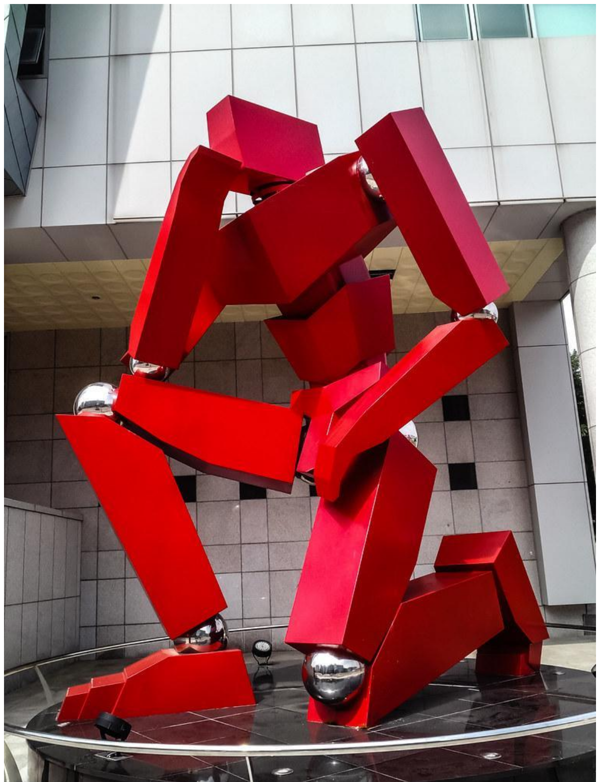
Difficulty level:
basic

Data Representation and Modeling

OpenDS4All

Thinking More Deeply about Data and Computation



"Thinking Robot" by purunuri is licensed under CC BY-NC-SA 2.0

We've seen:

- **semi-structured** HTML and **unstructured** text, represented using tables to be used for visualization and learning
- **manipulating tabular data**
  - **projection** (subsetting fields), **selection** (choosing rows meeting predicates), **loc** (extract or update cell), **apply** (compute function over each row/col/cell)
- linking tabular data
  - **merge**/join, outerjoin, and using string similarity to join

Now let's dive into more detail on **design**:

- How do we encode data? What are the implications?

A First Question:
What Are We Trying to Capture?

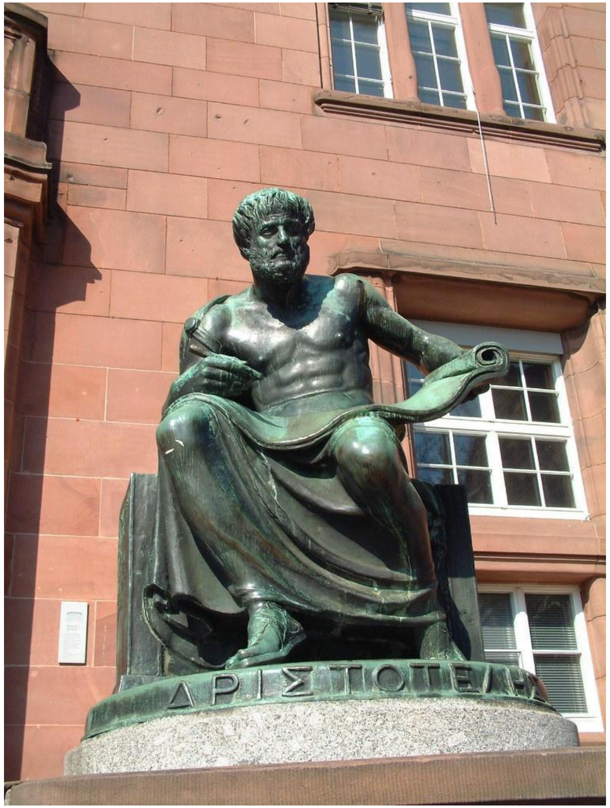
"Structured data should capture the semantics of the data"

What do we mean by "data semantics"?

This is a topic that has preoccupied philosophers since *at least* Aristotle and Plato

… and computer scientists for most of the lifetime of the field!

Part of the Goal: Modeling Concepts and Instances



The famous example from logic and philosophy, attributed to Aristotle:

- All men are mortal.
- Socrates is a man.
- Therefore, Socrates is mortal.

The premise: we have *concepts* which are classes of things, and *instances* of those concepts

- *Properties* of the concepts appear in the instances
- Instances *relate* to other instances

Data design is about trying to codify the above!

Some Starting Points

We model knowledge using notions dating back to ancient Greece:

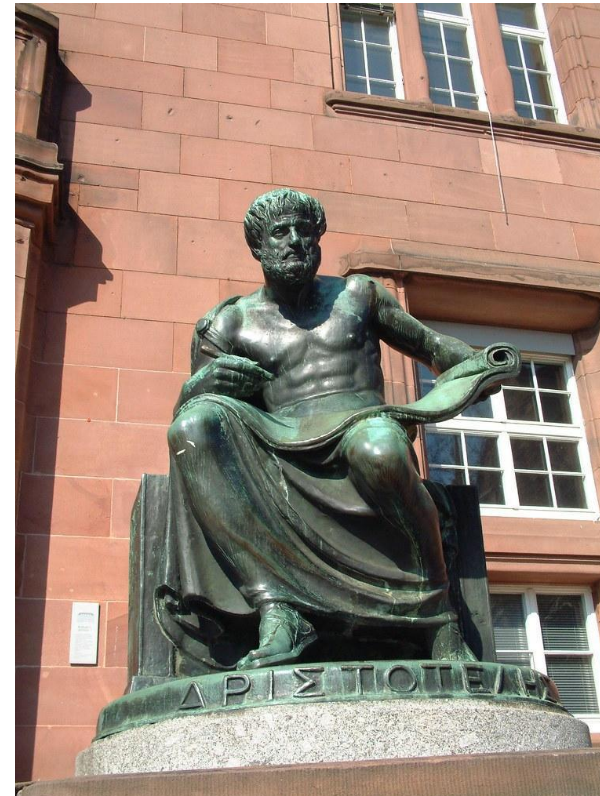
- **Classes**, concepts, or sets of entities – e.g., people
- **Instances** of those classes – e.g., Socrates, Aristotle, Plato
- Named **relationships** between classes – e.g., **people** have **teachers** who are other people (thus Aristotle has a teacher, namely Plato)
- Classes may also have **properties**, e.g., people have names or are mortal

There are different, equivalent ways of looking at these!

