SQL vs. NoSQL
Really: Relational vs. Non-Relational
sort of...

<table>
<thead>
<tr>
<th></th>
<th>Relational (RDBMS)</th>
<th>Non-Relational</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>Table-oriented</td>
<td>Document-oriented, key-value, graph-based, column-oriented, ...</td>
</tr>
<tr>
<td>Schema</td>
<td>Fixed schema</td>
<td>Dynamic schema</td>
</tr>
<tr>
<td>Joins</td>
<td>Used extensively</td>
<td>Used infrequently</td>
</tr>
<tr>
<td>Interface</td>
<td>SQL</td>
<td>Custom query language</td>
</tr>
<tr>
<td>Transactions</td>
<td>ACID</td>
<td>BASE?</td>
</tr>
</tbody>
</table>

```
SELECT * FROM people
WHERE age > 25;

db.people.find(
  { age: { $gt: 25 } }
)
```

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 in that application, e.g. movies. Instances of the noun, can have varying schemas.

BSON? Binary JSON (binary encoded JSON), which stands for binary)-encoded) JSON. Binary encoding makes it more compact and faster. Extends JSON with additional types like Dates, etc.
Flexible schema

Noun/Model, e.g. “Address” ⇔ Collection

Consider an “Address Book”:

|        | cell, email, mailbox number, dorm room, ...
|--------|--------------------------------------------------------------------
| Student | cell, email, office phone, office number, ...
| Faculty | email, office phone, office number, ...
| Office (e.g. Registrar) | email, office phone, office number, ...

Some common fields, but many differences

Document-oriented storage gives the flexibility to store just the exact information needed for each object

A major difference from RDBMS is the flexible schema, a Collection doesn’t impose anything on the documents it contains, 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”?

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 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.
"Raw" queries

```javascript
db.collection.find({field: { predicate }}, { fields });
db.collection.insertOne({ ... });
db.collection.updateMany(
    { field: { predicate } },
    { $set: { fields } }
);
```

Our typical usage

https://docs.mongodb.com/manual/reference/sql-comparison/

Word of caution: The MongoDB shell does not have complete parity with the MongoDB drivers

These queries are from the shell.
A movie has a one-to-many relationship with genres (i.e. movie “has many” genres)
There is a many-to-many relationship between Users and Movies via the ratings.
Often called a “has many-through” association.

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
Genres: One-to-few

```javascript
> db.movies.findOne()
{
  "_id" : ObjectId("5a69eb43ae7248f699f94aa"),
  "id" : 135397,
  "title" : "Jurassic World",
  ...
  "genre_ids" : [28, 12, 878, 53]  // “Embedded” array
}
```

How to query?

// All movies with genre_id 28
```
db.movies.find({ genre_ids: 28 })
```

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

**What if genres are unbounded, i.e. “one-to-very many”?**

There are a finite number of genres (~20) so it can’t grow unbounded. Best practice is to embed array in document. 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 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”).
Motivates “two-way referencing” strategy where we embed ratings along with reference into both “sides” of relationship. Pros/Cons?

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.

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

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

Opinionated comparison from the SQL perspective: