SQL, NOSQL, DATA WRANGLING & CLEANING

Prof. Mohammad Hajiaghayi & Arefeh Nasri
Wiki & Linkedin: @Mohammad Hajiaghayi
Twitter:@MTHajiaghayi
YouTube:@hajiaghayi [PLEASE SUBSCRIBE]
Instagram:@mhajiaghayi
Original slides prepared by Prof. Amol Deshpande

Lectures #8

DATA/MSML602: Principles of Data Science
TuTh6:00pm – 8:30pm

REVIEW OF LAST CLASS

- Modeling and Manipulating Data
- NumPy and SciPy
- Abstraction of Tables and Operations
- Python Pandas Library
- Final Project Questions?

TODAY'S CLASS

- Data Modeling
- SQL and Relational Databases
- NoSQL Databases
- Data Wrangling/Cleaning

OVERVIEW

- Data Modeling
  - Process of representing/capturing the structure in the data
  - **Data model**: A collection of concepts that describes how data is represented and accessed
  - **Schema:** A description of a specific collection of data, using a given data model
- Why?
  - Need to know the structure of the data/information (to some extent) to be able to write general purpose code
  - Lack of a data model makes it difficult to share data across programs, organizations, systems
  - Need to be able to integrate information from multiple sources
  - Efficiency: Can preprocess data to make access efficient (e.g., building a B-Tree on a field)

OVERVIEW

- A data model typically consists of:
  - Modeling Constructs: A collection of concepts used to represent the structure in the data
    - Typically need to represent types of *entities*, their *attributes*, types of *relationships* between *entities*, and *relationship attributes*
  - Integrity Constraints: Constraints to ensure data integrity (i.e., avoid errors)
  - Manipulation Languages: Constructs for manipulating the data
- We would like it to be:
  - Sufficiently expressive -- can capture real-world data well
  - Easy to use
  - Lends itself to good performance
- The history of modeling can be seen as an attempt to capture the structure in the data.

OVERVIEW

- Some examples of data models
  - Relational, Entity-relationship model, XML…
  - Object-oriented, Object-relational, RDF…
  - Current favorites in the industry: JSON, Protocol Buffers, Avro, Thrift, Property Graph

- Why so many models ?
  - Tension between descriptive power and ease of use/efficiency
  - More powerful models --> more datasets can be represented
  - More powerful models --> harder to use, to query, and less efficient

OVERVIEW

- Typically there are multiple levels of modeling
  - Physical modeling concerns itself with how the data is physically stored
  - Logical or Conceptual modeling concerns itself with type of information stored, the different entities, their attributes, and the relationships among those
- There may be several layers of logical/conceptual models to restrict the information flow (for security and/or ease-of-use)
- **Data independence:** The idea that you can change the representation of data w/o changing programs that operate on it.
- **Physical data independence:** I can change the layout of data on disk and my programs won't change index the data
  - partition/distribute/replicate the data
  - compress the data
  - sort the data

EARLY: HIERARCHICAL AND NETWORK MODELS

- Both allowed "connecting" records of different types (e.g., connect "accounts" with "customers")
- Network model attempted to be very general and flexible
  - Charlie Bachman received Turing Award
- IBM designed its IMS hierarchical database in 1966 for the Apollo space program; still around today
  - *.. more than 95 percent of the top Fortune 1000 companies use IMS to process more than 50 billion transactions a day and manage 15 million gigabytes of critical business data* (from IBM Website on IMS)
- Predates *hard disks*
- However, both models exposed too much of the internal data structures/pointers etc. to the users