- Using **logic** – "knowledge representation," a key idea in AI
- Using **knowledge graphs** – named relationships between classes, subclasses, instances, properties
- Using **entity-relationship modeling** – a special case of knowledge graphs

- These can all be used to inform our design of dataframes, hierarchical data, etc.

Modeling Classes, Instances, Properties Using Logical *Predicates*



We can use **logical assertions** to describe everything.

Classes: named, categorized collections of items

“All people are mortal” : Mortal(person).

Classes have specializations or subclasses:

“Men are people” : Subclass(man, person).

Classes have instances:

“Aristotle is a man” : Instance(Aristotle, man)

And we **infer** predicates from class to subclass, or class to instance, using ***rules***:

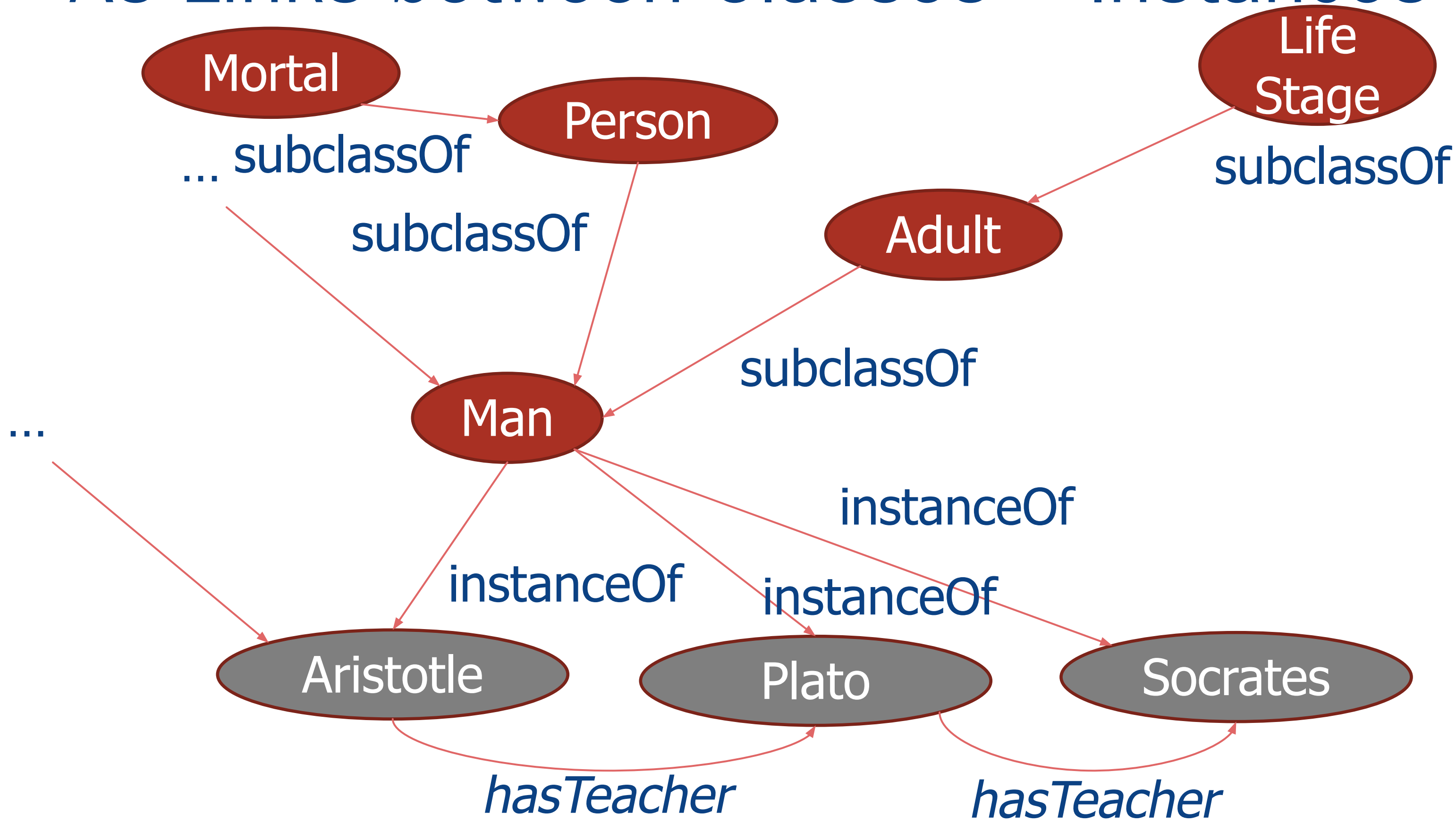
Mortal(x) ^ Subclass(y, x) □ Mortal(y)

Mortal(x) ^ Instance(y, x) □ Mortal(x)

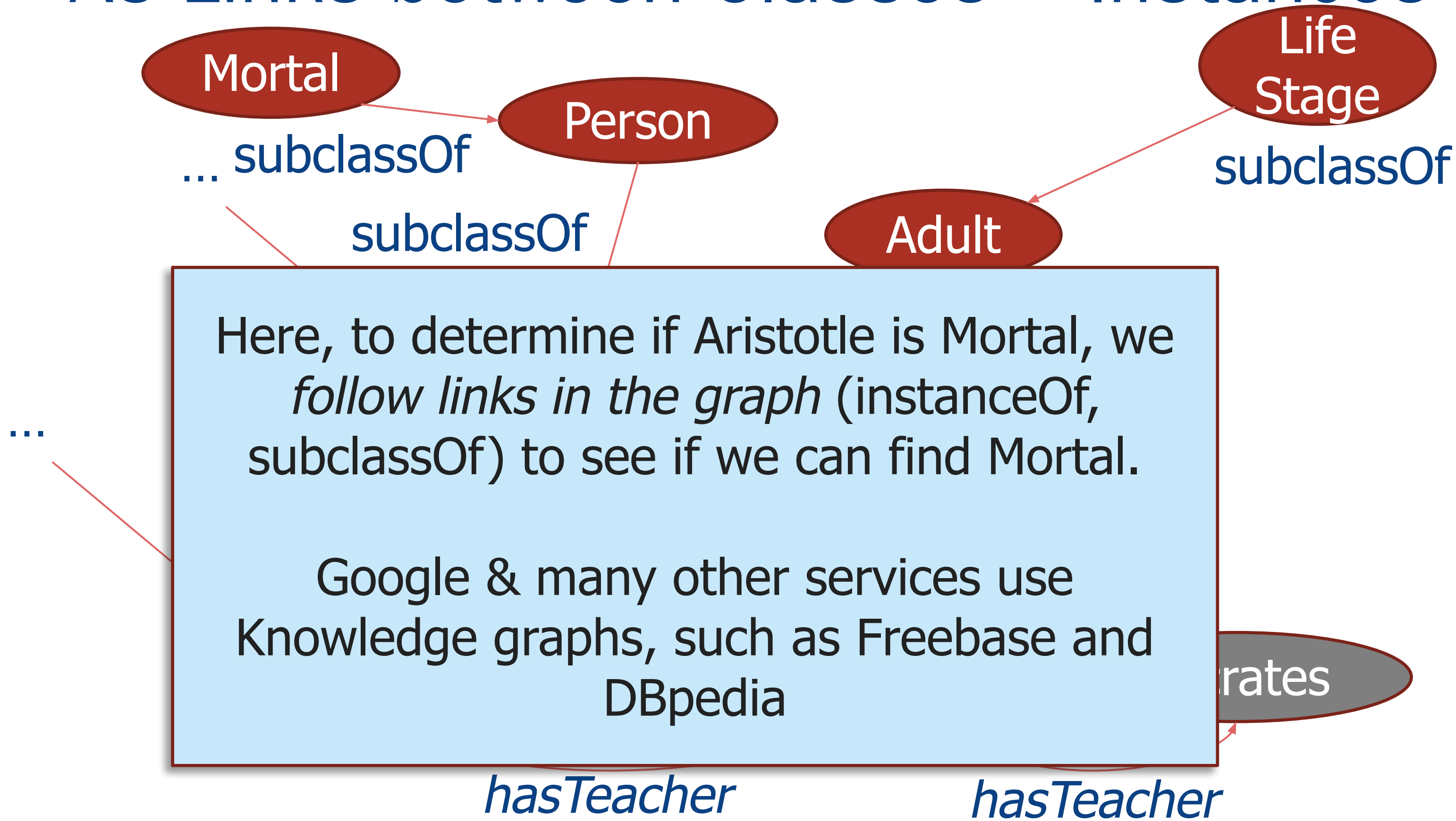
Mortal(person) ^ Subclass(man, person) □ Mortal(man)

Mortal(man) ^ Instance(Aristotle, man) □ Mortal(Aristotle)

We Can Instead Think of this As Links between Classes + Instances



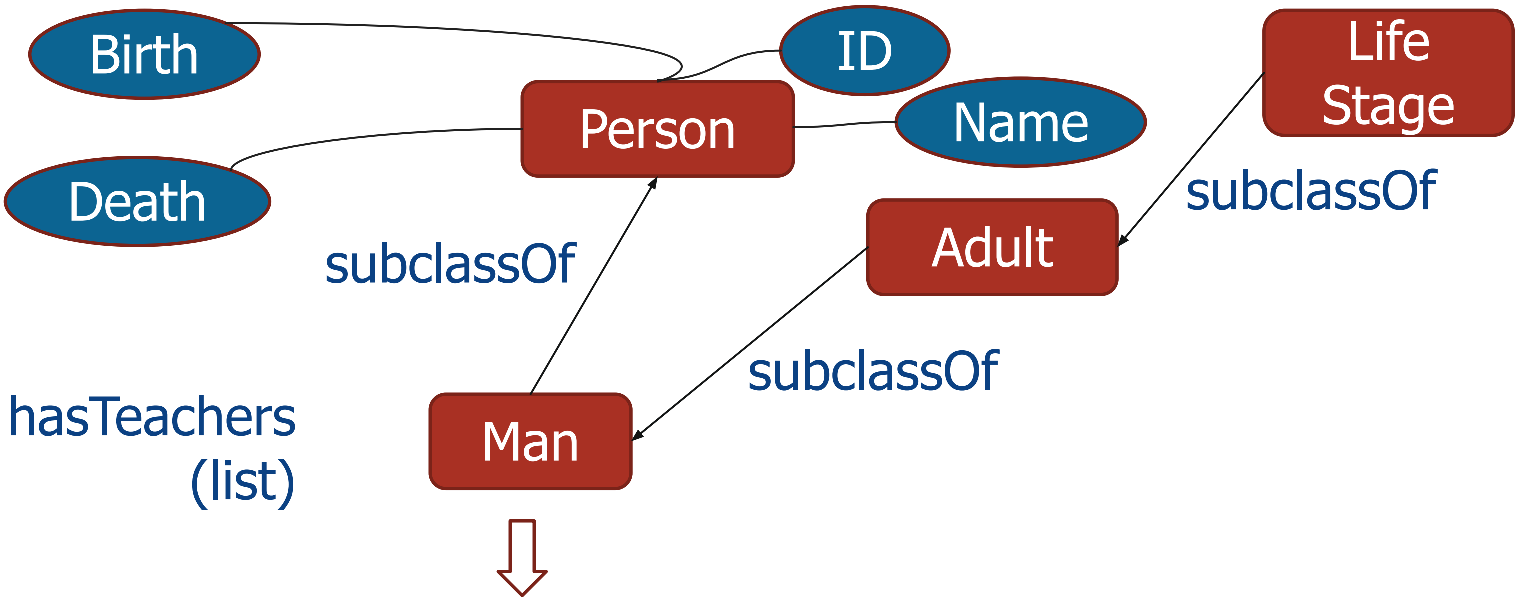
We Can Instead Think of this As Links between Classes + Instances



Here, to determine if Aristotle is Mortal, we *follow links in the graph* (instanceOf, subclassOf) to see if we can find Mortal.

Google & many other services use Knowledge graphs, such as Freebase and DBpedia

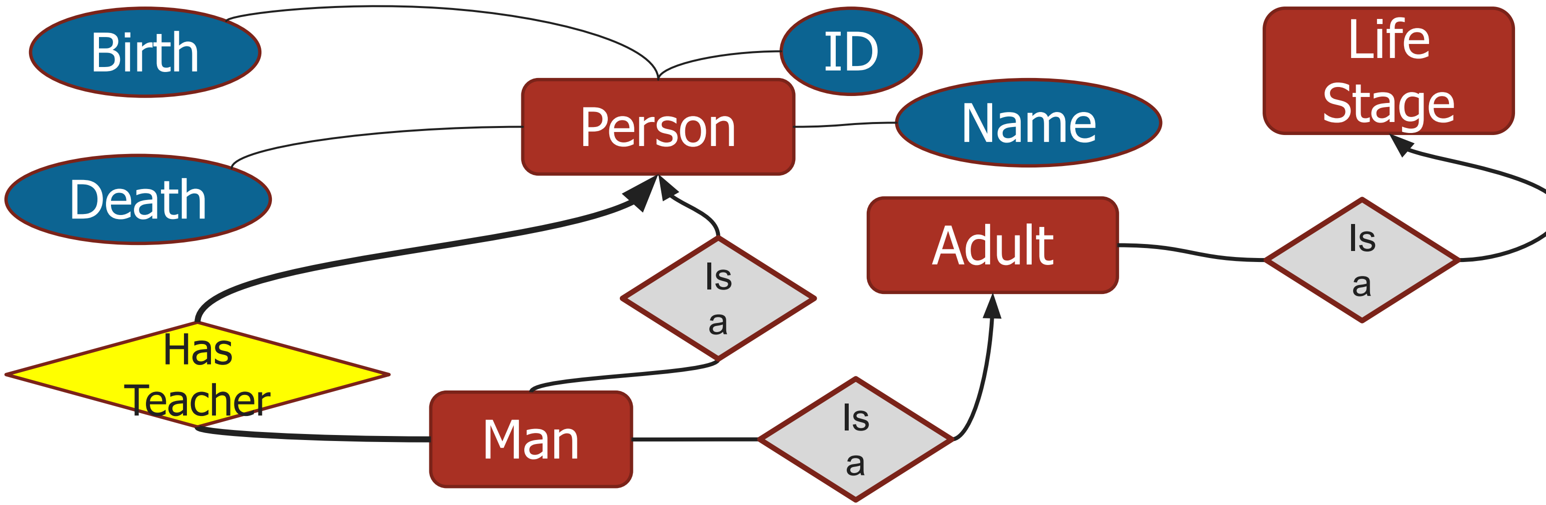
Entity-Relationship Graphs Model
Classes as Named Sets of Linked *Instances*



ID	Name	Birth	Death
1234	Aristotle	384 BC	322 BC
1233	Plato	428 BC	348 BC
1232	Socrates	470 BC	399 BC

Man is an entity set with many men, who are *also* people

Entity-Relationship Graphs: A Syntax for Entities, Properties, Relationships

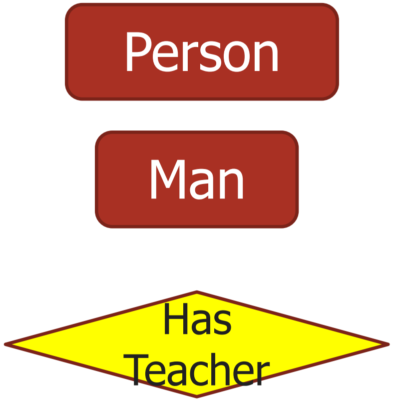


"Is a":

subclass inherits all *properties* of superclass

superclass includes all *members* of subclasses

Entities and Relationships Correspond to Relationships or Dataframes!



