cause more problems than they solve** in certain applications.
- **Non-atomic data types include semi-structured data, object-based data, textual data, and spatial data.**
- PostgreSQL, for example, allows columns to contain sub-values, such as arrays of base types and multi-dimensional arrays.



# limitations of the relational model

- 1. Performance Issues:** Dealing with **large datasets or complex joins** between tables can lead to **slow performance**. Optimizing indexing strategies can also be challenging
- 2. Scalability Challenges:** While generally scalable, managing the relational model as the database grows in size can become difficult. **Adding new tables or indexes can be time-consuming**, and **managing relationships between tables can become complex**
- 3. Cost:** Relational databases can be **expensive to license and maintain**, particularly for large-scale deployments. They often require dedicated hardware and specialized software, adding to the cost
- 4. Limited Flexibility:** The relational model is designed to **work with tables** that have predefined structures, making it difficult to work with unstructured or semi-structured data
- 5. Data Redundancy:** In some cases (Denormalization, Poor Design), the relational model can lead to data redundancy, which can impact data integrity and efficiency



# 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



# Structured data example

- The mathematical term “*relation*” specifies a formed set of data held as a table.
- In structured data. all row in a table has the same set of columns.

<b>id</b>	<b>name</b>	<b>age</b>
1	Jim	28
2	Pam	26
3	Michael	42

<b>id</b>	<b>subject</b>	<b>Teacher</b>
1	Languages	John Jones
2	Track	Wally West
3	Swimming	Arthur Curry
4	Computers	Victor Stone

<b>student_id</b>	<b>subject_id</b>	<b>grade</b>
2	1	98
1	2	100
1	4	75
3	3	60
2	4	76
3	2	88



# Semi-structured data

- **Semi-structured** data is information that doesn't consist of Structured data (relational database) but still has some structure to it.
- In *JavaScript Object Notation (JSON)* format. It also includes **key-value** stores and **graph** databases.

```
## Document 1 ##
{
  "customerID": "103248",
  "name": {
    "first": "AAA",
    "last": "BBB"
  },
  "address": {
    "street": "Main Street",
    "number": "101",
    "city": "Acity",
    "state": "NY"
  },
  "ccOnFile": "yes",
  "firstOrder": "02/28/2003"
}
```



## Semi-Structured Data

- **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
  - user1 = {"username": "user1", "email": "user1@email.com", "bio": "Hello, world!", "website": "www.user1.com", "phone number": "+1234567890"}
  - user2 = {"username": "user2", "email": "user2@email.com"}
- **Sparse column** representation: schema has a fixed but large set of attributes, by each tuple may store only a subset
  - user1\_preferences = {"likes\_sports": True, "likes\_cooking": False}
  - user2\_preferences = {"likes\_travel": True}



# Features of Semi-Structured Data Models

- **Multivalued data types:** allow attributes to contain non-atomic values.
  - **Sets, multisets**
    - E.g.,: User 1, set of interests {'Sport', 'Cooking', 'Travel', 'anime', 'jazz'}

UserID	UserName	Email
1	User1	user1@email.com
2	User2	user2@email.com

InterestID	Interest
1	Sports
2	Cooking
3	Travel

UserID	InterestID
1	1
1	3



## (Cont.)Features of Semi-Structured Data Models

- **Multivalued data types:** allow attributes to contain non-atomic values.
  - **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)*
  - In a relational database, you need to have a table for each of which due to normalization
    - **e-commerce sites** often list specifications or details for **each product** that they sell, such as **brand**, **model**, **size**, **color**, and numerous other product-specific details.
    - **specifications form the key**, and the **associated value is stored with the key**.



## (Cont.)Features of Semi-Structured Data Models

- **Key-value map:** Example: a social media application where users can post status updates

RELATIONAL DATABASE:

```
SELECT *
FROM Posts
LEFT JOIN Images ON Posts.post_id = Images.post_id
LEFT JOIN Locations ON Posts.post_id = Locations.post_id
LEFT JOIN Tags ON Posts.post_id = Tags.post_id
LEFT JOIN Reactions ON Posts.post_id = Reactions.post_id
LEFT JOIN Comments ON Posts.post_id = Comments.post_id
WHERE Posts.post_id = 123;
```

JSON:{

```
  "post_id": 123,
  "text": "Hello, world!",
  "images": ["img1.jpg", "img2.jpg"],
  "location": "Paris, France",
  "tags": ["friend1", "friend2"],
  "reactions": {"likes": 100, "loves": 50, "wows": 10},
  "comments": [
    {"user": "friend1", "text": "Great post!", "time": "2022-01-01T10:00:00Z"},
    {"user": "friend2", "text": "Thanks for sharing!", "time": "2022-01-01T11:00:00Z"}
  ]
}
```



# (Cont.) Features of Semi-Structured Data Models

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



# Nested Data Types

- Hierarchical data is common in many applications
  - Many databases support such types as part of their support for object-oriented data
- 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 & store complex data

- Example of JSON data

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

- Types: **integer, real, string, and**

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



# JSON

- JSON is **ubiquitous** in data exchange today
  - Widely used for **web services**
  - Most **modern applications** are architected around web services
- SQL extensions for
  - JSON types for storing JSON data
  - Extracting data from JSON objects using path expressions
    - E.g.  $V \rightarrow ID$ , or  $v.ID$
  - **Generating** JSON from **relational** data
    - E.g. `json.build_object('ID', 12345, 'name', 'Einstein')`
  - Creation of JSON collections using aggregation
    - E.g. `json_agg` **aggregate function** in PostgreSQL
  - **Syntax varies greatly across databases**



# XML

- XML uses tags to mark up text
- E.g.

```
<course>
  <course id> CS-101 </course id>
  <title> Intro. to Computer Science </title>
  <dept name> Comp. Sci. </dept name>
  <credits> 4 </credits>
</course>
```
- Tags make the **data self-documenting**
- Tags can be hierarchical



# Example of Data in XML

- <purchase order>  
    <identifier> P-101 </identifier>  
    <purchaser>  
        <name> Cray Z. Coyote </name>  
        <address> Route 66, Mesa Flats, Arizona 86047, 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>...</item>  
    </itemlist>  
    <total cost> 429.85 </total cost>  
    ....  
    </purchase order>



## XML Cont.

- **XQuery language developed to query nested XML structures**
  - Not widely used currently
- SQL extensions to support XML
  - Store XML data
  - Generate XML data from relational data
  - Extract data from XML data types
    - Path expressions
- See Chapter 30 (online) for more information



# Knowledge Representation

## ■ RDF: Resource Description Format

- is a standard way to make statements about resources.  
An RDF statement consists of three components, referred to as a *triple*:

- 1. Subject** is a resource being described by the triple.
- 2. Predicate** describes the **relationship** between the subject and the object.
- 3. Object** is a **resource** that is related to the **subject**.



# Knowledge Representation

- **RDF: Resource Description Format**
- RDF is a data representation standard based on the entity-relationship mode: **Subject(Entity)**, **Predicate(Attribute)**, **Object(Value)**
  - $(ID, \text{attribute-name}, \text{value})$
  - $(ID1, \text{relationship-name}, ID2)$
  - where ID, ID1 and ID2 are identifiers of entities; entities are also referred to as resources in RDF
  - **Unlike, the E-R model**, the RDF model only **supports binary relationships**, and it does not support more general **n-ary relationships**;
  - Each triple has a unique identifier which is International Resource Identifier (IRI)



# Knowledge Representation-Applications

- 1. Search Engines and Information Retrieval**
- 2. Recommendation Systems**
- 3. Healthcare and Life Sciences**
- 4. Fraud Detection and Financial Services**
- 5. Cybersecurity and Threat Intelligence**
- 6. Supply Chain and Logistics Management**
- 7. Enterprise Knowledge Management**
- 8. Academic and Research Applications**



# An example of a triple

Subject	Predicate	Object
Alireza	hasSpouse	Fatima
Alireza	hasAge	25



# Triple View of RDF Data

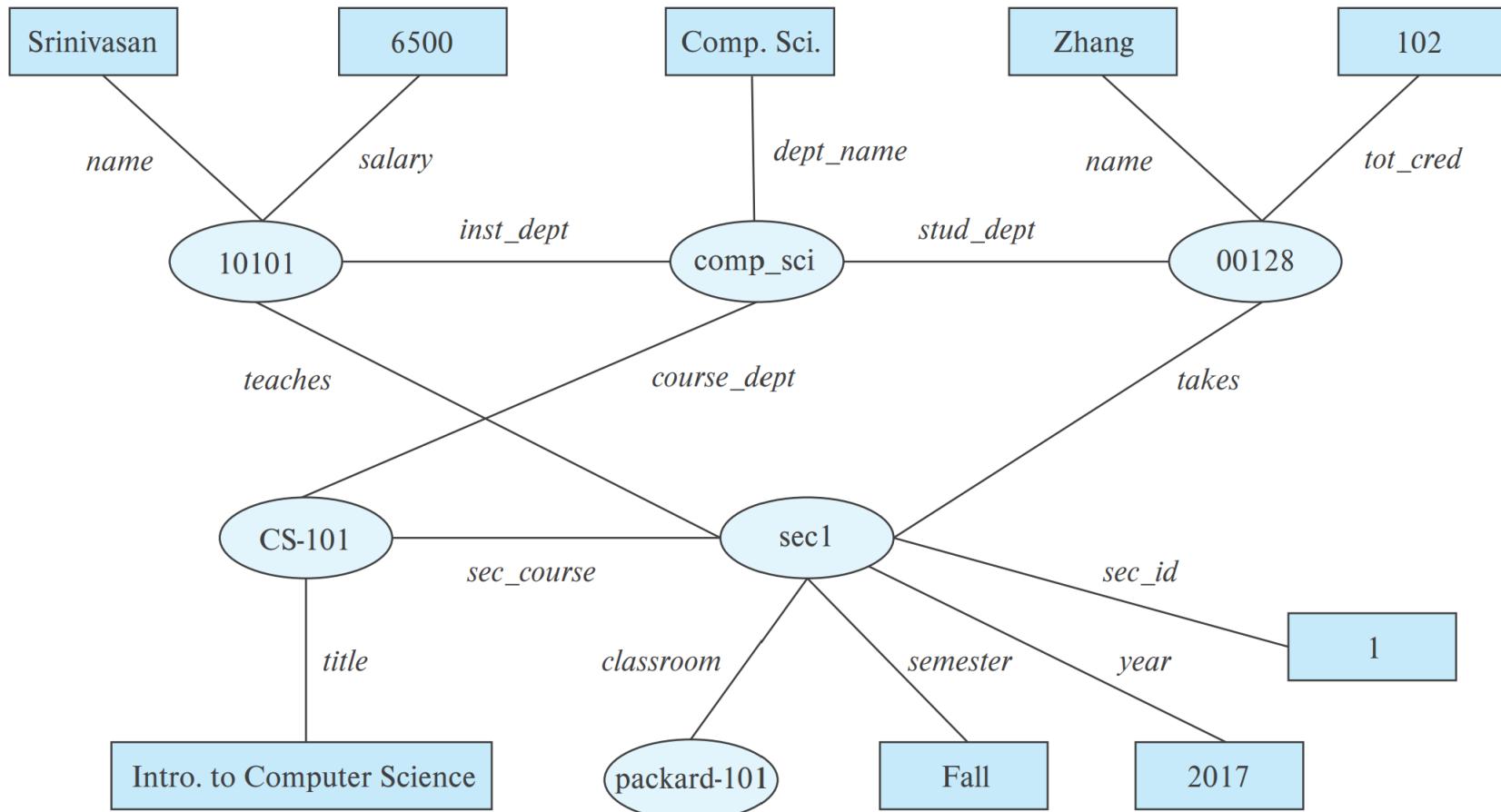
a small part of the University database

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 .



# Graph View of RDF Data

- **Knowledge graph**



**Figure 8.4** Graph representation of RDF data.



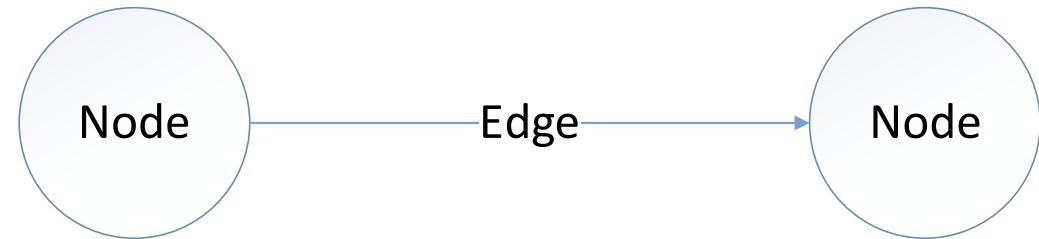
# Knowledge Representation

- Representation of **human knowledge** is a long-standing **goal of AI**
- A representation of **information using the RDF graph model** (or its variants and extensions) is referred to as a knowledge graph.
- **Knowledge graphs** are used for **a variety of purposes**. One such application is to store facts that are harvested from a variety of data sources, such as Wikipedia, Wikidata, and other sources on the web.
- **RDF: Resource Description Format**
  - Simplified representation for facts, represented as triples (*subject, predicate, object*)
    - E.g., (NBA-2019, *winner*, Raptors)  
(Washington-DC, *capital-of*, USA)  
(Washington-DC, *population*, 6,200,000)

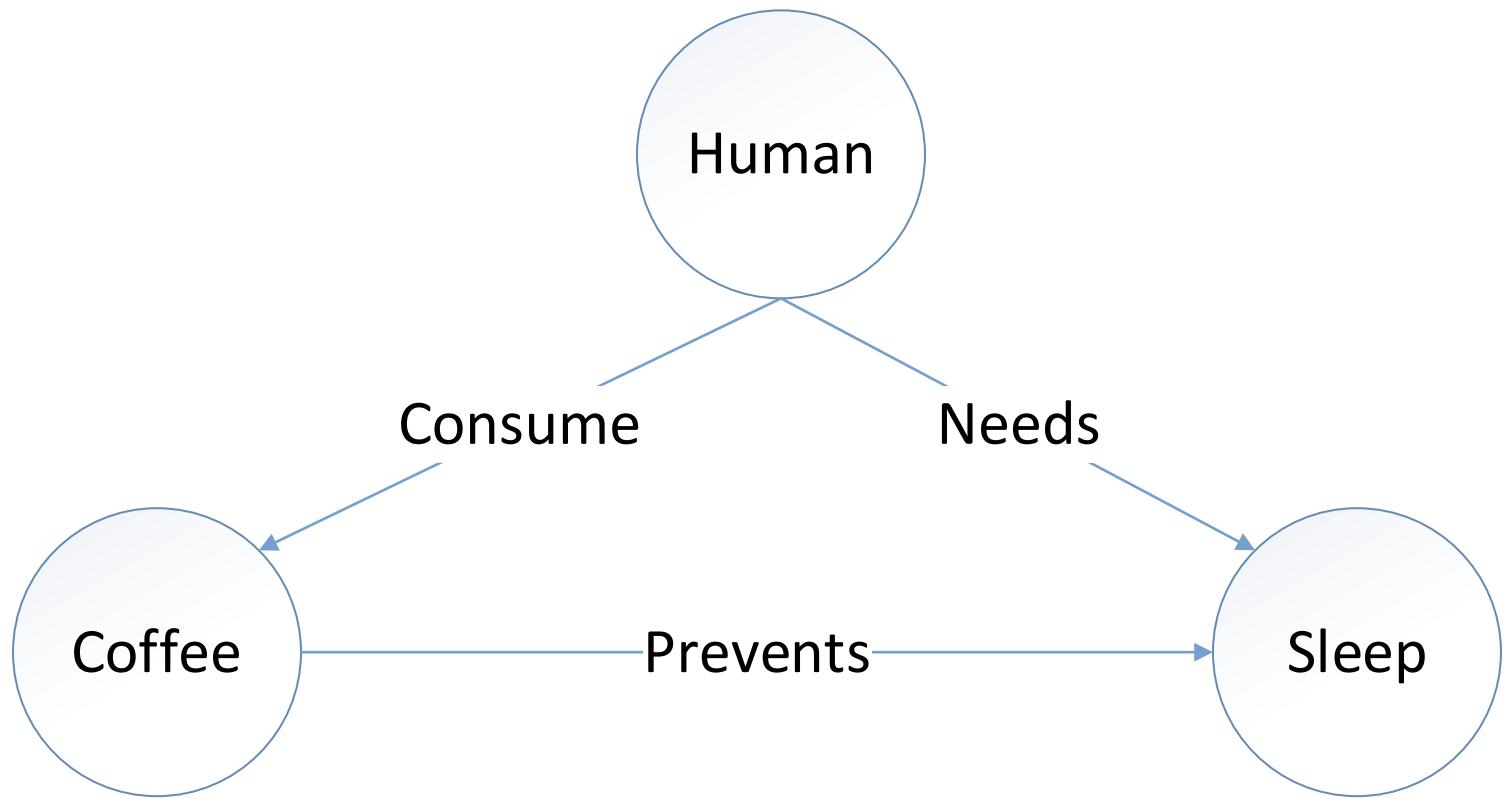


# Knowledge graph

- **Resurgence of interest in Knowledge Graphs.**
  - Search engines
  - Data integration
  - Artificial Intelligence



(580) CS520: 2021  
Knowledge Graphs Seminar  
Session 1 - YouTube





# Data Integration

- For example Data reside in multiple sources.
  - Company directory, product catalog, government database, weather report,
- Answering queries requires combining data from multiple sources.
  - We need to provide translations of data between multiple sources.
    - Direct mappings
    - Shared schema



# Data Integration

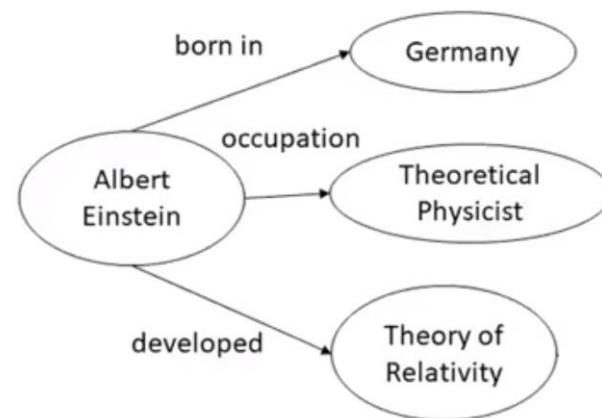
- Schema-free approach to data integration.
  - Convert the relational data from multiple sources into triples.
    - Stored in a graph database.
      - Referred to as a knowledge graph.
    - Deal with schema mappings/translations on "pay as you go" basis.
      - I. Visualization
      - II. Optimized for graph traversals



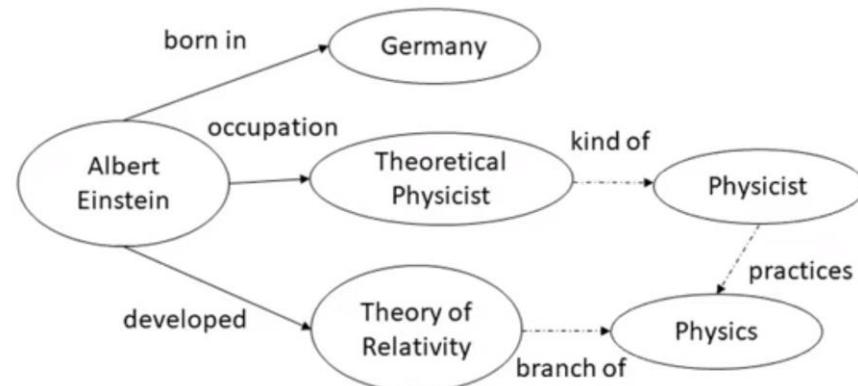
# Knowledge graph: Natural Language Processing

- Entity Extraction:

- Albert Einstein was a German-born theoretical physicist who developed the theory of relativity.
- Relation Extraction->

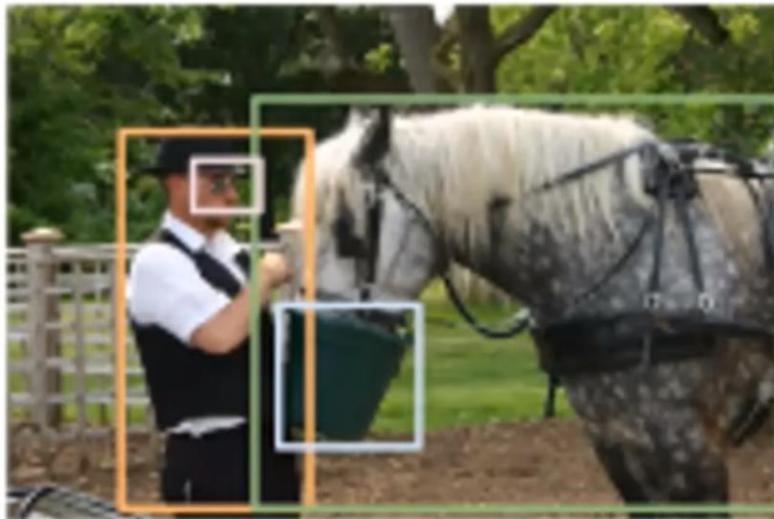


- New relations
  - Questions answering
  - Common reasoning

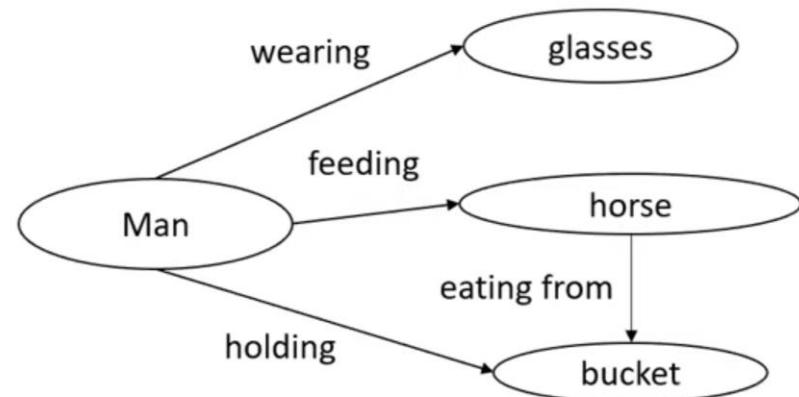


# Knowledge graph in Object detection

## Object Detection



- Edge Detection



# WIKIDATA. KNOWLEDGE

<https://www.wikidata.org/wiki/Q80053>

Item Discussion Read View history Search Wikidata 


Wiki Loves  
FOLKLORE
Photograph your local culture, help Wikipedia and win!

## Tabriz (Q80053)

city in East Azerbaijan Province, Iran

▼ In more languages Configure

English	Tabriz	city in East Azerbaijan Province, Iran	شهر تبریز
Persian		پرجمعیت‌ترین شهر در شمال‌غرب ایران	تبریز
German	Täbris	Hauptstadt von Ost-Aserbaidschan im Iran	Tabris Tabriz Täbris
French	Tabriz	commune iranienne, Azerbaïdjan oriental	Tebriz
Language	Label	Description	Also known as

All entered languages

### Statements

city of Iran	▶ 1 reference
million city	▼ 0 references
instance of	◀ 0 references

image	 Panorama of Tabriz.jpg 2,800 × 1,200; 3.71 MB
official name	(Persian) تبریز ▶ 1 reference

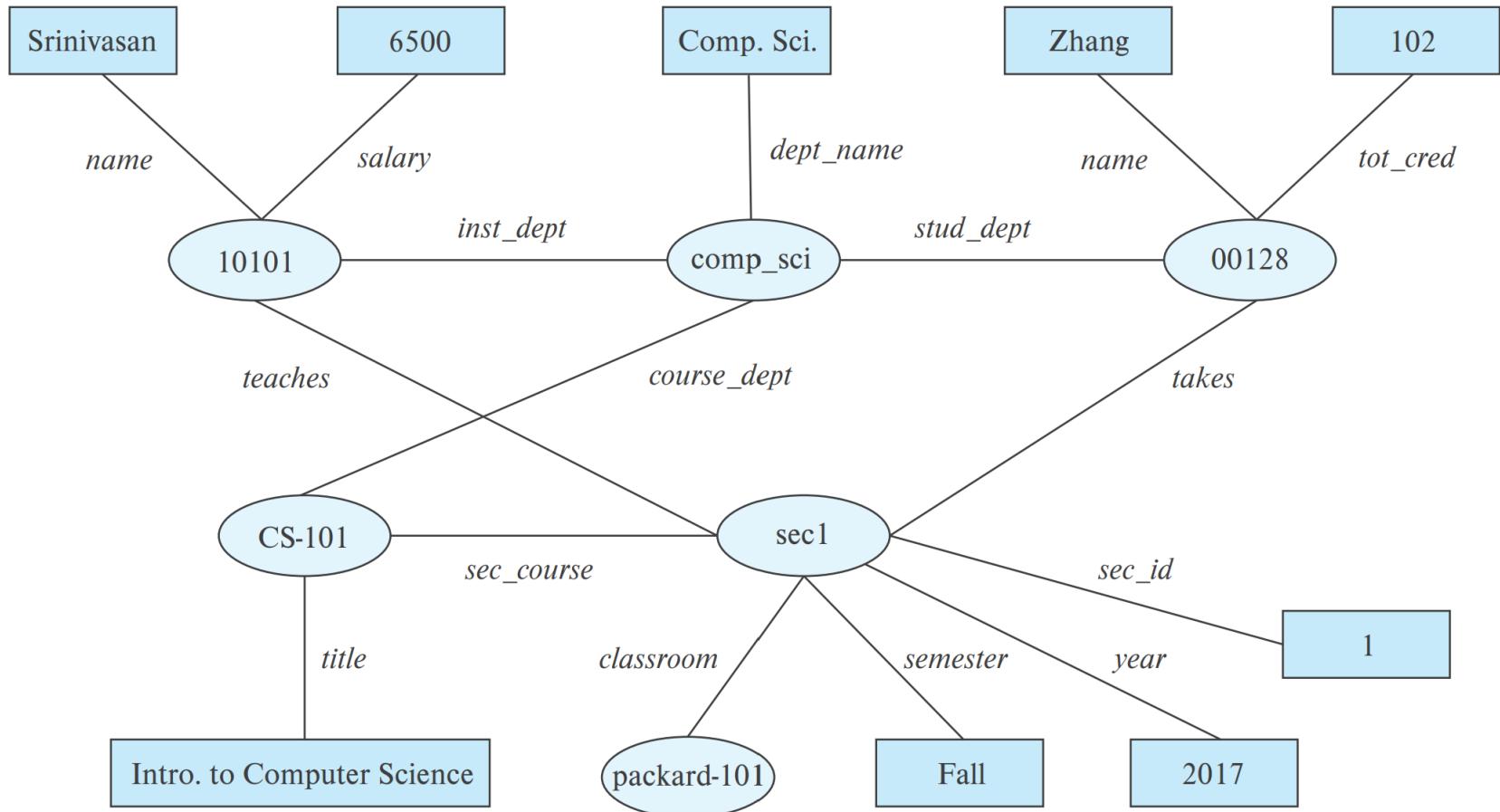


# Querying RDF: SPARQL

- **SPARQL** is a query language **designed to query RDF data**.
  - is based on triple patterns, which look like RDF triples but may contain variables. For example,
- Triple patterns
  - **?cid *title* "Intro. to Computer Science"**
    - match all triples whose predicate is “title” and object is “Intro. to Computer Science”



# University Database



**Figure 8.4** Graph representation of RDF data.



# Querying RDF: SPARQL

- **Another Example of SPARQL:**
  - **?cid title "Intro. to Computer Science"**  
**?sid sec\_course ?cid**
    - On the university-triple dataset, the first triple pattern matches the triple:
      - (CS-101, title, "Intro. to Computer Science"),
    - the second triple pattern matches
      - (sec1, course, CS-101).
    - The **shared variable ?cid enforces a join condition** between the two triple patterns.

---

# What is SPARQL?

**SPARQL is a SQL-like query language for RDF graph data with the following query types:**

- SELECT returns tabular results
- CONSTRUCT creates a new RDF graph based on query results
- ASK returns ‘yes’ if the query has a solution, otherwise ‘no’
- DESCRIBE returns RDF graph data about a resource; useful when the query client does not know the structure of the RDF data in the data source
- INSERT inserts triples into a graph
- DELETE deletes triples from a graph





# Graph View of RDF Data

- Knowledge graph

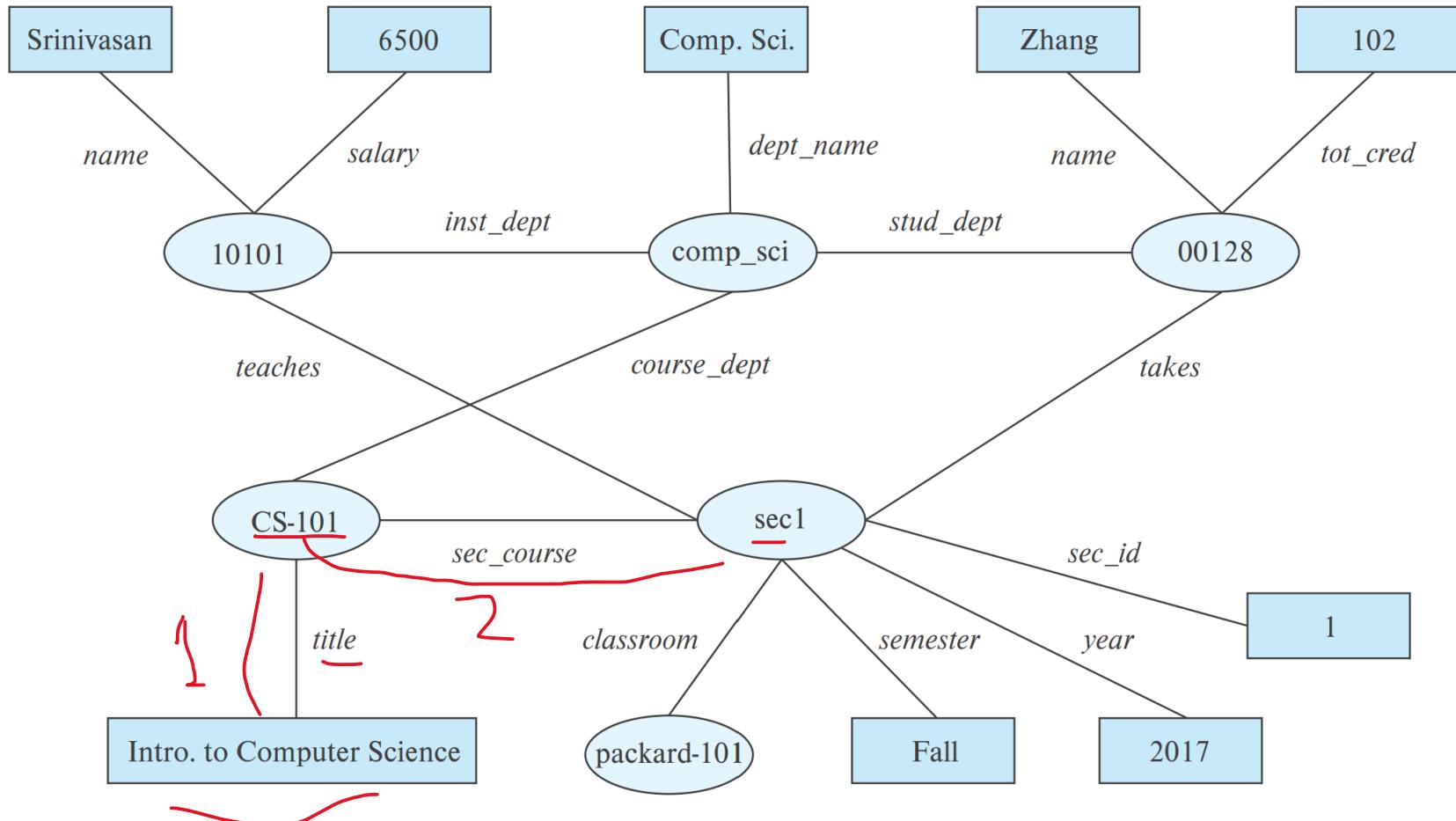


Figure 8.4 Graph representation of RDF data.



# Querying RDF: SPARQL

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

The following query retrieves **names of all students** who have **taken a section** whose **course is titled “Intro. to Computer Science”**.



# Graph View of RDF Data

- Knowledge graph

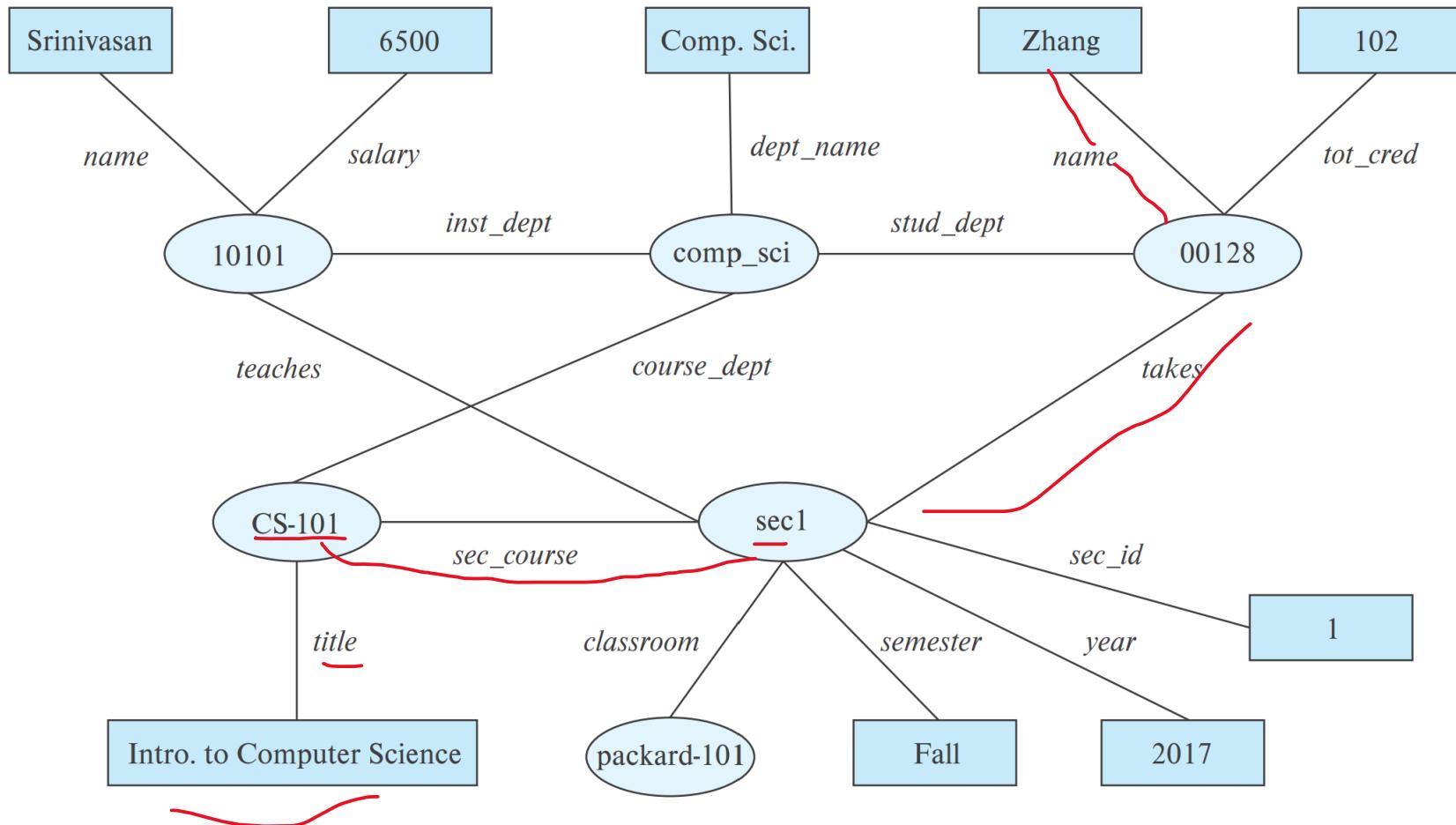


Figure 8.4 Graph representation of RDF data.

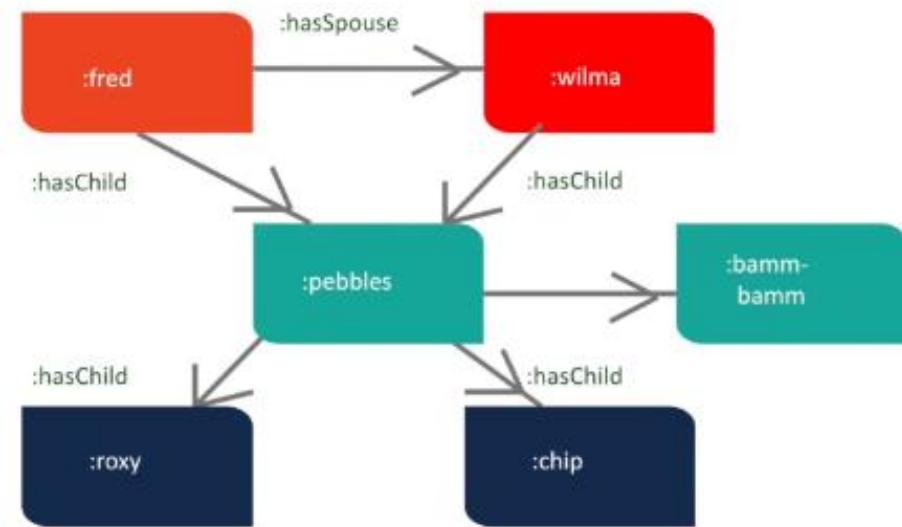
# Using SPARQL to Insert Triples

To create an RDF graph, perform these steps:

- Define prefixes to IRIs with the PREFIX keyword
- Use INSERT DATA to signify you want to insert statements. Write the subject-predicate-object statements (triples).
- Execute this query.

PREFIX br: <http://bedrock/>

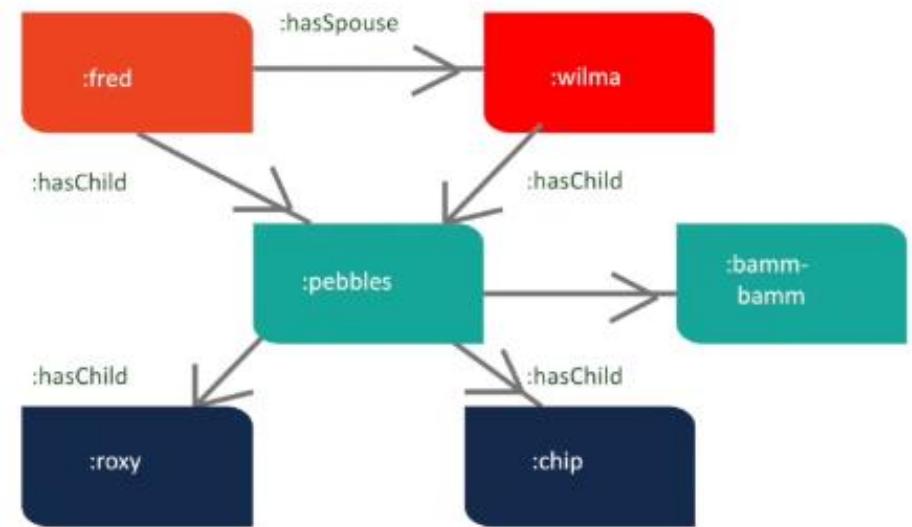
```
INSERT DATA {  
  :fred :hasSpouse :wilma .  
  :fred :hasChild :pebbles .  
  :wilma :hasChild :pebbles .  
  :pebbles :hasSpouse :bamm-bamm ;  
  :hasChild :roxy, :chip .  
}
```



In SPARQL, a **prefix** is a shorthand notation for a **namespace URI** (Uniform Resource Identifier). It simplifies writing and reading URIs in the query. For example:

PREFIX br: <http://bedrock/>

```
INSERT DATA {  
  :fred :hasSpouse :wilma .  
  :fred :hasChild :pebbles .  
  :wilma :hasChild :pebbles .  
  :pebbles :hasSpouse :bamm-bamm ;  
  :hasChild :roxy, :chip .  
}
```



# Using SPARQL to Select Triples

To access the RDF graph you just created, perform these steps:

- Define prefixes to IRIs with the PREFIX keyword.
- Use SELECT to signify you want to select certain information, and WHERE to signify your conditions, restrictions and filters.
- Execute this query.

```
PREFIX br: <http://bedrock/>
SELECT ?subject ?predicate ?object
WHERE {?subject ?predicate ?object}
```

Subject	Predicate	Object
br:fred	br:hasChild	br:pebbles
br:pebbles	br:hasChild	br:roxy
br:pebbles	br:hasChild	br:chip
br:wilma	br:hasChild	br:pebbles

# Using SPARQL to Find Fred's Grandchildren

To find Fred's grandchildren, first find out if Fred has any grandchildren:

- Define prefixes to IRIs with the PREFIX keyword
- Use ASK to discover whether Fred has a grandchild, and WHERE to signify your conditions.

```
PREFIX br: <http://bedrock/>
ASK
WHERE {
    br:fred  br:hasChild  ?child .
    ?child  br:hasChild  ?grandChild .
}
```

→ YES

## Using SPARQL to Find Fred's Grandchildren

Now that we know he has at least one grandchild, perform these steps to find the grandchild(ren):

- Define prefixes to IRIs with the PREFIX keyword
- Use SELECT to signify you want to select a grandchild, and WHERE to signify your conditions.

```
PREFIX br: <http://bedrock/>
SELECT ?grandChild
WHERE {
    br:fred    br:hasChild      ?child .
    ?child     br:hasChild      ?grandChild .
}
```

grandChild

1. br:roxy
2. br:chip



## RDF Representation (Cont.)

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



## RDF Representation (Cont.)

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



# RDF DATA STORES:

The slide displays a grid of logos for various RDF data stores. The logos are arranged in several rows:

- Row 1:** YarcData (A CRAY COMPANY), FRANZ INC., DYDRA
- Row 2:** MarkLogic, Oracle Labs NOSQL DATABASE, PGX, ORACLE DATABASE, 12c
- Row 3:** OPENLINK VIRTUOSO, Jena, rdf4j
- Row 4:** neo4j, azegraph, ontotext, IBM DB2.
- Row 5:** CumulusRDF (partially visible on the left)

On the far left, there is a vertical column of icons representing different RDF data stores, though they are not all fully visible or labeled.



# Object Orientation

- **Object-relational data model** provides richer type system
  - with complex data types and object orientation
- Applications are often written in **object-oriented programming languages**
  - Type system does not match relational type system
  - Switching between imperative language and SQL is troublesome



# Object Orientation

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



# Object-Oriented Programming and Databases

- **Point 1:** Many database **applications are written using an object-oriented** programming language.
  - Examples: Java, Python, C++
- **Point 2:** These **applications need to store and fetch data from databases.**
- **Point 3:** There is a type difference between the native type system of the object-oriented programming language and the relational model supported by databases.
- **Point 4:** Data need to be translated between the two models whenever they are fetched or stored.



# Object-Relational Database Systems

- SQL allows creation of structured user-defined types:
  - **create type Person**  
*(ID varchar(20) primary key,  
name varchar(20),  
address varchar(20)) ref from(/D);*
  - **create table people of Person;**
- Then we create a new person as follows:
  - **insert into people (ID, name, address) values ('12345', 'Srinivasan', '23 Coyote Run')**



# Object-Relational Database Systems

- Table user types
  - **create type interest as table (**  
**topic varchar(20),**  
**degree\_of\_interest int);**
  - SQL Server allows table-valued types(e.g., interest type) to be declared as shown in the following example:  
**create table users (**  
**ID varchar(20),**  
**name varchar(20),**  
**interests interest);**
- Array, multiset data types also supported by many databases



# Type and Table Inheritance

- Type inheritance
  - **create type Person**  
*(ID varchar(20) primary key,  
name varchar(20),  
address varchar(20)) ref from(ID);*
  - We want to store extra information in the database about people who are students
  - **create type Student under Person**  
*(degree varchar(20)) ;*  
**create type Teacher under Person**  
*(salary integer);*
  - Both Student and Teacher inherit the attributes of Person



# Type and Table Inheritance

- Table inheritance syntax in PostgreSQL and oracle
  - **create table** *students*  
*(degree varchar(20))*  
**inherits** *people*;
  - **create table** *teachers*  
*(salary integer)*  
**inherits** *people*;
  - **create table** *people of Person*;  
**create table** *students of Student*  
**under** *people*;
  - **create table** *teachers of Teacher*  
**under** *people*;
- As a result, every attribute present in the table *people* is also present in the subtables *students* and *teachers*.



# Reference Types

- For example, we could define the Person type as follows, with a reference-type declaration:
- Creating reference types
  - **create type Person**  
**(*ID* varchar(20) primary key,**  
**name varchar(20),**  
**address varchar(20))**  
**ref from(*ID*);**  
**create table people of Person;**



# Reference Types

- a type Department with a field name and a field head that is a reference to the type Person.
- ```
create type Department (
    dept_name varchar(20),
    head ref(Person) scope people);
```

  
**create table** departments of *Department*  
  
**insert into** departments **values** ('CS', '12345')
- scope clause above completes **the definition of the foreign key** from departments.head to the people relation.



# Object-Relational Mapping

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



## ORM benefits

- Simplify the job of **developers** by providing an **object model**, while still leveraging the power of a robust relational database.
- ORM systems can offer significant **performance improvements** when **operating on objects cached in memory**, compared to direct access to the underlying database.
- ORM can **use any number of databases** to store data, **all with the same high-level code**.
- ORM systems **abstract away minor SQL differences** between databases. This makes migration from one database to another relatively straightforward when using an ORM, as opposed to the significant challenges posed by SQL differences.



# ORM EXAMPLE

Imagine you're building an online store. You need to manage data like products, customers, and orders.

## Without ORM:

- You'd **write complex SQL queries** to interact with the database: **add new products, update order details, etc.**
- You'd need to **understand the specific syntax for your chosen database** (MySQL, PostgreSQL, etc.).
- **Switching databases would require rewriting most or all of your SQL code**, due to different syntax and functionalities.

## With ORM:

- You'd **define your data structures as objects (Product, Customer, Order)**.
- You'd use simple, object-oriented methods to interact with your data:
  - `product.save()`, `order.update_status()`.
- The ORM system translates these object operations into the appropriate SQL queries
- Switching databases might involve some configuration changes within the ORM, but most of your code would remain the same.



# Performance issue of ORM

- We have a User table in a database and we are using an ORM system in our application.
- we want to update the status of all users to inactive.

```
users = session.query(User).all()
for user in users:
    user.status = 'inactive'
    session.commit()
```

- This result in a **separate UPDATE statement for each user**, which could be very inefficient if there are a large number of users.
- Alternative where we bypass the ORM and write the update directly in SQL

```
session.execute("UPDATE users SET status='inactive'")
session.commit()
```

- Executing a single UPDATE statement for all users is significantly more efficient than issuing individual statements for each user.

# Textual Data

---

- **Information Retrieval Definition:**
  - Information retrieval is the process of querying unstructured textual data. Which is used in
  - **Traditional Model:** Textual information is organized into documents.
  - **Database Context:** A text-valued attribute can be considered a document.
  - **Web Context:** Each web page can be considered a document.

## Textual Data (Cont.)

---

- **Keyword Description:** Desired **documents** are typically described by a set of **keywords**.
- **Examples:** **Keywords** such as “**database system**” can locate **documents** on **database** systems. “**Stock**” and “**scandal**” can locate **articles** about **stock-market scandals**.
- **Document Keywords:** Documents have a set of keywords associated with them. Typically, all **words in the documents are considered keywords**.
- **Keyword Query:** A keyword **query retrieves documents whose set of keywords contains all the keywords in the query**.

# Textual Data

---

- **Information retrieval:** querying of unstructured data
  - **Simple model of keyword queries:** given query keywords, retrieve documents containing all the keywords
  - More advanced models **rank relevance of documents**
  - Today, keyword queries return many types of information as answers
    - E.g., a query “cricket” typically returns information about ongoing cricket matches like scores, league table
- Relevance ranking
  - Essential since there are usually many documents matching keywords



# Keyword Search and Information Retrieval

- **Keyword Search**
  - Initially **targeted at document repositories** within **organizations** or **domain-specific document repositories** such as research publications.
  - Now, also important for **documents stored** in a database.
- **Keyword-based Information Retrieval**
  - Used for **retrieving not only textual data**, but also **other types of data**.
  - **Video and audio data that have descriptive keywords** associated with them can be retrieved.
  - Example: A **video movie may have keywords** such as its **title, director, actors**, and **genre**. An image or **video** clip may have **tags**, which are **keywords** describing the image or video clip.



# Keyword Search and Information Retrieval

- ## Web Search Engines

- At the core, they are information retrieval systems.
- They **retrieve and store web pages by crawling the web.**

# Relevance Ranking

---

- **Document Set Size**
    - The set of all documents that contain the keywords in a query may be very large.
    - There are billions of documents on the web, and most keyword queries on a web search engine find hundreds of thousands of documents containing some or all of the keywords.
  - **Relevance of Documents**
    - Not all the documents are equally relevant to a keyword query.
    - Information-retrieval systems estimate the relevance of documents to a query and return only highly ranked documents as answers.
- 71



# Ranking using TF-IDF

- Term: **keyword occurring** in a document/query
- **Term Frequency:**  $TF(d, t)$ , **the relevance of a term  $t$  to a document  $d$** 
  - One definition:
  - where
    - $n(d,t)$  = number of **occurrences** of term  $t$  in document  $d$
    - $n(d)$  = **number of terms** in **document  $d$**
    - takes the length of the document into account.
    - The relevance grows with **more occurrences of a term in the document**.



# Query and Keyword Relevance

- **Multiple Keywords**
  - A query  $Q$  may contain multiple keywords.
  - The relevance of a document to a query with two or more keywords is estimated by combining the **relevance measures of the document for each keyword**.
- **Combining Measures**
  - A **simple way** of combining the measures is to **add them up**.



# Query and Keyword Relevance

- **Keyword Frequency**
  - Not all terms used as **keywords** are **equal**.
  - Suppose a query uses two terms, one of which occurs frequently, such as “**database**”, and another that is less frequent, such as “**Silberschatz**”.
  - A document containing “**Silberschatz**” but not “**database**” should be ranked higher than a document containing the term “**database**” but not “**Silberschatz**”.



# Ranking using TF-IDF

- **Inverse document frequency:**  $IDF(t)$ 
  - One definition:
  - $IDF(t) = \frac{1}{n(t)}$
  - $n(t)$  denotes the number of documents (among those indexed by the system) that contain the term  $t$
- **Relevance** of a document  $d$  to a set of terms  $Q$ 
  - *One definition:*  $r(d, Q) = \sum_{t \in Q} TF(d, t) * IDF(t)$
  - Other definitions
    - take **proximity** of words into account
    - **Stop words** are often ignored

# Example for Inverse Document Frequency (IDF)

---

- Imagine a **collection** with **three documents** (A, B, C) and **two keywords** ("mushroom" and "poisonous mushroom"):
- \* Document A: "**Mushrooms** are living organisms that..."
- \* Document B: "There are many types of **mushrooms**, some edible and others **poisonous**."
- \* Document C: "**Poisonous mushrooms** can be very dangerous."
- **Calculating IDF:**
- \* Keyword "mushroom": ?
- \* Keyword "poisonous mushroom": ?

## Example for IDF (Cont.)

---

- **Calculating IDF:**
- \* Keyword "mushroom":
- \* Number of documents containing "mushroom" ( $n(\text{mushroom})$ ) = 3
- \*  $\text{IDF}(\text{mushroom}) = 1 / 3 \approx 0.33$
- \* Keyword "poisonous mushroom":
- \* Number of documents containing "poisonous mushroom" ( $n(\text{"poisonous mushroom"})$ ) = 2
- \*  $\text{IDF}(\text{"poisonous mushroom"}) = 1 / 2 \approx 0.50$

## Example of document relevance calculation

---

### Scenario:

Imagine a collection with three documents (A, B, C) and two keywords ("music" and "concert"):

- **Document A:** "I enjoy listening to music." (Mentions "music" once)
- **Document B:** "Last weekend, I went to a music concert." (Mentions both "music" and "concert" once)
- **Document C:** "Did you know the speed of sound varies depending on the medium?" (Mentions neither "music" nor "concert")

## Example of document relevance calculation (Cont.)

---

**Calculating Relevance ( $r(d, Q)$ ) for the query "music concert":**

### 1. Document A:

- **TF(A, "music") = 1**: Document A mentions "music" once.
- **TF(A, "concert") = 0**: Document A doesn't mention "concert".
- **IDF("music") = 0.2 (assumed value)**: Let's assume "music" is a common term with a lower IDF.
- **IDF("concert") = 0.8 (assumed value)**: Let's assume "concert" is a less common term with a higher IDF.
- **$r(A, "music concert") = TF(A, "music") * IDF("music") + TF(A, "concert") * IDF("concert") = 1 * 0.2 + 0 * 0.8 = 0.2$**

## Example of document relevance calculation (Cont.)

---

**Calculating Relevance ( $r(d, Q)$ ) for the query "music concert":**

**2. Document B:**

- $TF(B, \text{"music"}) = 1$ : Document B mentions "music" once.
- $TF(B, \text{"concert"}) = 1$ : Document B mentions "concert" once.
- $IDF(\text{"music"}) = 0.2$  (assumed value): Consistent with Document A.
- $IDF(\text{"concert"}) = 0.8$  (assumed value): Consistent with Document A.
- $r(B, \text{"music concert"}) = TF(B, \text{"music"}) * IDF(\text{"music"}) + TF(B, \text{"concert"}) * IDF(\text{"concert"}) = 1 * 0.2 + 1 * 0.8 = 1.0$

## Example of document relevance calculation (Cont.)

---

### 3. Document C:

- $\text{TF}(C, \text{"music"}) = 0$ : Document C doesn't mention "music".
- $\text{TF}(C, \text{"concert"}) = 0$ : Document C doesn't mention "concert".
- $\text{IDF}(\text{"music"}) = 0.2 \text{ (assumed value)}$ : Consistent with Documents A and B.
- $\text{IDF}(\text{"concert"}) = 0.8 \text{ (assumed value)}$ : Consistent with Documents A and B.
- $r(C, \text{"music concert"}) = \text{TF}(C, \text{"music"}) * \text{IDF}(\text{"music"}) + \text{TF}(C, \text{"concert"}) * \text{IDF}(\text{"concert"}) = 0 * 0.2 + 0 * 0.8 = 0$



# Stop Words in Information Retrieval

- **Information-retrieval** systems define a **set of words**, called **stop words**, **containing 100** or so of the most common words, and ignore these words when indexing a document.
  - Such words are **not used as keywords**, and **they are discarded** if present in the keywords supplied by the user.
    - and,” “or,” “a,” and so on.
  - **Including them** in the indexing process and user queries can **increase processing time** and **potentially distract** the system from identifying more informative keywords.



# Proximity of Terms

- **Proximity of Terms**
  - Another factor taken into account when a query contains multiple terms is the proximity of the terms in the document.
  - If the **terms occur close to each other** in the document, the **document will be ranked higher than if they occur far apart**.
  - The formula for  $r(d, Q)$  can be modified to take **proximity** of the terms into account.



# Ranking Using Hyperlinks

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



# Ranking Using Hyperlinks

- Let  $T[i, j]$  be the **probability** that a **random walker** who is on **page  $i$**  **will click on the link to page  $j$** 
  - Assuming all links from  $i$  has an equal **probability of being followed**:  $T[i, j] = \frac{1}{N_i}$
  - $N_i$ : number of **outgoing links from Page  $i$**
- Then **PageRank[j]** as  $P[j]$  for each page  $j$  can be defined as  $P[j] = \frac{\delta}{N} + (1 - \delta) * \sum_{i=1}^N (T[i, j] * P[i])$ 
  - $N$  = total number of pages
  - $\delta$  = a constant usually set to **0.15**



# Ranking Using Hyperlinks

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

## Example

a simple web network with four web pages: A, B, C, D

$$T = \begin{pmatrix} 0 & 0.5 & 0.5 & 0.5 \\ 0.5 & 0 & 0.5 & 0 \\ 0.5 & 0.5 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

- an initial PageRank vector  $P$  where each page has an initial score of  $0.25 = 1/N = 1/4$ , so:

$$P[j] = \frac{\delta}{N} + (1 - \delta) * \sum_{i=1}^N (T[i,j] \cdot P[i])$$

$$P = \begin{pmatrix} 0.25 \\ 0.25 \\ 0.25 \\ 0.25 \end{pmatrix}$$

- Let's calculate  $P[j]$  for each page:

- $P[A] = 0.15/4 + (1 - 0.15) \times [(0 \times 0.25) + (0.5 \times 0.25) + (0.5 \times 0.25) + (0.5 \times 0.25)] = 0.35625$
- After one iteration of the PageRank algorithm, the PageRank scores are approximately.



# Retrieval Effectiveness

- Measures of effectiveness
  - **Precision:** what percentage of returned results are relevant
  - **Recall:** what percentage of relevant results were returned
    - since search engines find a very large number of answers, **precision and recall numbers are usually measured by “@K”**, where K is the number of answers viewed



# Spatial Data



# Spatial Data

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

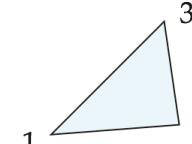
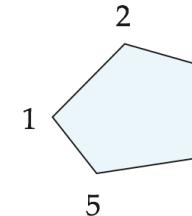
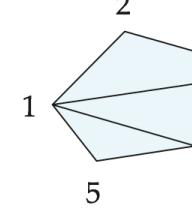


# Represented of Geometric Information

- Various geometric constructs can be represented in a database in a normalized fashion (see next slide)
- A **line segment** can be represented by the coordinates of its endpoints.
- A **polyline** or **linestring** consists of a connected sequence of line segments and can be represented by a list containing the coordinates of the endpoints of the segments, in sequence.
- **Polygons** are represented by a list of vertices in order.
  - The list of vertices specifies the boundary of a polygonal region.
  - Can also be represented as a set of triangles (**triangulation**)



# Representation of Geometric Constructs

| object       | representation                                                                                                                                                                                                                                             |
|--------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| line segment | <br>$\{(x_1, y_1), (x_2, y_2)\}$                                                                                                                                          |
| triangle     | <br>$\{(x_1, y_1), (x_2, y_2), (x_3, y_3)\}$                                                                                                                              |
| polygon      | <br>$\{(x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4), (x_5, y_5)\}$                                                                                                      |
| polygon      | <br>$\{(x_1, y_1), (x_2, y_2), (x_3, y_3), \text{ID1}\}$<br>$\{(x_1, y_1), (x_3, y_3), (x_4, y_4), \text{ID1}\}$<br>$\{(x_1, y_1), (x_4, y_4), (x_5, y_5), \text{ID1}\}$ |



## For example

- SQL Server and PostGIS support the geometry and geography types
  - **subtypes** such as **point**, **linestring**, **curve**, **polygon**, as collections of these types called multipoint, multilinestring, multicurve and multipolygon.
- Textual representations of these types are defined by the OGC standards, and can be converted to internal representations using conversion functions.
  - For example,
    - **LINESTRING**(1 1, 2 3, 4 4) defines a line that connects points (1, 1), (2, 3) and (4, 4),
    - **POLYGON**((1 1, 2 3, 4 4, 1 1)) defines a triangle defined by these points.



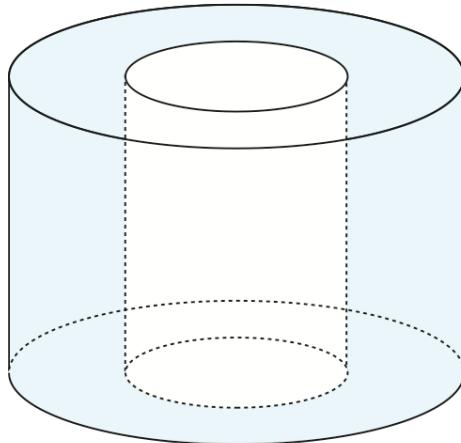
# To Design Databases

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

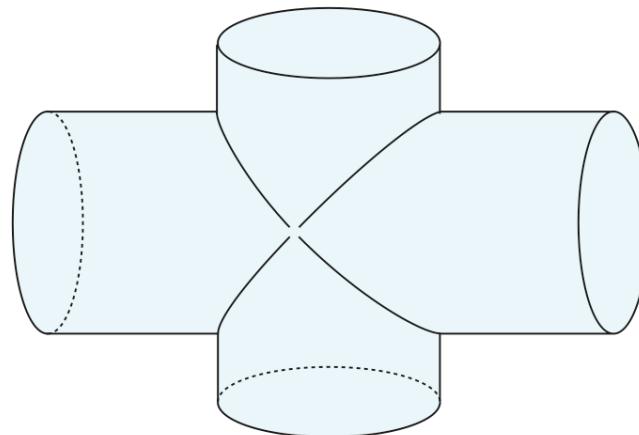


# Representation of Geometric Constructs

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



(a) Difference of cylinders



(b) Union of cylinders



# Geographic Data

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



# Geographic Data (Cont.)

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



## Spatial Queries

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



# End of Lecture 1