CS 677: Big Data

Data Models

Lecture 8

Today's Schedule

- Data Models
- Data Access

A Bit of History

- ~1970: relational databases
  - SQL, relational models
- ~2009: surge in popularity of "NoSQL" systems
  - Relaxed consistency, de-emphasizing transactions, new data models
- ~2012: "NewSQL" systems
  - Tabular data model, ACID support, but built on distributed principles

What's the Best Approach?

- As usual, it depends.
- If you have **small**, structured data then a relational database is your best option
  - Generally these don't scale well, though!
- If you need to scale out massively, then choose a NoSQL storage system
- If you need good scalability and the flexibility of a relational database, NewSQL might be the way to go

Representing our Data

- We've discussed quite a few systems thus far
- Why? Because when you're dealing with big data, you often design **systems** rather than **schemas**
- General-purpose databases are a good start but begin to be a bottleneck once scaling is necessary
- Choosing a data model means choosing between trade-offs

Today's Schedule

- **Data Models**
- Data Access

Data Models

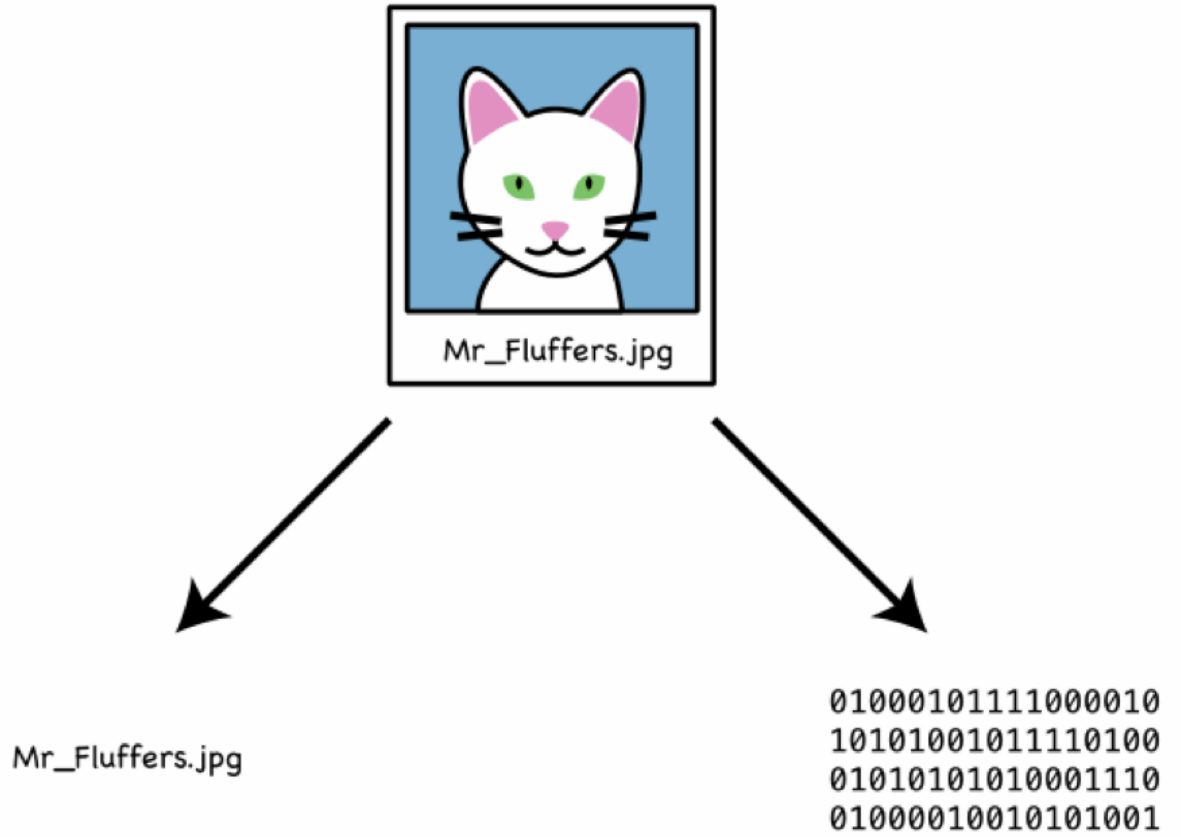
- Key-value
- Document
- Tabular
- Graph

Key-Value Stores

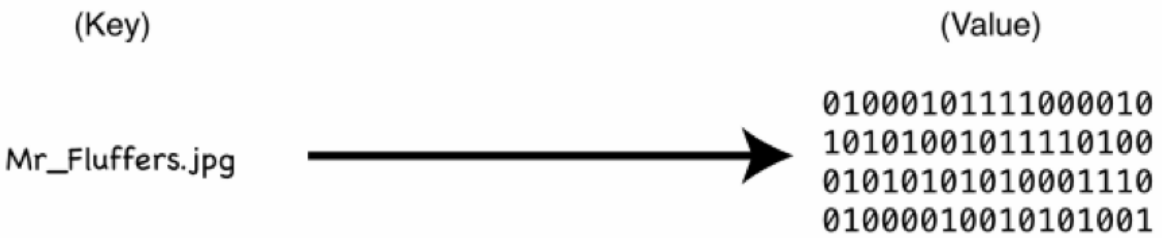
- Data model similar to a hash table
  - Flat namespace
- Simple functionality
  - Put(key, value)
  - Get(key)
  - No query/search support
- Frequent uses:
  - General (file) storage
  - Object caches
  - Common theme: you don't care what is inside the files

Key-Value Data Model

Key-Value Data Model



Key-Value Data Model



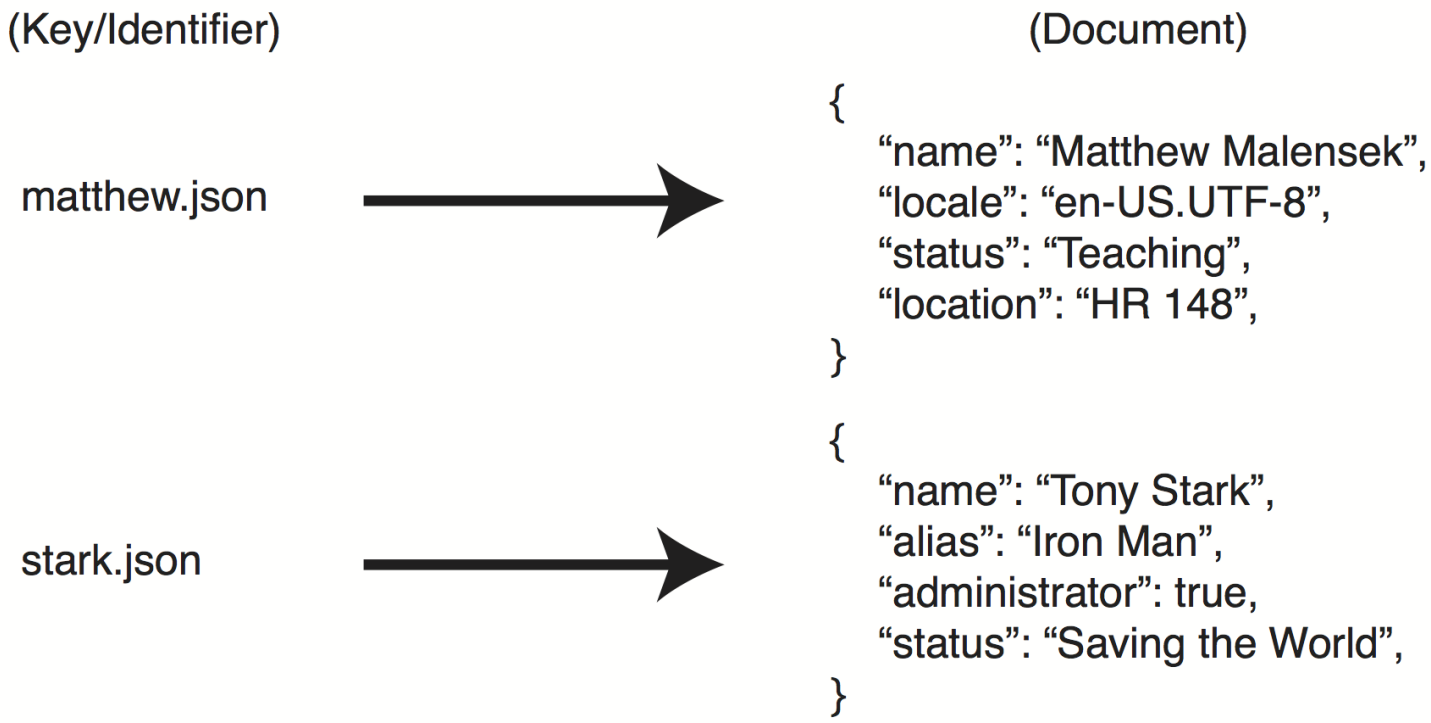
Content-Addressable Storage

- In some cases you don't want to store data with a file name or identifier
- With CAS, just use the content's hash key directly
  - put(my_file.txt) ➡ 0x123456789
- Use cases:
  - Preventing duplicate data from being stored
  - Verifying the integrity of documents
  - Pulling in file updates

Choose Key-Value When:

- You don't need to know what the files contain
- You don't need an index
  - (This can be provided separately, though)
- You won't be doing range queries over the data
- You want extreme scalability
- You want to be able to easily cache + replicate data

Document Store Data Model



Document Stores

- Often represented using JSON, YAML, etc.
- Beyond key-value semantics, document stores also allow **content-aware** searches
- Support a wide variety of data types
  - Serialization formats, multidimensional arrays
- Generally use **inverted indexes** to support queries
- Index options:
  - Domain-specific indexer
  - Automatic based on text (JSON, XML)

Other Document Types

- We are all familiar with JSON and XML
- Scientific document types:
  - NetCDF (from Unidata)
  - HDF5
  - GRIB (World Meteorological Organization)
- And of course, plain text, ODF, .doc(x)

Choose Document Storage When:

- You need the system to be aware of the file contents
- The data is mostly schemaless
  - May contain different sets of fields
    - Things change…
    - This is as “real world” as it gets
- Your application (client side or server side) expects a particular data format
  - e.g., it only operates on NetCDF files

Tabular Storage

- This is the most popular data model
- Variants:
  - Row Stores
  - Column Stores
  - DataFrames
- *Some examples*: Parquet, ORCFile, Resilient Distributed Datasets (RDDs), and many languages/frameworks have their own DataFrame implementation…

Tabular Storage

Name	Address	Phone	Birth Date
Matthew	1625 W Oak St	(970) 379-4929	2/27/22
Michelle	NULL	(327) 876-5309	11/16/81
Bob	1600 Pennsylvania Ave	(202) 456-1111	08/04/61

Row Stores

- Densely populated tables (relations)
  - No data? Insert NULL in its place
    - 😢
- Fixed set of data types
- Schema does not frequently change
- Caveats:
  - All tables must have at least one primary key column
  - Data partitioning is often explicit

Choose Row Stores when:

- You want the entire record
  - If I look up user 1398, I want to see their address, name, etc.
- You have well-defined ways to get the records you want: lookups should be straightforward
  - Try to avoid expensive distributed operations such as joins
- You **won't** be accessing a single column across all records