Entity set: represents all of the entities of a type, and their properties

- Person: ID, name, birth, death
- Man: inherits the same fields, possibly adds new ones (not shown)

Relationship set: represents a link between people

- *HasTeacher*(teacher: ID of Person, student: ID of Person)

Person (Also: Man)

ID	Name	Birth	Death
1234	Aristotle	384 BC	322 BC
1233	Plato	428 BC	348 BC
1232	Socrates	470 BC	399 BC

HasTeacher

Teacher	Student
1233	1234
1232	1233

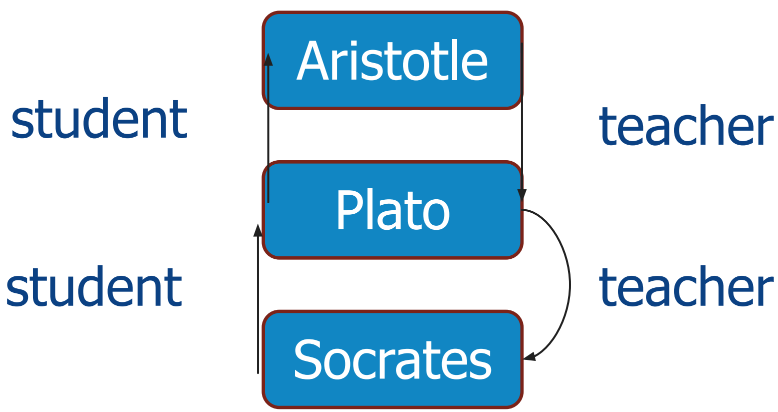
The Tables Let Us Encode a Graph within the Data!

Person

ID	Name	Birth	Death
1234	Aristotle	384 BC	322 BC
1233	Plato	428 BC	348 BC
1232	Socrates	470 BC	399 BC

HasTeacher

Teacher	Student
1233	1234
1232	1233



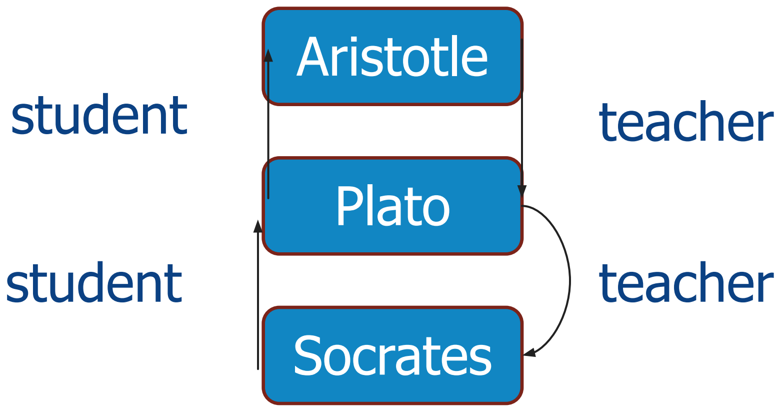
The Tables Let Us Encode a Graph within the Data!

Person

ID	Name	Birth	Death
1234	Aristotle	384 BC	322 BC
1233	Plato	428 BC	348 BC
1232	Socrates	470 BC	399 BC

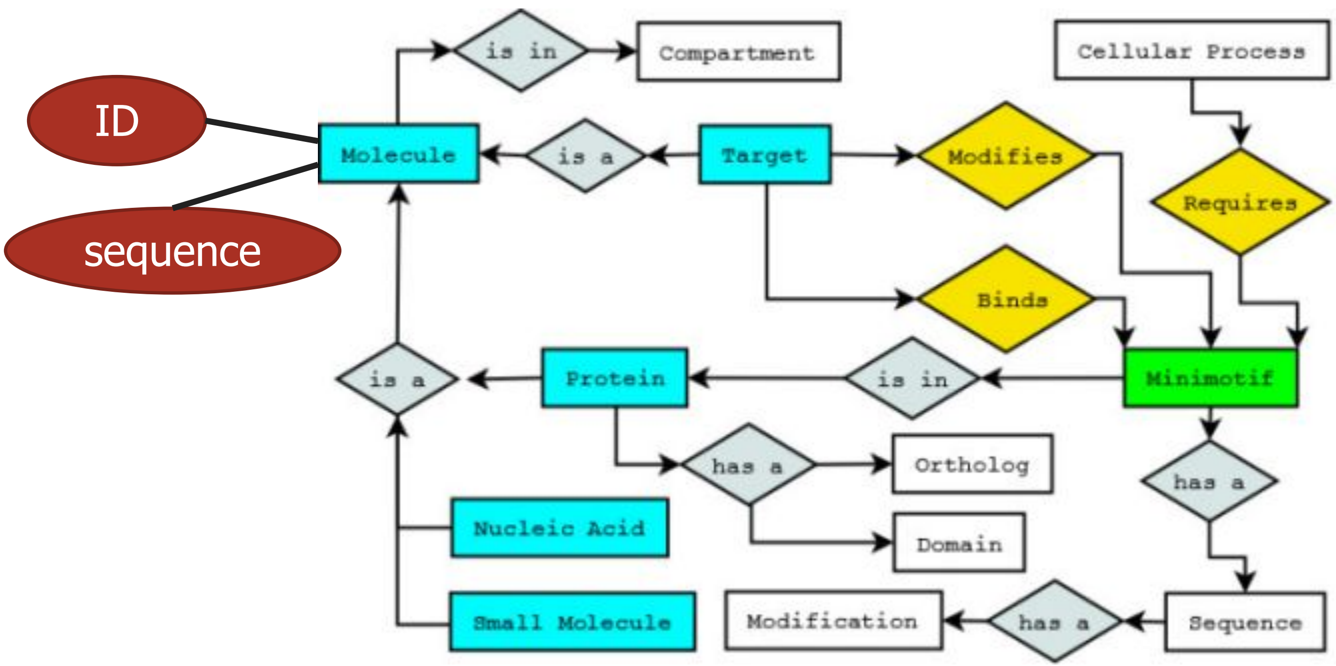
HasTeacher

Teacher	Student
1233	1234
1232	1233



In-Class Exercise:
Express using dataframe operations:
"Who is the teacher of Aristotle's teacher?"
"Show the entire tree of people taught by Socrates"?

ER is a General Model: A Graph of Entities & Relationships



Vyas et al, BMC Genomics 2009, *A proposed syntax for Minimotif Semantics, version 1*

From the Basics of Entity-Relationship Diagrams to General Data(base) Design

Deciding on the entities, relationships, and constraints is part of *database design*

- There are ways to do this to minimize the errors in the database, and make it easiest to keep *consistent*

For this class: we'll assume we do simple E-R diagrams with properties

… and that each node becomes a Dataframe

Considering Non-“Flat” Data

A Common Point of Confusion

- “Relational data can only capture flat relationships”

- **Not true**: it represents graphs, which can be traversed by queries!

… Though it might be *more convenient* to represent certain data structures!

Hierarchy vs Relations (“NoSQL” vs “SQL”)

Sometimes it’s convenient to take data we could codify as a graph:

And instead save it as a *tree* or *forest*:

[{‘person’: {‘name’: ‘jai’, phones: [{‘mfr’: ‘Apple’, ‘model’: …},
{‘mfr’: ‘Samsung’, ‘model’: …}},
{‘person’: {‘name’: ‘kai’, phones: [{‘mfr’: ‘Apple’, ‘model’: …}]}]

This is what NoSQL databases do!

NoSQL
"Not-only SQL"

Typically store **nested objects**, or possibly **binary objects**, by IDs or keys

Note that a nested object can be captured in relations, via multiple tables!

Some well-known NoSQL systems:

- MongoDB: stores JSON, i.e., lists and dictionaries
- Google Bigtable: stores tuples with irregular properties
- Amazon S3: stores binary files by key

Major differences from SQL databases:

- Querying is often much simpler, e.g. they often don't do joins!
- They support limited notions of **consistency** when you update

Recap: Basic Concepts

Knowledge is typically represented as **concepts or classes**, which can be generally thought of as corresponding to tables

- But there is also a notion of **subclassing** (inheriting fields)
- And of **instances** (rows in the tables)

Knowledge representation often describes these relationships as constraints

We can capture knowledge using graphs with nodes (entity sets, concepts) and edges (relationship sets)

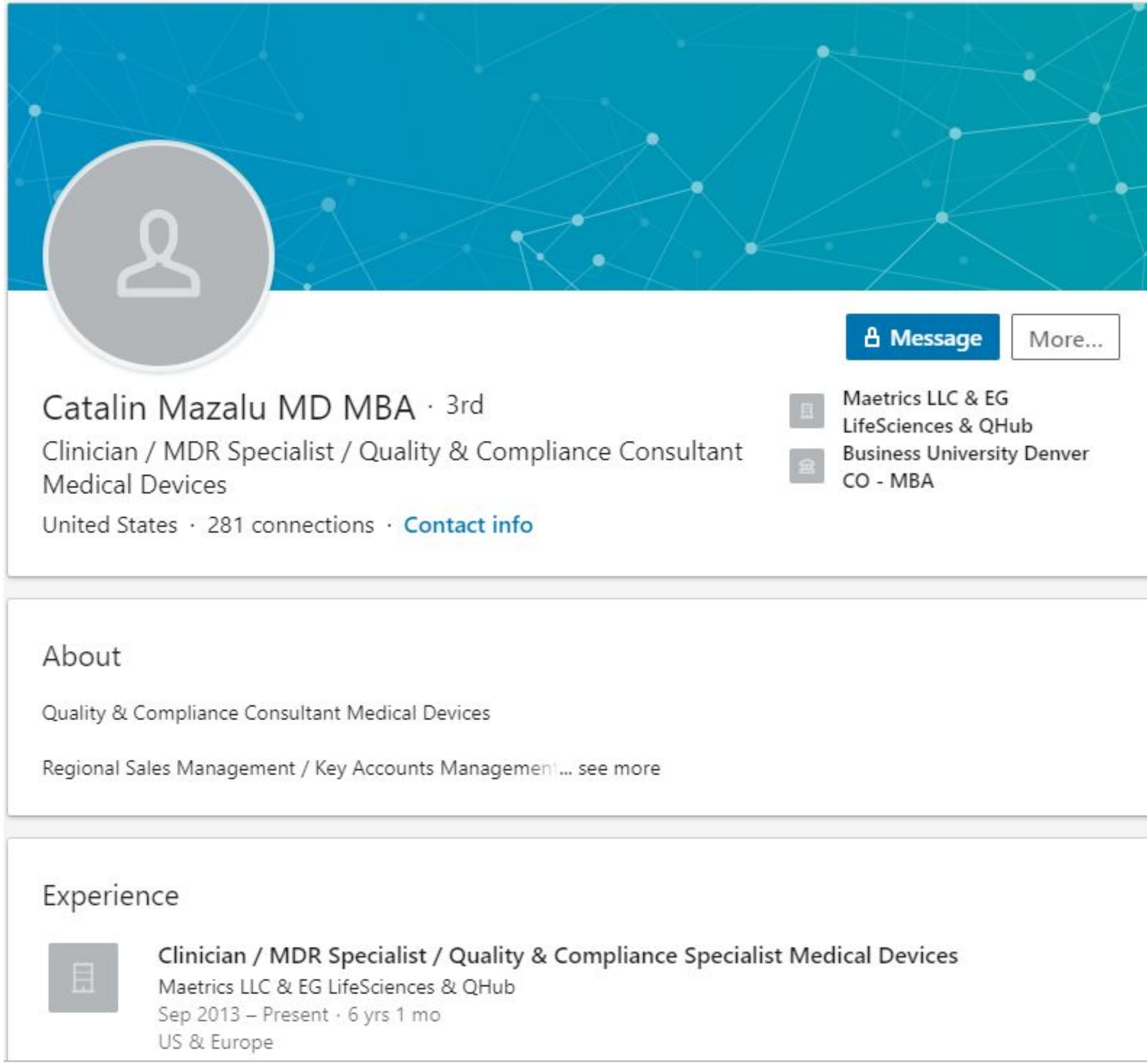
- Entity-relationship diagrams show this
- Entity sets and relationship sets can both become tables!
- Graphs + queries can be used to capture any kind of data and relationships (not always conveniently)

NoSQL systems support hierarchy, which "pivots" the graph into a *tree with a root*

Let's Work on Data Modeling, Given a Real Dataset!

