Recap

- Data-savviness is the future!
- "Classical" relational databases
  - Notion of a DBMS
  - The relational data model and algebra: bags and sets
  - SQL Queries, Modifications, DDL
  - Database Design
  - Views, constraints, triggers, and indexes
  - Query processing & optimization
  - Transactions
- Next: Semi-structured data and document stores

We've focused on relational databases…

- Typical approach to using relational databases:
  - Declare a schema, declare constraints (via DDL)
    - Schema: collection of relations, each w/ a fixed set of atomic attributes
  - Then start making modifications to the data
    - Adding, deleting, updating tuples
  - Rarely, if ever, does the schema change
    - Can do so via the ALTER TABLE
    - These are usually very expensive operations
  - Data on the other hand, changes very frequently
- This is one reason why relational databases are often viewed to be "rigid"

Classical Database Assumption 1

- The assumption that we will be trying to target today is:
  - *Data systems should manage data using a well-defined schema comprising of relations, each with a fixed set of atomic attributes, with the schema changing rarely*

- Q: why might we want to remove this assumption? What are ways in which we may want to change it?

Why Change the Assumption?

- *Data systems should manage data using a well-defined schema comprising of relations, each with a fixed set of atomic attributes, with the schema changing rarely*

- Flexibility may be valuable, especially in modern internet applications where requirements may change constantly
- It may be more "natural" to store data in specific formats over others
- It may be more compact in certain cases

Ways to Relax the Assumption

- *Data systems should manage data using a well-defined schema comprising of relations, each with a fixed set of atomic attributes, with the schema changing rarely*

  - Sub-assumption 1: There is a fixed set of attributes
  - Sub-assumption 2: Attributes are atomic (i.e., they can't be a set or array, for example)
  - Sub-assumption 3: Attributes can't be nested (i.e., attributes can't contain relations!)

Enter Semi-Structured Data

- Semi-structured data is a data representation or data model that is less "rigid" than structured data (the focus of classic databases)
- Removes all three of the following assumptions
    1. There is a fixed set of attributes
        - *Flexible schema*
    2. Attributes are atomic
        - *Attributes can be multivalued*
    3. Attributes can't be nested
        - *Attributes can be nested*
- Exemplified by two popular formats, XML and JSON
    - Start with JSON first

JSON

- Stands for JavaScript Object Notation
- Textual representation of complex data types (but also can be stored in binary for more compact representation)
- Widely used for transmitting data between applications and storing complex data
  - Especially widely used in internet applications, transferring data to the browser
  - Manipulated easily by JavaScript within the browser

JSON Example: My own!

- I use JSON to store the list of all of my papers on my website:

List of papers

Each paper has a list of attributes

```
papers=[
     {
         "title":"Benchmarking Spreadsheet Systems",
         "link": "https://people.eecs.berkeley.edu/~adityagp/papers/spreadsheet_bench.pdf",
         "authors": "Sajjadur Rahman, Kelly Mack, Mangesh Bendre, Ruilin Zhang, Karrie Karahalios, Aditya Parameswaran",
         “conf”:"SIGMOD Int'l Conf. on Management of Data",
         "location": "Portland, USA",
         "date": "June 2020",
     },
     {
         "title":"ShapeSearch: A Flexible and Efficient System for Shape-based Exploration of Trendlines",
         "link": "https://arxiv.org/abs/1811.07977",
         “authors”:"Tarique Siddiqui, Zesheng Wang, Paul Luh, Karrie Karahalios, Aditya Parameswaran",
         “status”:"Unpublished Manuscript",
         "date": "June 2020",
     },
    …
]
```

Notice the “**key : value**” notation, with commas to separate KV pairs

JSON Example: My own!

- I use JSON to store the list of all of my papers on my website:

```
papers=[
     {
         “title": "Benchmarking Spreadsheet Systems",
         "link": "https://people.eecs.berkeley.edu/~adityagp/papers/spreadsheet_bench.pdf",
         "authors": "Sajjadur Rahman, Kelly Mack, Mangesh Bendre, Ruilin Zhang, Karrie Karahalios, Aditya Parameswaran",
         “conf":"SIGMOD Int'l Conf. on Management of Data",
         "location": "Portland, USA",
         "date": "June 2020",
     },
     {
         “title": "ShapeSearch: A Flexible and Efficient System for Shape-based Exploration of Trendlines",
         "link": "https://arxiv.org/abs/1811.07977",
         “authors":"Tarique Siddiqui, Zesheng Wang, Paul Luh, Karrie Karahalios, Aditya Parameswaran",
         “status”:"Unpublished Manuscript",
         "date": "June 2020",
     },
    …
]
```

How might I represent this in the relational model?

Representing In the Relational Model

- If I knew in advance the set of attributes, I could do one giant relation:
  - (title, link, authors, conf, location, date, status)
    - conf, location hold only for published papers
    - status holds only for unpublished papers
  - Q: Downsides:
    - conf, location may be empty for unpublished papers
    - status will be empty for published papers
- Or I could split it up into two:
  - (title, link, authors, date, status) for unpublished papers
  - (title, link, authors, conf, location, date) for published papers
  - Q: Downsides:
    - Need to union if I want to combine info across relations
- Or I could split it up into three:
  - (title, link, authors, date) for all papers
  - (title, conf, location) for published papers
  - (title, status) for unpublished papers
  - Q: Downsides:
    - Need to join/union if I want to combine info across relations

Why Did I Use JSON?

- Flexibility in set of attributes
  - Can add a new attribute for the new tuples without changing others
- Self-describing
  - The key-value format forces me to list what I'm talking about explicitly
  - Making it more human-readable
- JSON can be easily parsed within JavaScript in the browser for rendering
- This is very small data! I have <100 papers

Another JSON Example

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

How might I represent this in the relational model?

- Types are: integer, string, real, and
  - Objects, denoted by { } are key-value maps, i.e., sets of (attribute name, value) pairs
  - Arrays, denoted by [] are key-value maps, indexed by the offset, e.g., the second child is Eduard Einstein

Another JSON Example

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

How might I represent this in the relational model?

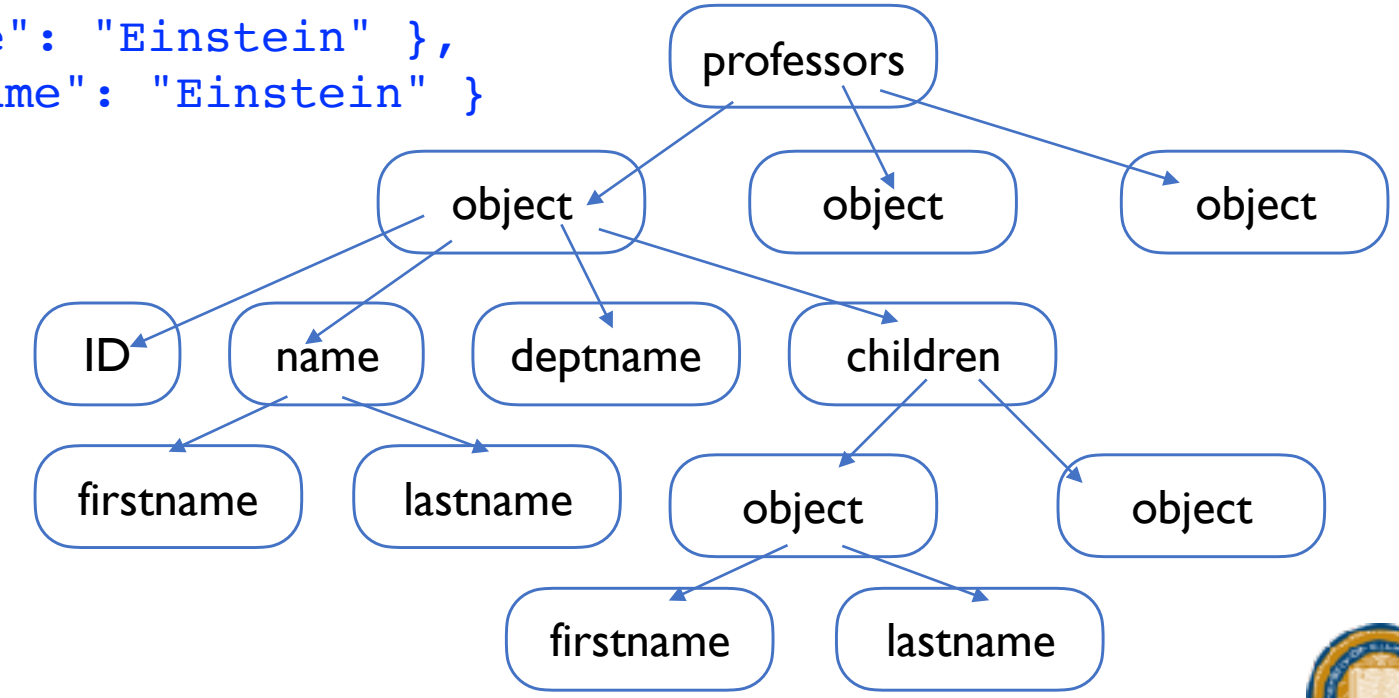
- Flattened (denormalized) representation: (ID, firstname, lastname, deptname, child)
  - Q: What is bad about this representation?
    - Repeated information for each child
- Normalized representation (ID, firstname, lastname, deptname) (ID, child)
  - Q: What is bad about this representation?
    - Unnecessary joins
- Nested JSON representation avoids unnecessary joins and repeated information!

Another JSON Example

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

JSON is a nested or hierarchical representation of data

Can go many “levels” deep!



Pros for JSON

- Flexibility in attributes
- Avoids unnecessary joins (everything in "one place")
- Easier to read

- JSON seems magical! Let's take another example to understand its issues …

Our Familiar Film-Actor Dataset

- We can represent our film-actor data in JSON, centered on Films or Actors
- Say we decide to center it on Films:
  - JSON: [list of film information]
  - Each film = { …. film attributes …., [list of actor information] }
  - Each actor = { … actor attributes …}
  - Q: Do we see any issues?
  - Each actor's attributes will be represented as many times as the number of movies they star in!
    - Update anomalies, deletion anomalies, redundancy ….
- Semi-structured data models have the same issues as denormalized data!
- Another issue: if I only wanted to look up information about actors, I need to go through the entire film information even though I don't need to.

- One way to avoid both of these problems is to factor out the actor information into a separate JSON file
  - But we might as well use structured data at that point

The pros/cons for JSON

Pros:

- Flexibility in attributes
- Avoids unnecessary joins (everything in "one place")
- Easier to read

Q: Cons?

- To avoid joins, JSON encourages redundancies (and therefore anomalies)
- By packing in everything into one structure, JSON make it harder to only look at portions of the data
- By being "self-describing" JSON is also harder to make compact
    - Repeated "keys" occupy space
    - However, there are binary JSON representations that make steps towards more compactness
    - We can make it even more compact if we knew that most tuples have a fixed structure
        - But this makes JSON less flexible

Document Stores: JSON-Native Data Systems

- Document Stores are data systems that primarily operate on JSON
  - Each JSON item is called a "document"
    - This is the basic unit of retrieval in a document store
  - Document stores support searching across documents
    - Essentially selections / filters
    - Also support some primitive aggregation
    - They typically don't support joins (beyond what is already done via denormalization)
    - They may support some form of primitive indexing to retrieve documents based on certain criteria
- One example popular document store is MongoDB
- Demo!

JSON in Relational Systems

- Relational database systems have also started supporting JSON as a data type
  - Declare a column to contain either json or jsonb (binary)
    - CREATE TABLE nobel_prizes (nobel json)
  - In our previous example, we will have a row per document
    - i.e., a row corresponding to prizes for a given year and field
- Allowing mixing of relational and semi-structured syntax

```
SELECT * FROM nobel_prizes
WHERE nobel @> '{"year":"1990"}';
```


XML

- XML is an older semi-structured data format (from the late 90s)
- Short for eXtensible Markup Language

```
<course>
    <course id> INFO290A </course id>
     <title> Human-in-the-loop Data Management </title>
     <dept name> I School </dept name>
     <credits> 4 </credits>
</course>
```

- Slightly different style: opening and closing tags
- Q: what does this remind you of?
- HTML: another markup language: Hyper Text Markup Language

Larger Example

```
<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>
```

Flexible
Support
For Complex
Data Types

XML Query Languages

- A lot of effort went into developing XML formalism and query languages
  - Schema specification
    - DTD (document type definition) — simpler, less powerful
    - XML Schema — complex, more powerful
  - Query languages
    - XPath: a "path" based query language — simpler, less powerful
      - recall, XML, like JSON can be represented as a tree
      - Path from root to specific node with certain characteristics
    - XQuery — complex, more powerful

Is Semi-Structured Data New?

- Turns out that's not that new…
  - In fact, semi-structured data existed prior to structured data!
  - Some of the earliest database systems used nested or hierarchical data models
- Eventually most of these were abandoned, for several reasons:
  - Complex and cumbersome to traverse these nested relations to find specific bits of data
    - (e.g., traversing a film hierarchy to find information about actors)
  - Nested model mixes physical and logical data organization
    - In the relational model, we can rearrange the data on disk and it would still be equivalent
    - We can't do the same thing for nested models, and therefore can't optimize it for queries
  - Lots of redundancy, leading to update and deletion anomalies
- Overall, it is clear that relational databases "won" in the long term!

  - See "What Goes Around Comes Around" by Hellerstein and Stonebraker

So what do we do?

- It makes sense to use document stores if our data is inherently denormalized, and:
  - We only want to operate on the denormalized data as a whole
    - We rarely want to look at pieces of the data
      - e.g., actors in the film-actor case when data is org. by film
  - We don't need the full power of SQL, e.g., no joins
  - The schema often changes, and we rarely update old data
  - Redundancy doesn't create issues
    - e.g., either not a lot of redundancy or redundant data isn't updated often
- One compelling example is to have the entire user profile within Facebook in a single JSON document
  - Keep all "user" information together
  - But this makes it harder to do join-oriented computation, e.g., find all friends of a given user who are based in Berkeley
- A promising alternative is to use JSON or XML within relational databases, giving the best of both worlds
- But, there are very good use-cases for JSON/XML…

One Excellent Use Case: Data Sharing, Exchange, and Transfer

- Transferring data across media
  - e.g., across network, across different data systems
- Sharing or exchanging data
  - e.g., on the internet, "export to JSON"

- A perfect use-case for semi-structured data
  - High emphasis on flexibility
  - Self-describing nature is highly valuable for portability

Summary

- The assumption that we tried to target today is:
  - *Data systems should manage data using a well-defined schema comprising of relations, each with a fixed set of atomic attributes, with the schema changing rarely*

- JSON/XML as a possible semi-structured data representation that is flexible, multivalued, and nested
- Document stores as a possible implementation
- Or, JSON / XML data types within relational databases