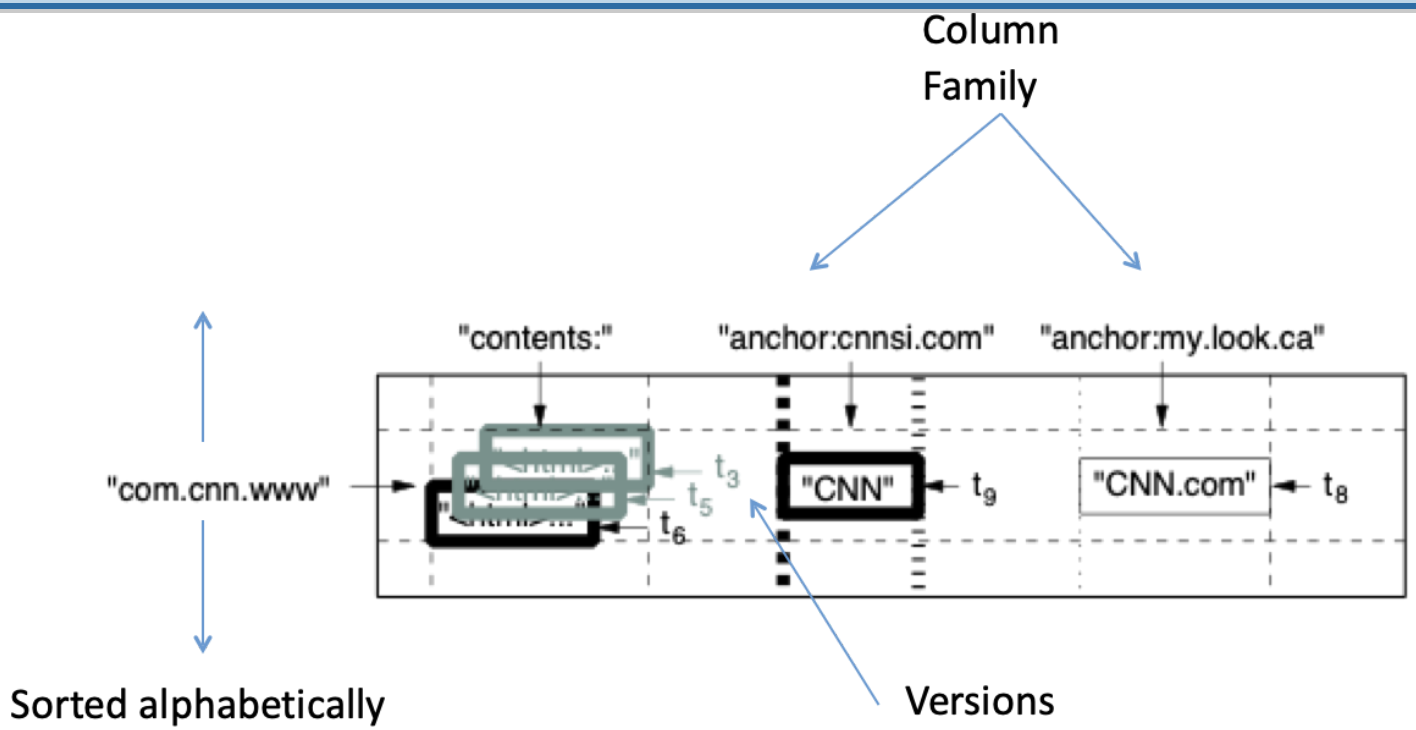
Column Stores [1/2]

- Multidimensional key-value stores
  - Values are arbitrary byte arrays
- Can be sparsely populated
- A **row key** references a set of **column families**
  - Writes under a row key are atomic
  - Keys are stored in lexicographic order to facilitate scanning across records
- Often include column-based **versioning**

Column Stores [2/2]

- Useful when analyzing one or two dimensions or features at a time
  - Optimized for selecting **columns**, not entire records
- Affords more flexibility: can allow document-style ad-hoc feature columns
  - Sparse representation
- Another way to think about column stores: they're more or less just hierarchical key value stores

Column Store Data Model



Source: Chang et al., "Bigtable: A Distributed Storage System for Structured Data"

Use Column Stores When:

- You can analyze one feature at a time
- You want to explore the relationships between features
- You need sparse representation or flexibility in what the column families contain

DataFrames

- In recent years, multidimensional data is often represented as **DataFrames**
  - Very similar to the tabular data model but designed for machine learning and data mining use cases
  - Dimensions are usually columns, rows are observations/samples/entries
- R, Pandas, Spark, etc. use this abstraction
- Schema is not as strictly defined: might be ascertained when loading/importing the data for the first time

Joins

- Joins are expensive
  - What is a 'join' again?
- Distributed joins are even more expensive
  - Computational cost + latency
- Your two options:
  - Avoid at all costs
  - Design your entire system around joins
- Why mention this? Many of the traditional database design approaches make frequent use of joins!

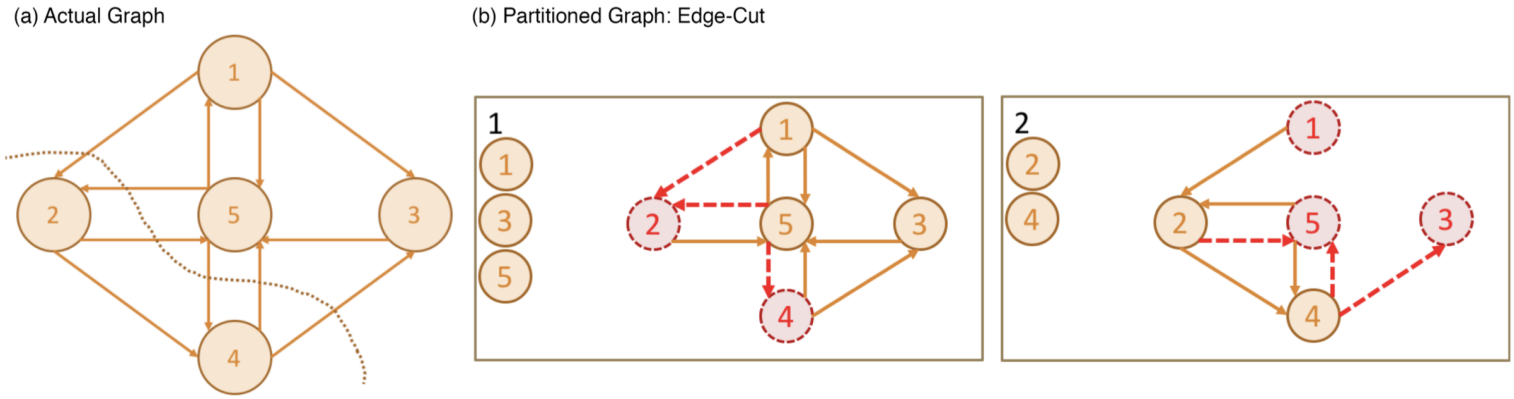
Graph Stores

- $G = (V, E)$
  - *Vertices*: nodes in the graph
  - *Edges*: links between the nodes
- Graph stores represent data as relationships modeled as a collection of vertices and edges
  - Can store data in **both** vertices and edges
  - Query via DSL or SQL
- Relational databases provide some level of graph support: links between entities via foreign keys
  - Fairly restrictive
  - **BAD** performance on large graphs

Graph Partitioning

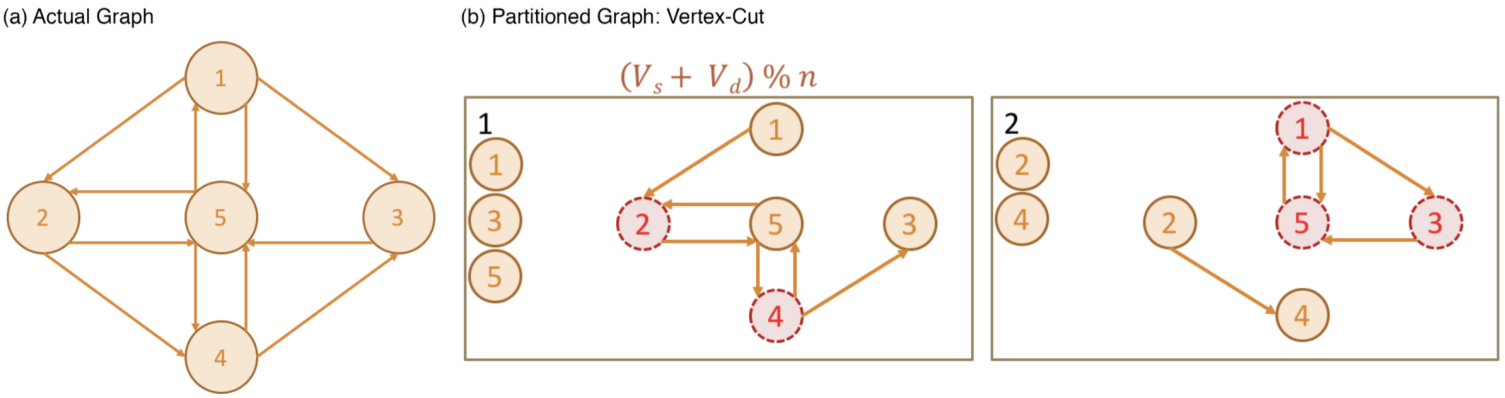
- One primary concern with graph storage systems is figuring out how to partition the graph
- A naïve approach: hash or randomize the vertex (node) placement, and then use network connections for the edges
  - This works, but adds latency
- Better: co-locate similar vertices to reduce communications to separate physical machines
  - Similar could mean: graph distance, physical (geographical) distance, etc…

Edge Cut: 2 Partitions



- Which partition? v
  - v: vertex ID; n: number of partitions
- Some edges span both partitions (red)
  - These are duplicated

Vertex Cut: 2 Partitions



- Which partition? $(v_{source} + v_{destination})$
  - E.g., (1, 3) => 4 => Partition 2
    (1, 2) => 3 => Partition 1
- Some vertices span both partitions (red) – Virtual nodes

Choose Graph Stores When:

- The relationships between data points matter
  - This applies to a surprising amount of datasets
- Mining interactions between entities
  - Facebook, Twitter, etc.
  - Social media = graphs!

    Adding a friend = edge in the graph
- You don't often need the data in tabular format
  - Transforming from a graph back to rows can be costly
    - How do you know what features are available? Not all vertices might have the same attributes

Today's Schedule

- Data Models
- **Data Access**

Easy Access: Key-Value

- First, let's talk about the easy ways to allow users to access data from our data models
- Key-value storage: user is required to provide a key
  - We can also let the user search through an index of keys, but this is usually more expensive
    - Only useful if searches are infrequent
  - Store keys in order (e.g., alphabetical, geospatial): allows targeted lookups
    - Imagine sorting file names or keys across a DHT (this is what FB's Cassandra does)

Modified Key-Value Retrieval

- For column storage, you have embedded key-value lookups within a row
- The workflow: find the row key you want, then select column families of interest, keep drilling down
  - `getRow("alice").getColFamily("address").getCol("city")`
    - `=> San Francisco`
- Many systems provide a nicer interface for doing this (domain specific languages)

Query Languages

- We can take the hierarchical key-value lookup system and translate SQL-like queries to it
  - This can be restricted to maintain good performance, or fancy features like joins can be allowed at the cost of latency
- Another approach: Apache Pig (based on Google Sawzall) is a procedural language that translates to distributed computations
- Row and column stores can support variants of SQL

Pig Wordcount

```
input_lines = LOAD '/tmp/all-pages-on-internet' AS (line:chararray);
-- Extract words from each line and put them into a pig bag
-- datatype, then flatten the bag to get one word on each row
words = FOREACH input_lines GENERATE FLATTEN(TOKENIZE(line)) AS word;

-- filter out any words that are just white spaces
filtered_words = FILTER words BY word MATCHES '\\w+’;

-- create a group for each word
word_groups = GROUP filtered_words BY word;

-- count the entries in each group
word_count = FOREACH word_groups GENERATE COUNT(filtered_words) AS count, group AS word;

-- order the records by count
ordered_word_count = ORDER word_count BY count DESC;
STORE ordered_word_count INTO '/tmp/number-of-words-on-internet';
```

Document Query: GraphQL

```
{
  human(id: "1000") {
    name
    height
  }
}
```

- Returns

```
{
  "data": {
    "human": {
      "name": "Luke Skywalker",
      "height": 1.72
    }
  }
}
```

Graph DSL: Gremlin

Apache TinkerPop: the go-to for graphs

```
// What are the names of Gremlin's friends' friends?
g.V().has("name","gremlin")
    .out("knows").out("knows").values("name")

// Get a ranking of the most relevant products for Gremlin
// given his purchase history.
g.V().has("name","gremlin").out("bought").aggregate("stash")
    .in("bought").out("bought")
        .where(not(within("stash")))
    .groupCount()
        .order(local).by(values, desc)
```

Wrapping Up

- These models influence both storage and retrieval
- Simple data models can allow increased automation; the system can do more for you when the data is simple!
- Well-defined schemas provide greater query flexibility but require more configuration
- Strong consistency is most common when records are fine-grained
  - Way easier to lock!