1. Extracted data from LinkedIn
   - ~3M people, stored as a ~9GB list of lines made up of JSON
   - JSON is nested dictionaries and lists – i.e., NoSQL-style !
   - We'll focus on how to parse and store the "slightly hierarchical" data

2. Then we'll work out an example with *very* hierarchical data – HTML



Message
More...

Catalin Mazalu MD MBA · 3rd

Clinician / MDR Specialist / Quality & Compliance Consultant Medical Devices

United States · 281 connections · Contact info

Maetrics LLC & EG LifeSciences & QHub

Business University Denver CO - MBA

About

Quality & Compliance Consultant Medical Devices

Regional Sales Management / Key Accounts Managemen... see more

Experience

Clinician / MDR Specialist / Quality & Compliance Specialist Medical Devices

Maetrics LLC & EG LifeSciences & QHub

Sep 2013 – Present · 6 yrs 1 mo

US & Europe


```
{
 "_id":"in-00000001",
 "name":{
    "family_name":"Mazalu MBA",
    "given_name":"Dr Catalin"
 },
 "locality":"United States",
 "skills":[
    "Key Account Development",
    "Strategic Planning",
    "Market Planning",
    "Team Leadership",
    "Negotiation",
    "Forecasting",
    "Key Account Management",
    "Sales Management",
    "New Business Development",
    "Business Planning",
    "Cross-functional Team Leader
    "Budgeting",
    "Strategy Development",
    "Business Strategy",
    "Consultative Selling",
    "Medical Devices",
    "Customer Relations",
```

Parsing Even Not-So-Big Data Is Painfully Slow!

```
%%time
# 100,000 records from Linkedin
linked_in = open('linkedaa')

people = []

for line in linked_in:
    person = json.loads(line)
    people.append(person)

people_df = pd.DataFrame(people)
people_df[people_df['industry'] == 'Medical Devices']
```

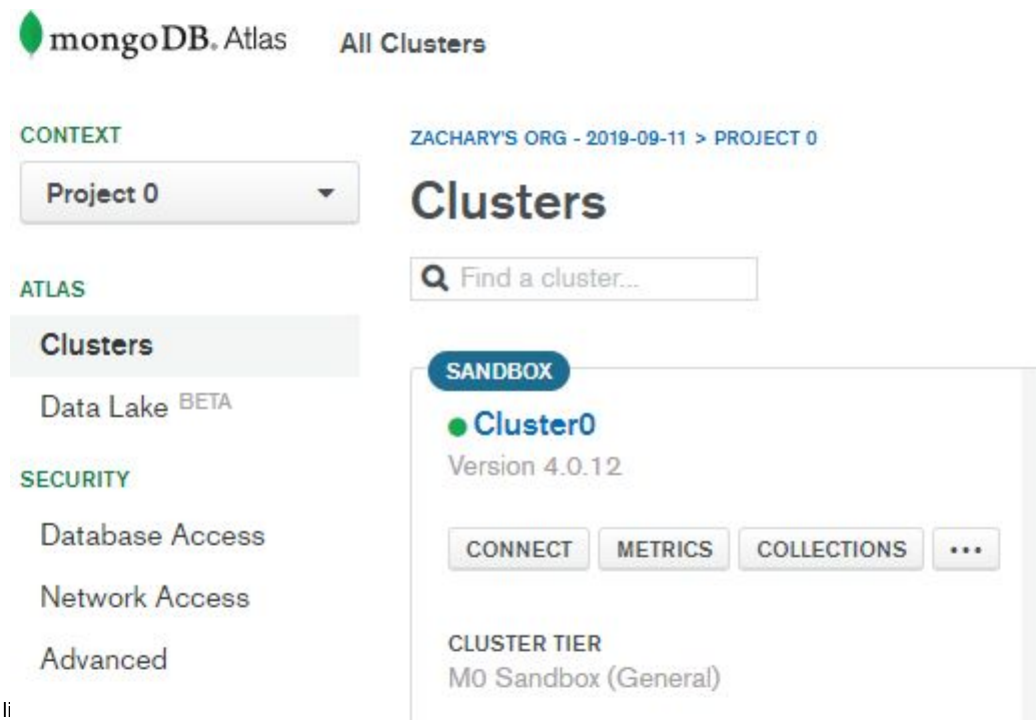
```
CPU times: user 58.2 s, sys: 1min 57s, total: 2min 55s
Wall time: 3min 19s
```

	_id	name	locality	skills	industry	summary
0	in-00000001	{'family_name': 'Mazalu MBA', 'given_name': 'D...	United States	[Key Account Development, Strategic Planning, ...	Medical Devices	SALES MANAGEMENT / BUSINESS DEVELOPMENT / PROJ...
161	in-13806219531	{'family_name': 'Gao', 'given_name': 'Tony'}	China	[ISO 13485, Medical Devices]	Medical Devices	NaN

Can We Do Better?

Maybe save the data in a way that doesn't require parsing of strings?

https://cloud.mongodb.com



MongoDB NoSQL DBMS Lets Us Store + Fetch Hierarchical Data

```
client =
MongoClient('mongodb+srv://cis545:1course4all@cluster0-cy1yu.mongodb.
net/test?retryWrites=true&w=majority')

linkedin_db = client['linkedin']
linked_in = open('linkedin.json')

for line in linked_in:
    person = json.loads(line)
    linkedin_db.posts.insert_one(person)
```

Data in MongoDB


```
_id: "in-00001"
> education: Array
> group: Object
> name: Object
  overview_html: "<dl id="overview"><dt id="overview-summary-current-title" class="summa..."
  locality: "Antwerp Area, Belgium"
> skills: Array
  industry: "Pharmaceuticals"
  interval: 20
v experience: Array
  > 0: Object
  > 1: Object
  v 2: Object
      org: "Columbia University"
      title: "Associate Research Scientist"
      start: "August 2006"
      desc: "Work on peptide to restore wt p53 function in cancer."
  > 3: Object
  > 4: Object
  summary: "Ph.D. scientist with background in cancer research, translational medi..."
  url: "http://be.linkedin.com/in/00001"
> also_view: Array
  specilities: "Biomarkers in Oncology, Cancer Genomics, Molecular Profiling of Cancer..."
> events: Array
```

Finding Things, in a Dataframe vs in MongoDB

```
def find_skills_in_list(skill):
    for post in list_for_comparison:
        if 'skills' in post:
            skills = post['skills']
            for this_skill in skills:
                if this_skill == skill:
                    return post
    return None
```

```
def find_skills_in_mongodb(skill):
    return linkedin_db.posts.find_one({'skills': skill})
```

How Do We Convert Hierarchical Data to Dataframes?

Hierarchical data doesn't work well for visualization or machine learning


```
_id: "in-00001"
> education: Array
> group: Object
> name: Object
  overview_html: "<dl id="overview"><dt id="overview-summary-current-title" class="summa..."
  locality: "Antwerp Area, Belgium"
> skills: Array
  industry: "Pharmaceuticals"
  interval: 20
v experience: Array
  > 0: Object
  > 1: Object
  v 2: Object
      org: "Columbia University"
      title: "Associate Research Scientist"
      start: "August 2006"
      desc: "Work on peptide to restore wt p53 function in cancer."
  > 3: Object
  > 4: Object
  summary: "Ph.D. scientist with background in cancer research, translational medi..."
  url: "http://be.linkedin.com/in/00001"
> also_view: Array
  specilities: "Biomarkers in Oncology, Cancer Genomics, Molecular Profiling of Cancer..."
> events: Array
```

The Basic Idea: Nesting Becomes Links ("Key/Foreign Key")

_id: "in-00001"
› education: Array
› group: Object
› name: Object
overview_html: "<dl id="overview"><dt id="overview-summary-cu
locality: "Antwerp Area, Belgium"
› skills: Array
industry: "Pharmaceuticals"
interval: 20
⌄ experience: Array
› 0: Object
› 1: Object
⌄ 2: Object
org: "Columbia University"
title: "Associate Research Scientist"
start: "August 2006"
desc: "Work on peptide to restore wt p53 function in ca
› 3: Object
› 4: Object
summary: "Ph.D. scientist with background in cancer research,
url: "http://be.linkedin.com/in/00001"
› also_view: Array
specilities: "Biomarkers in Oncology, Cancer Genomics, Molecu
› events: Array

people

_id	Overview_html	locality	industry	...
in-00001	<dl id=...	Antwerp Area	Pharmaceu	

experience

person	org	title	start	desc
in-00001	Columbia	Assoc...	August	Wor...
in-00001	...	...	...	...

Reassembling through (Outer) Joins

```
pd.read_sql_query("select _id, org" +\
                  " from people left join experience on _id=person ",\
                  conn)
```

_id	org
in-00001	Albert Einstein Medical Center
in-00001	Columbia University
in-00001	Johnson and Johnson

```
pd.read_sql_query("select _id, \'[\' + group_concat(org) + \']\'" +\
                  " from people left join experience on _id=person "+\
                  " group by _id", conn)
```

_id	experience
in-00000001	None
in-00001	Albert Einstein Medical Center,Columbia Univer...
in-00006	UCSF,Wyss Institute for Biologically Inspired ...

Views

Sometimes we use a query enough that we want to give its results a name, and make it essentially a table (which we then use in other queries!)

```
conn.execute('begin transaction')
conn.execute('drop view if exists people_experience')
conn.execute("create view people_experience as " +\
             " select _id, group_concat(org) as experience " +\
             " from people left join experience on _id=person group by _id")
conn.execute('commit')

pd.read_sql_query('select * from people_experience', conn)
```

Occasional Considerations: Access and Consistency

Sometimes we may need to allow for failures and "undo"…

- We saw "BEGIN TRANSACTION … COMMIT"
- There is also "ROLLBACK"

Relational DBMS typically provide atomic **transactions** for this; most NoSQL DBMSs don't

A second consideration when the data is shared: what happens when multiple users are editing and querying at the same time?

- **Concurrency control** (how do we handle concurrent updates) and **consistency** (when do I see changes)

Summary of Data Modeling

We have a large hierarchical dataset for LinkedIn

- It takes a long time to load / parse

We can load it into MongoDB, which stores it ~directly

- Can retrieve by patterns, a bit like XPath

We can split it into dataframes or SQL tables, and we can reassemble by joins

- Grouping with concatenation can rebuild our sets, if we really want
- And *views* let us give a name to the reassembled results

If data isn't static, we should consider **transactions** and **concurrency**