What goes around comes around; Stonebraker and Hellestein

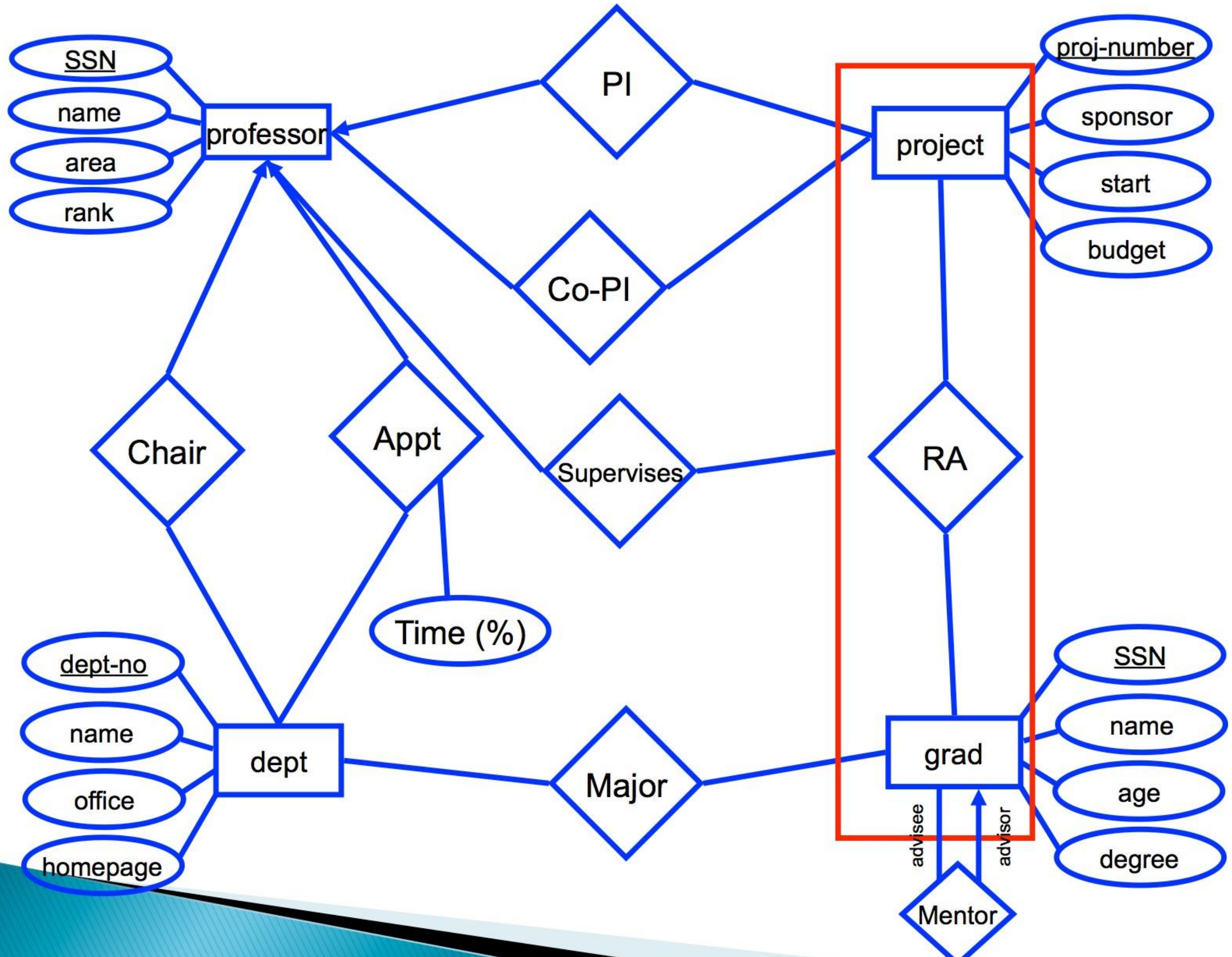
1970'S: RELATIONAL MODEL

- Origins in Set Theory
  - Some early work by D.L.Childs (somewhat forgotten)
  - Edgar F. "Ted" Codd: Developed the relational model
- Elegant, formal model that provided almost complete *data independence*
  - Users didn't need to worry about how the data was stored, processed
  - High level query language (relational algebra)
- Notion of *normal forms*
  - Allowed one to reason about and remove redundancies
- Led to two influential projects: INGRES (Berkeley), System R (IBM)
  - Also paved the way for a 1977 startup called "Software Development Laboratories"
  - Didn't care about IMS/IDMS compatibility (as IBM had to)
- Many debates in the early 70's between Relational Model proponents and Network Model proponents

1976: ENTITY-RELATIONSHIP MODEL

- Proposed by Peter Chen
- Database should be thought of as Entities, that are connected by Relationships
  - Made comeback as ORMs (Ruby on Rails, Django etc).
- Never gained traction as the main underlying data model
  - No Query Language
  - Easy to translate to Relational Model
- Very successful as a "conceptual"/"logical" model
  - Used to make sense of the data
  - Used to come up with the initial set of models

1976: ENTITY-RELATIONSHIP MODEL



1980'S: OTHER MODELS

- Many models trying to enrich the basic relational model to add set-valued attributes, aggregation, generalization
- GEM
- Semantic Data Models
- Object-oriented Data Model: to get around *impedance mismatch* between programming languages and databases
- Object-relational Data model: allow user-defined types -- gets many benefits of OO while keeping the essence of relational model
  - No real differentiation today from pure relational model

- See: "What goes around comes around": Michael Stonebraker, Joe Hellerstein;
  - For a very nice overview and history

90'S: XML

- XML: eXtensible Markup Language
  - Intended for *semi-structured* data
  - Flexible schema
- Niche usecase when you really think about it
- Popular as wire format (for exchanging data)
  - Overtaken by JSON

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Edited by XMLSpy -->
<CATALOG>
    <CD>
        <TITLE>Empire Burlesque</TITLE>
        <ARTIST>Bob Dylan</ARTIST>
        <COUNTRY>USA</COUNTRY>
        <COMPANY>Columbia</COMPANY>
        <PRICE>10.90</PRICE>
        <YEAR>1985</YEAR>
    </CD>
    <CD>
        <TITLE>Hide your heart</TITLE>
        <ARTIST>Bonnie Tyler</ARTIST>
        <COUNTRY>UK</COUNTRY>
        <COMPANY>CBS Records</COMPANY>
        <PRICE>9.90</PRICE>
        <YEAR>1988</YEAR>
    </CD>
    ...
```

90'S: RDF

- Originally intended as a "metadata data model"
- Key construct: a "subject-predicate-object" triple
  - E.g., subject=sky - predicate=has-the-color - object=blue
- Direct mapping to a labeled, directed multi-graph
- Typically stored in relational databases, or what are called "triple-stores"
- But some graph database products out there as well (e.g., DEX)

- Very common in Semantic Web and Knowledge Graph Community

90'S: RDF

- <rdf:RDF xmlns:contact="http://www.w3.org/2000/10/swap/pim/contact#" xmlns:eric="http://www.w3.org/People/EM/contact#" xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
- <rdf:Description rdf:about="http://www.w3.org/People/EM/contact#me"> contact:fullNameEric Miller</contact:fullName> </rdf:Description>
- <rdf:Description rdf:about="http://www.w3.org/People/EM/contact#me"> <contact:mailbox rdf:resource="mailto:e.miller123(at)example"/> </rdf:Description>
- <rdf:Description rdf:about="http://www.w3.org/People/EM/contact#me"> contact:personalTitleDr.</contact:personalTitle> </rdf:Description>
- <rdf:Description rdf:about="http://www.w3.org/People/EM/contact#me"> <rdf:type rdf:resource="http://www.w3.org/2000/10/swap/pim/contact#Person"/> </rdf:Description>
- </rdf:RDF>

90'S: RDF


http://www.w3.org/2000/10/swap/pim/contact#Person
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
http://www.w3.org/People/EM/contact#me
http://www.w3.org/2000/10/swap/pim/contact#fullName
Eric Miller
http://www.w3.org/2000/10/swap/pim/contact#mailbox
mailto:em@w3.org
http://www.w3.org/2000/10/swap/pim/contact#personalTitle
Dr.

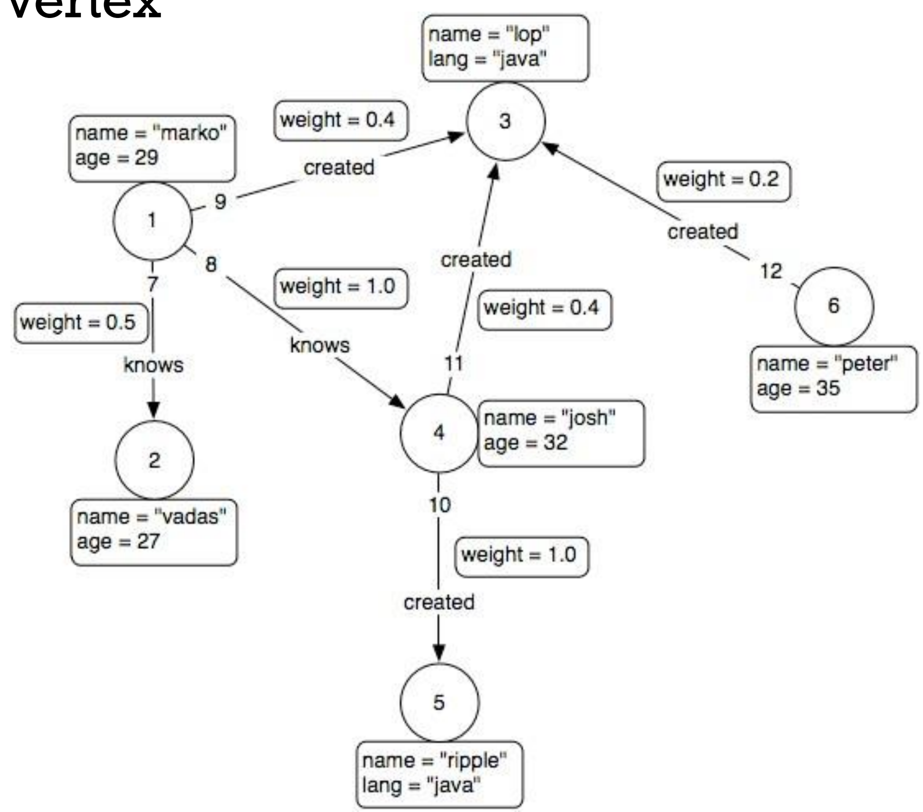
00's: JSON

- Similar to XML: Hierarchical data model
  - Some differences like support for Arrays
- Overtaking XML as a wire format
  - Likely because of Javascript usage
- Many databases out there support it natively (MongoDB)

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 25,
  "height_cm": 167.6,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [],
  "spouse": null
}
```

00'S: PROPERTY GRAPH MODEL

- Developed for graph databases
- Basically a edge- and vertex-labeled graph, with properties associated with each edge and vertex



RELATED: SERIALIZATION FORMATS

- Need a way for programs/systems to send data to each other
- Several recent technologies all based around schemas
- Protocol Buffers: Developed by Google
  - Schema is mostly relational, with support for optional fields and some other constructs
  - Schema specified using a .proto file

```
message Person {
    required int32 id = 1;
    required string name = 2;
    optional string email = 3;
}
```

RELATED: SERIALIZATION FORMATS

- Need a way for programs/systems to send data to each other
- Several recent technologies all based around schemas
- Protocol Buffers: Developed by Google
  - Compiled by protoc to produce C++, Java, or Python code
  - Programs can be written in any of those languages, e.g., C++:

```
Person person;
person.set_id(123);
person.set_name("Bob");
person.set_email("bob@example.com");
fstream out("person.pb", ios::out | ios::binary | ios::trunc);
person.SerializeToOstream(&out);
out.close();
```

RELATED: SERIALIZATION FORMATS

- Avro: Richer data structures, JSON-specified schema

```
{
    "namespace": "example.avro",
    "type": "record",
    "name": "User",
    "fields": [
            {"name": "name", "type": "string"},
            {"name": "favorite_number",  "type": ["int", "null"]},
            {"name": "favorite_color", "type": ["string", "null"]}
    ]
}
```

TODAY'S CLASS

- Data Modeling
- SQL and Relational Databases
- NoSQL Databases
- Data Wrangling/Cleaning

RELATION

- Simplest relation: a table aka tabular data full of tuples
- Based on the concept of *mathematical relation*

Columns
(also called attributes)

Tuples, rows,
records

ID	age	wgt_kg	hgt_cm
1	12.2	42.3	145.1
2	11.0	40.8	143.8
3	15.6	65.3	165.3
4	35.1	84.2	185.8

PRIMARY KEYS

ID	age	wgt_kg	hgt_cm	nat_id
1	12.2	42.3	145.1	1
2	11.0	40.8	143.8	1
3	15.6	65.3	165.3	2
4	35.1	84.2	185.8	1
5	18.1	62.2	176.2	3
6	19.6	82.1	180.1	1

ID	Nationality
1	USA
2	Canada
3	Mexico

- The primary key is a unique identifier for every tuple in a relation
- Doesn’t have to be called ID
- May consist of >1 attribute

AREN'T THESE CALLED "INDEXES"?

- Yes, in Pandas; but not in the database world

- For most databases, an "index" is a data structure used to speed up retrieval of specific tuples

- For example, to find all tuples with nat_id = 2:
  - We can either scan the table – O(N)
  - Or use an "index" (e.g., binary tree) – O(log N)

FOREIGN KEYS

ID	age	wgt_kg	hgt_cm	nat_id
1	12.2	42.3	145.1	1
2	11.0	40.8	143.8	1
3	15.6	65.3	165.3	2
4	35.1	84.2	185.8	1
5	18.1	62.2	176.2	3
6	19.6	82.1	180.1	1

ID	Nationality
1	USA
2	Canada
3	Mexico

- Foreign keys are attributes (columns) that point to a different table's primary key
  - A table can have multiple foreign keys

RELATION SCHEMA

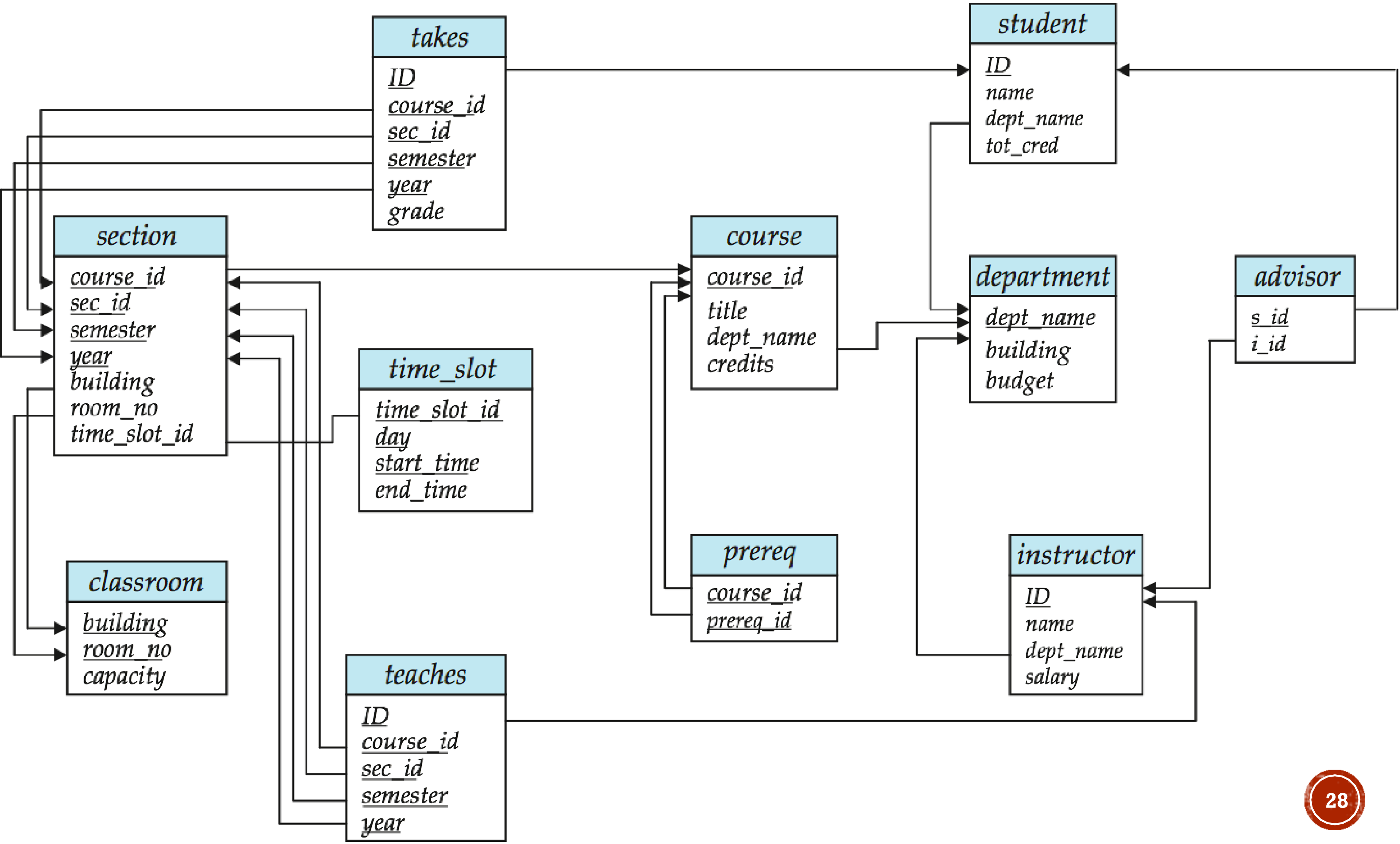
- A list of all the attribute names, and their *domains*

```
create table department
    (dept_name varchar(20),
    building varchar(15),
     budget numeric(12,2) check (budget > 0),
     primary key (dept_name)
    );
```

SQL Statements
To create Tables

```
create table instructor (
    ID        char(5),
    name    varchar(20) not null,
    dept_name  varchar(20),
    salary   numeric(8,2),
    primary key (ID),
    foreign key (dept_name) references department
)
```

SCHEMA DIAGRAMS



RELATIONSHIPS

- Primary keys and foreign keys define interactions between different tables aka entities. Four types:
  - One-to-one
  - One-to-one-or-none
  - One-to-many and many-to-one
  - Many-to-many

- Connects (one, many) of the rows in one table to (one, many) of the rows in another table

ONE-TO-MANY & MANY-TO-ONE

- One person can have one nationality in this example, but one nationality can include many people.

Person → Nationality

ID	age	wgt_kg	hgt_cm	nat_id
1	12.2	42.3	145.1	1
2	11.0	40.8	143.8	1
3	15.6	65.3	165.3	2
4	35.1	84.2	185.8	1
5	18.1	62.2	176.2	3
6	19.6	82.1	180.1	1

ID	Nationality
1	USA
2	Canada
3	Mexico

ONE-TO-ONE

- Two tables have a one-to-one relationship if every tuple in the first tables corresponds to exactly one entry in the other

Person	SSN

- In general, you won't be using these (why not just merge the rows into one table?) unless:
  - Split a big row between SSD and HDD or distributed
  - Restrict access to part of a row (some DBMSs allow column-level access control, but not all)
  - Caching, partitioning, & serious stuff: take CMSC424

ONE-TO-ONE-OR-NONE

- Say we want to keep track of people's cats:

Person ID	Cat1	Cat2
1	Chairman Meow	Fuzz Aldrin
4	Anderson Pooper	Meowly Cyrus
5	Gigabyte	Megabyte

- People with IDs 2 and 3 do not own cats, and are not in the table. Each person has at most one entry in the table.

Person

Cat

MANY-TO-MANY

- Say we want to keep track of people's cats' colorings:

ID	Name
1	Megabyte
2	Meowly Cyrus
3	Fuzz Aldrin
4	Chairman Meow
5	Anderson Pooper
6	Gigabyte

Cat ID	Color ID	Amount
1	1	50
1	2	50
2	2	20
2	4	40
2	5	40
3	1	100

- One column per color, too many columns, too many nulls
- Each cat can have many colors, and each color many cats

ASSOCIATIVE TABLES

Cats

ID	Name
1	Megabyte
2	Meowly Cyrus
3	Fuzz Aldrin
4	Chairman Meow
5	Anderson Pooper
6	Gigabyte

Cat ID	Color ID	Amount
1	1	50
1	2	50
2	2	20
2	4	40
2	5	40
3	1	100

Colors

ID	Name
1	Black
2	Brown
3	White
4	Orange
5	Neon Green
6	Invisible

- Primary key
  - [Cat ID, Color ID] (+ [Color ID, Cat ID], case-dependent)
- Foreign key(s)
  - Cat ID and Color ID

ASIDE: PANDAS

- So, this kinda feels like pandas …
  - And pandas kinda feels like a relational data system …
- Pandas is not strictly a relational data system:
  - No notion of primary / foreign keys
- It does have indexes (and multi-column indexes):
  - pandas.Index: ordered, sliceable set storing axis labels
  - pandas.MultiIndex: hierarchical index
- Rule of thumb: do heavy, rough lifting at the relational DB level, then fine-grained slicing and dicing and viz with pandas

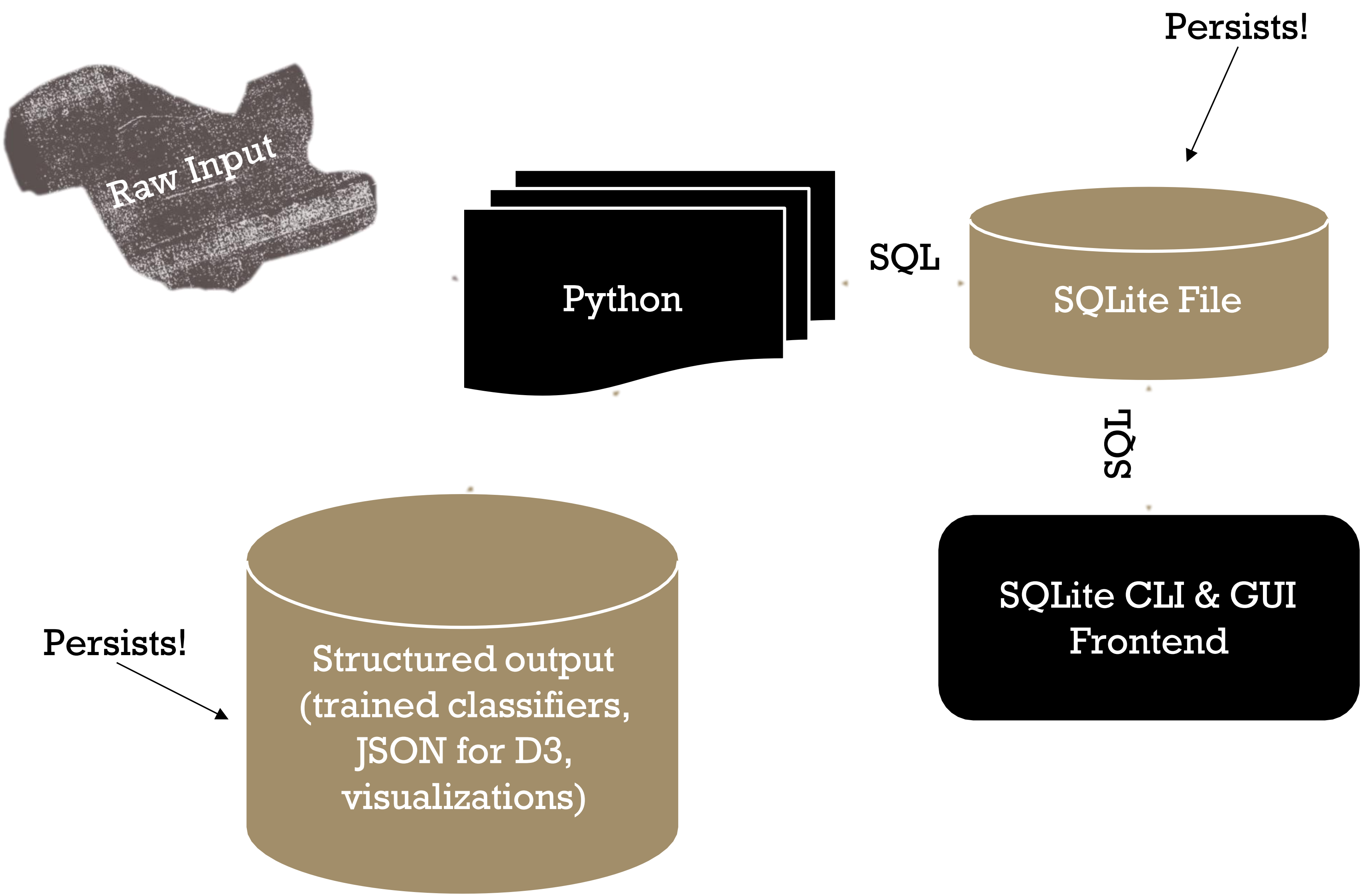
TODAY'S CLASS

- Data Modeling
- SQL and Relational Databases
  - SQLite and Python
  - Other Relational Databases
  - Key features outside of SQL
- NoSQL Databases
- Data Wrangling/Cleaning

SQLITE

- On-disk relational database management system (RDMS)
  - Applications connect directly to a file
- Most RDMSs have applications connect to a server:
  - Advantages include greater concurrency, less restrictive locking
  - Disadvantages include, for this class, setup time ©
- Installation:
  - `conda install -c anaconda sqlite`
  - Preinstalled on Docker image
- All interactions use Structured Query Language (SQL)

HOW A RELATIONAL DB FITS INTO YOUR WORKFLOW



CRASH COURSE IN SQL (IN PYTHON)

```
import sqlite3

# Create a database and connect to it
conn = sqlite3.connect("cmsc320.db")
cursor = conn.cursor()

# do cool stuff
conn.close()
```

- Cursor: temporary work area in system memory for manipulating SQL statements and return values
- If you do not close the connection (`conn.close()`) or commit transaction (conn.commit()), any outstanding transactions are rolled back

CRASH COURSE IN SQL (IN PYTHON)

```
# Make a table
cursor.execute("""
CREATE TABLE cats (
    id INTEGER PRIMARY KEY,
    name TEXT
)""")
```

?????????

id	name

cats

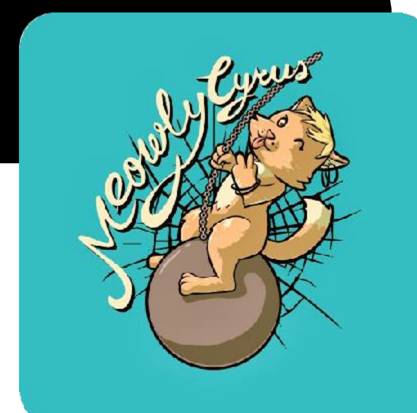
- Capitalization doesn't matter for SQL reserved words
  - SELECT = select = SeLeCt
- Rule of thumb: capitalize keywords for readability

CRASH COURSE IN SQL (IN PYTHON)

```
# Insert into the table
cursor.execute("INSERT INTO cats VALUE (1, 'Megabyte')")
cursor.execute("INSERT INTO cats VALUE (2, 'Meowly Cyrus')")
cursor.execute("INSERT INTO cats VALUE (3, 'Fuzz Aldrin')")
conn.commit()
```

id	name
1	Megabyte
2	Meowly Cyrus
3	Fuzz Aldrin

```
# Delete row(s) from the table
cursor.execute("DELETE FROM cats WHERE id == 2");
conn.commit()
```



id	name
1	Megabyte
3	Fuzz Aldrin

CRASH COURSE IN SQL (IN PYTHON)

```
# Read all rows from a table
for row in cursor.execute("SELECT * FROM cats"):
    print(row)
```

```
# Read all rows into pandas DataFrame
pd.read_sql_query("SELECT * FROM cats", conn, index_col="id")
```

id	name
1	Megabyte
3	Fuzz Aldrin

- index_col="id": treat column with label "id" as an index
- index_col=1: treat column #1 (i.e., "name") as an index
- (Can also do multi-indexing.)

CRASH COURSE IN SQL (IN PYTHON)

- Use WHERE to filter out rows – allows regular expressions, but different syntax

```
for row in cursor.execute("SELECT * FROM cats WHERE name like
'%XYZ%'"):
    print(row)
```

JOINING DATA

- A join operation merges two or more tables into a single relation. Different ways of doing this:
  - Inner
  - Left
  - Right
  - Full Outer

- Join operations are done on columns that explicitly link the tables together

INNER JOINS

id	name
1	Megabyte
2	Meowly Cyrus
3	Fuzz Aldrin
4	Chairman Meow
5	Anderson Pooper
6	Gigabyte

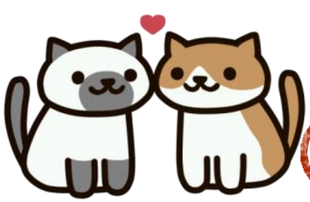
`cats`

cat_id	last_visit
1	02-16-2017
2	02-14-2017
5	02-03-2017

`visits`

- Inner join returns merged rows that share the same value in the column they are being joined on (`id` and `cat_id`).

id	name	last_visit
1	Megabyte	02-16-2017
2	Meowly Cyrus	02-14-2017
5	Anderson Pooper	02-03-2017



INNER JOINS

```
# Inner join in pandas
df_cats = pd.read_sql_query("SELECT * from cats", conn)
df_visits = pd.read_sql_query("SELECT * from visits", conn)
df_cats.merge(df_visits, how = "inner",
              left_on = "id", right_on = "cat_id")
```

```
# Inner join in SQL / SQLite via Python
cursor.execute("""
               SELECT
                   *
               FROM
                   cats, visits
               WHERE
                   cats.id == visits.cat_id
               """)
```

LEFT JOINS

- Inner joins are the most common type of joins (get results that appear in both tables)
- Left joins: all the results from the left table, only some matching results from the right table
- Left join (`cats`, `visits`) on (`id`, `cat_id`)

id	name	last_visit
1	Megabyte	02-16-2017
2	Meowly Cyrus	02-14-2017
3	Fuzz Aldrin	NULL
4	Chairman Meow	NULL
5	Anderson Pooper	02-03-2017
6	Gigabyte	NULL

RIGHT JOINS

- Take a guess!
- Right join (`cats,visits`) on (`id,cat_id`)

id	name
1	Megabyte
2	Meowly Cyrus
3	Fuzz Aldrin
4	Chairman Meow
5	Anderson Pooper
6	Gigabyte

`cats`

cat_id	last_visit
1	02-16-2017
2	02-14-2017
5	02-03-2017
7	02-19-2017
12	02-21-2017

`visits`

id	name	last_visit
1	Megabyte	02-16-2017
2	Meowly Cyrus	02-14-2017
5	Anderson Pooper	02-03-2017
7	NULL	02-19-2017
12	NULL	02-21-2017

LEFT/RIGHT JOINS

```
# Left join in pandas
df_cats.merge(df_visits, how = "left",
                  left_on = "id", right_on = "cat_id")
```

```
# Left join in SQL / SQLite via Python
cursor.execute("SELECT * FROM cats LEFT JOIN visits ON
                 cats.id == visits.cat_id")
```

```
# Right join in pandas df_cats.merge(df_visits, how =
"right",
              left_on = "id", right_on = "cat_id")
```

```
# Right join in SQL / SQLite via Python
cursor.execute("SELECT * FROM cats RIGHT JOIN visits ON
                 cats.id == visits.cat_id")
```

FULL OUTER JOIN

- Combines the left and the right join

id	name	last_visit
1	Megabyte	02-16-2017
2	Meowly Cyrus	02-14-2017
3	Fuzz Aldrin	NULL
4	Chairman Meow	NULL
5	Anderson Pooper	02-03-2017
6	Gigabyte	NULL
7	NULL	02-19-2017
12	NULL	02-21-2017

```
# Outer join in pandas
df_cats.merge(df_visits, how = "outer",
              left_on = "id", right_on = "cat_id")
```

```
# OUTER join in SQL / SQLite via Python
cursor.execute("SELECT * FROM cats FULL OUTER JOIN visits
ON cats.id == visits.cat_id")
```

GROUP BY AGGREGATES

```
SELECT nat_id, AVG(age) as average_age
FROM persons GROUP BY nat_id
```

ID	age	wgt_kg	hgt_cm	nat_id
1	12.2	42.3	145.1	1
2	11.0	40.8	143.8	1
3	15.6	65.3	165.3	2
4	35.1	84.2	185.8	1
5	18.1	62.2	176.2	3
6	19.6	82.1	180.1	1

nat_id	average_age
1	19.48
2	15.6
3	18.1

RAW SQL IN PANDAS

- If you "think in SQL" already, you'll be fine with pandas (or):
  - `conda install -c anaconda sqldf`
  - Info: https://pypi.org/project/sqldf/
  - (also see pandasql)

```
# Write the query text

q = """
    SELECT   *
    FROM
        cats LIMIT
    10;"""

# Run and Store in a DataFrame
df = sqldf.run(q)
```

TODAY'S CLASS

- Data Modeling
- SQL and Relational Databases
  - SQLite and Python
  - Other Relational Databases
  - Key features outside of SQL
- NoSQL Databases
- Data Wrangling/Cleaning

STATE OF THE ART: RELATIONAL DATABASES

- Oracle, IBM DB2, Microsoft SQL Server, Sybase
- Open source alternatives
  - MySQL, PostgreSQL, Apache Derby, BerkeleyDB (mainly a storage engine – no SQL), …
- Data Warehousing Solutions
  - Geared towards very large volumes of data and on analyzing them
  - Long list: Google BigQuery, Teradata, Oracle Exadata, Netezza (based on FPGAs), Aster Data (founded 2005), Vertica (column-based), Kickfire, Xtremedata (released 2009), Sybase IQ, Greenplum (eBay, Fox Networks use them)
  - Usually sell package/services and charge per TB of managed data
  - Many (especially recent ones) start with MySQL or PostgreSQL and make them parallel/faster etc..

STATE OF THE ART: RELATIONAL DATABASES

- Many cloud-based solutions today
- Amazon RDS
  - Basically AWS-managed MySQL or PostgreSQL or MariaDB or Oracle or SQL Server
  - Amazon also has its own equivalent to MySQL called "AuroraDB"
- Amazon Redshift
  - Data Warehousing in Amazon
- Azure and GCloud also support MySQL, PostgreSQL and SQL Server

- Most big data frameworks export an SQL interface
  - E.g., Hadoop Hive, Apache Spark, Flink etc.
  - *Would you call them "relational databases"?*

- Even many key-value stores do, but in a limited form
  - E.g., Apache Cassandra

TODAY'S CLASS

- Data Modeling
- SQL and Relational Databases
  - SQLite and Python
  - Other Relational Databases
  - Key features outside of SQL
- NoSQL Databases
- Data Wrangling/Cleaning

KEY OTHER FEATURES

- Update Transactions and ACID properties
  - Strict guarantees on correctness of updates because of the use-cases
  - Even today: RDBMSs are used at the backend for important information
- Integrity Constraints
  - Add constraints to the data so that incorrect data cannot be added
  - E.g., if two tuples have the same "zipcode", they must have the same "state"
  - E.g., two different users cannot have the same account id

KEY OTHER FEATURES

- Bulk Loading
  - Ability to quickly load and unload databases
  - Usually optimized because of its importance
- Much more complex types of SQL queries
  - See CMSC424 slides for a gentle introduction
  - PostgreSQL manual a great resource too
- Triggers see in particular GCloud SubPub
  - Ability to take an action based on some property being satisfied
  - E.g., if the inventory too low, place an order
- More features being added all the time
  - E.g., SQL Server just announced support for "graph queries"

TODAY'S CLASS

- Data Modeling
- SQL and Relational Databases
- NoSQL Databases
- Data Wrangling/Cleaning

NOSQL DATABASES

- Somewhat vague classification
- Document Databases: Typically using JSON or XML data model
  - CouchDB, ArangoDB, MongoDB, OrientDB, Elastic, …
- Key-value Stores: Very basic key-based interface
  - Redis, Cassandra, FoundationDB, Hbase
- Graph Databases: RDF or Property Graph Model
  - Neo4j, Titan, OrientDB, several others in recent years
- Many others (e.g., time-series databases, message stores)
- Many of the systems hard to classify as they support multiple modalities

TYPICAL NOSQL API

- Basic API access:
  - get(key) -- Extract the value given a key
  - put(key, value) -- Create or update the value given its key
  - delete(key) -- Remove the key and its associated value
  - execute(key, operation, parameters) -- Invoke an operation to the value (given its key) which is a special data structure (e.g. List, Set, Map .... etc).

WHEN TO USE WHAT?

- Advantages of NoSQL Stores
  - Easier to get started
  - Don't need schemas (plus and minus)
  - Easier to scale out to a cluster of computers
  - Nicer tie-in with Javascript and other (especially MongoDB)
  - Some Niche use cases better supported
  - Usually cheaper to buy (should count maintenance though)
- Advantages of Relational Databases
  - Very mature and robust
  - ACID transactions
  - Generally perform better for a single machine
  - Much better querying ability – less need to do stuff like joins outside

- Rule of thumb: Have a very good reason if not using a relational database (e.g., PostgreSQL)

TODAY'S CLASS

- Data Modeling
- SQL and Relational Databases
- NoSQL Databases
- Data Wrangling/Cleaning
  - Quantitative/Statistical Cleaning
  - Qualitative/Logical Cleaning
  - Missing Data
  - Entity Resolution

OVERVIEW

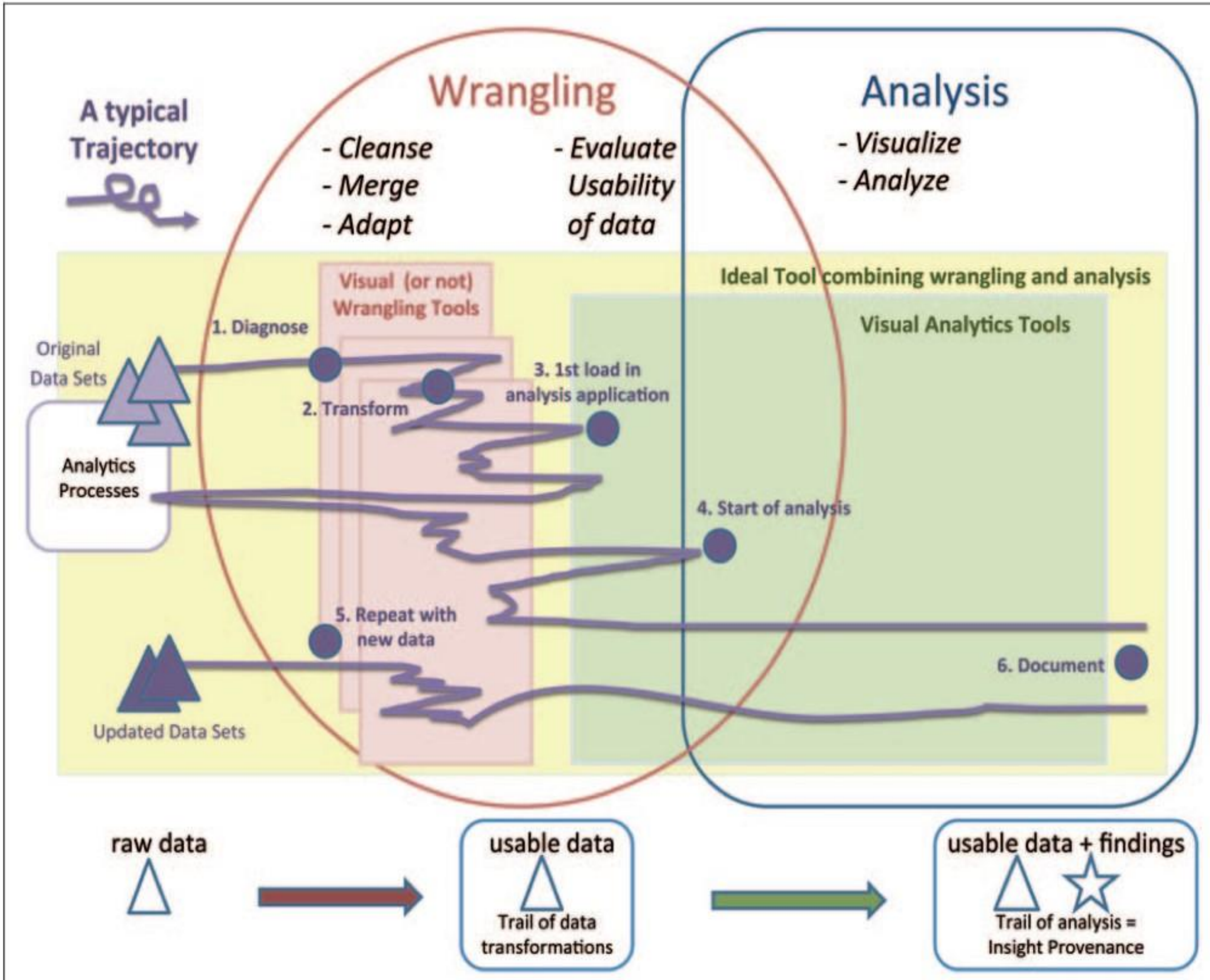
- Goal: get data into a structured form suitable for analysis
  - Variously called: data preparation, data munging, data curation
  - Also often called ETL (Extract-Transform-Load) process
- Often the step where majority of time (80-90%) is spent
- Key steps:
  - Scraping: extracting information from sources, e.g., webpages, spreadsheets
  - Data transformation: to get it into the right structure
  - Data integration: combine information from multiple sources
  - Information extraction: extracting structured information from unstructured/text sources
  - Data cleaning: remove inconsistencies/errors

OVERVIEW

- Goal: get data into a structured form suitable for analysis
  - Variously called: data preparation, data munging, data curation
  - Also often called ETL (Extract-Transform-Load) process
- Often the step where majority of time (80-90%) is spent
- Key steps:
  - Scraping: extracting information from sources, e.g., webpages, spreadsheets
  - Data transformation: to get it into the right structure
  - Information extraction: extracting structured information from unstructured/text sources
  - Data integration: combine information from multiple sources
  - **Data cleaning: remove inconsistencies/errors**

Already covered

In a few classes


A typical Trajectory
Wrangling
- Cleanse
- Merge
- Adapt
- Evaluate Usability of data
Analysis
- Visualize
- Analyze
Ideal Tool combining wrangling and analysis
Visual (or not) Wrangling Tools
Visual Analytics Tools
Original Data Sets
Analytics Processes
1. Diagnose
2. Transform
3. 1st load in analysis application
4. Start of analysis
5. Repeat with new data
6. Document
Updated Data Sets
raw data
usable data
Trail of data transformations
usable data + findings
Trail of analysis = Insight Provenance

OVERVIEW

- Many of the problems are not easy to formalize, and have seen little work
  - E.g., Cleaning
  - Others aspects of integration, e.g., schema mapping, have been studied in depth

- A mish-mash of tools typically used
  - Visual (e.g., Trifacta), or not (UNIX grep/sed/awk, Pandas)
  - Ad hoc programs for cleaning data, depending on the exact type of errors
  - Different types of transformation tools
  - Visualization and exploratory data analysis to understand and remove outliers/noise
  - Several tools for setting up the actual pipelines, assuming the individual steps are programmed (e.g., Talend, AWS Glue)