Last time we talked about RDBMS (Relational Database Management Systems). Recall the role for the database is to provide persistence for our applications (the ability to persist data over time) in a way that is more scalable and efficient than our initial in-memory data structures (e.g. a Map). Recall the database enables us to work with more data than will fit in main memory, implement fast and complex queries, and implement fault tolerant updates and more. Today we are going to talk about an alternate database technology, MongoDB. Mongo is a document-oriented database (in contrast to the table oriented RDBMS).
Recall our previous comparison of relational and non-relational approaches. As we noted, our focus today is MongoDB, a document-oriented database. Unlike the RDBMS where stored rows in tables, our mental model for MongoDB is JSON documents stored in a large Map (with a special _id as a key). Although unlike the actual Map container we can implement sophisticated queries across the contents of those documents.

SQL: Structured Query Language
ACID: ACID (Atomicity, Consistency, Isolation, Durability) is a set of properties of database transactions intended to guarantee validity even in the event of errors, power failures, etc.
BASE: Basically available, Soft state and Eventually consistent

Historically, NoSQL databases were BASE, and this is how they managed scale. MongoDB now claims to have ACID transactions. Difference in which you want comes down to your needs. It is okay to occasionally miss a comment as a tradeoff for low latency, but not cool to miss a financial transaction.
You would create a database for a unique application, and a collection should hold a specific thing in that application, e.g. films. Note that unlike RDBMS, instances of the noun (i.e. different documents in the collection), can have varying schemas.

BSON? Binary JSON (or binary-encoded JSON). Binary encoding makes it more compact and faster. Extends JSON with additional types like Dates, etc.
A major difference from a RDBMS is the flexible schema, a Collection doesn’t require its documents to have any particular structure, i.e. they don’t have to have to share any fields (think of it as just a big box). Although if the documents didn’t share any fields that would be code smell – are they really the same “noun”? And when there is commonality, e.g. e-mail, we can provide schema validations similar the model schema we implemented in the RDBMS practical. The key is that those validations, e.g. types, are “opt-in”.

Not imposing a schema provides lots of flexibility. Consider an address book... What are tradeoffs though?

- Fewer tools for maintaining data validity (ACID – did I leave DB in consistent state)
- Schema management moves to the application
- ...

How to approach this in RDBMS? No one “right” way. One example is single table inheritance, in which you you define your table as the union of all attributes and include a class tag that indicates what child object that row should be mapped to, e.g. StudentAddress.
Here are example queries and the tools we will use when working with Mongo. Because the Mongo documents are already JSON objects (or close enough), the ORM (which helped transform database rows to JS objects and back again) potentially plays a smaller role here (i.e. the interactions with the MongoDB driver are already pretty high-level).

This slide and this lecture is not intended as a comprehensive introduction to Mongo. Instead the goal is to give you a primer on the operations and the data modeling to prepare you for the practical where you will implement a Mongo-backed server for Simplepedia and the decision you will need to make in your project about database technology to use.

In the notes (and as part of the practical) there is a link to a mapping between SQL and MongoDB. If you are familiar with SQL it is a helping mapping, if not, it is concise summary of different types of queries you might want to perform.

https://docs.mongodb.com/manual/reference/sql-comparison/
A quick reminder of the models in our FilmExplorer application. We are going to focus on how we implement the associations in Mongo. Recall that:

- A film has a one-to-many relationship with genres (i.e. film "has many" genres)
- There is a many-to-many relationship between Users and Films via the ratings. Often called a “has many-through” association.

These terms describe a general pattern of associations and so the techniques that we will talk about will be relevant to other examples of the same kind of one-to-many or many-to-many relationships.

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
Genres is a one-to-many relationship. But for implementation purposes it is useful to note that is really a “one-to-few” relationship. That is there are a finite number of genres (~ 20) so it can’t grow unbounded. In these cases the best practice is to embed an array in the document. And since that is common practice, Mongo has explicit query support for embedded arrays.

There is, however, a maximum limit on MongoDB documents (16MB) so if your relation is truly one to many (where the objects are large) you could embed “child” references (instead of embedding objects), or if truly unbounded, you could use “parent” references (like in RDBMS “has many” / “belongs to”). In general through Mongo data models are typically “de-normalized” that is related data is embedded in one side of the relation.

```javascript
> db.films.findOne()
{ 
  "_id" : ObjectId("5a69eb43a1e7248f699794aa"),
  "id" : 135397,
  "title" : "Jurassic World",
  ...
  "genre_ids" : [28, 12, 878, 53]
}

How to query?
// All movies with genre_id 28
db.films.find({ genre_ids: 28 })

// All movies with genre_id 28 and 12
db.films.find({ genre_ids: { $all: [28, 12] } })
```

What if genres are unbounded, i.e. “one-to-very many”?
Users ⇔ Ratings ⇔ Films?

Should we embed films in users (or vice versa)? No. Models on both sides needs to stand-alone.
Do we need ratings from both users and films? Yes, e.g. show “my” ratings or films’s ratings.

Pros/Cons?

What about the many-to-many relationship between users and films? In the RDBMS context there is an establish approach to designing the schema. With Mongo there is more flexibility and so the best approach will depend on how we plan to use our data. In this case we need to access to both entities on their own (as opposed to always accessing films via users or vice-versa). And we want to access the ratings from both sides. Those needs motivates a “two-way referencing” strategy where we embed ratings along with reference into both “sides” of relationship.

What if a movie had a million ratings? Would this strategy work? No. General strategy: Eliminate references in movies, and instead just keep summary (average ratings). References stay on user side (assuming user won’t rate a million movies). Details will depend on needs of your application.

A key take home message is that unlike the RDBMS example, our data modeling approach in Mongo is going to be more application specific.

Should we use RDBMS or MongoDB?

*No right or wrong answer, just tradeoffs*

Is your data:
- Highly relational? *+RDBMS*
- Highly polymorphic? *+MongoDB*

Does your application have:
- Complex queries? *+RDBMS*
- Strong data integrity requirements? *+RDBMS*

Getting started cost:
- Uncertain initial schema *+MongoDB*

In the notes I included links to blog posts arguing different perspectives on this question.

Opinionated comparison from MongoDB’s perspective
https://www.mongodb.com/compare/mongodb-mysql

Opinionated comparison from the SQL perspective: