6.1040: Software Design

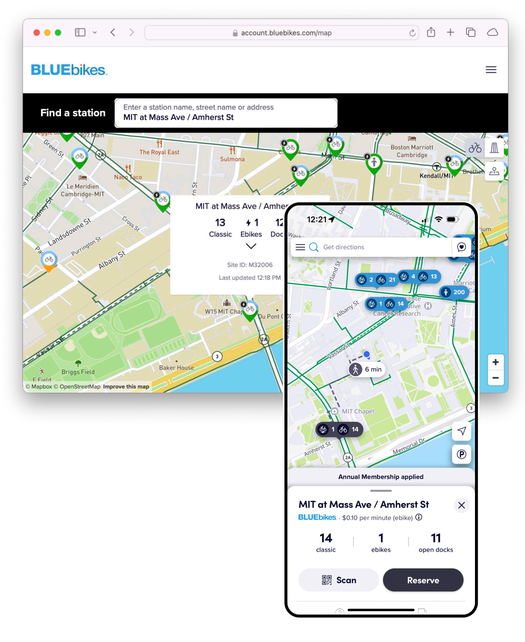
Data Design

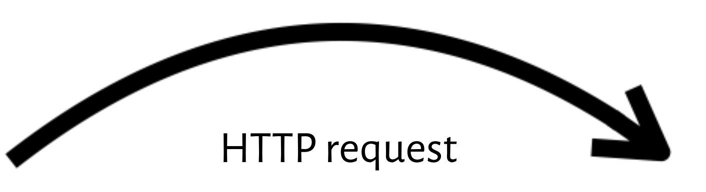
Arvind Satyanarayan & Max Goldman

with material by Daniel Jackson

Fall ’24

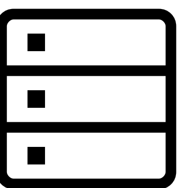
Client / Server





HTTP request

application server



process request
& build response

HTTP response

database server

Today

Different database models

Classical data modeling

Concept data modeling

Relational state

Implementation considerations

Movies playing near Back Bay East, Boston, MA

All Genres

Crazy Rich Asians
Drama/Come...

White Boy Rick
Drama/Myste...

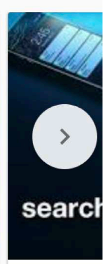
Peppermint
Drama/Thrille...

Fahrenheit 11/9
Political cine...

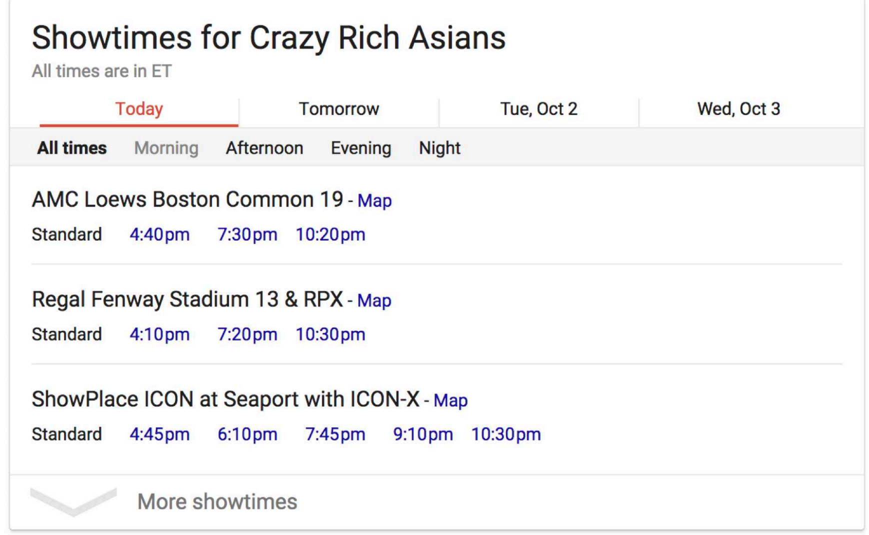
Life Itself
Drama/Roma...

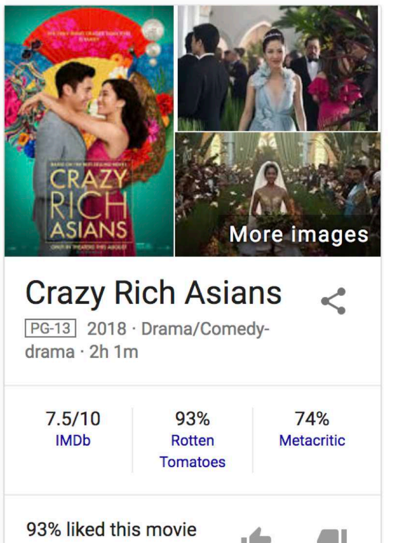
The Meg
Thriller/Fanta...

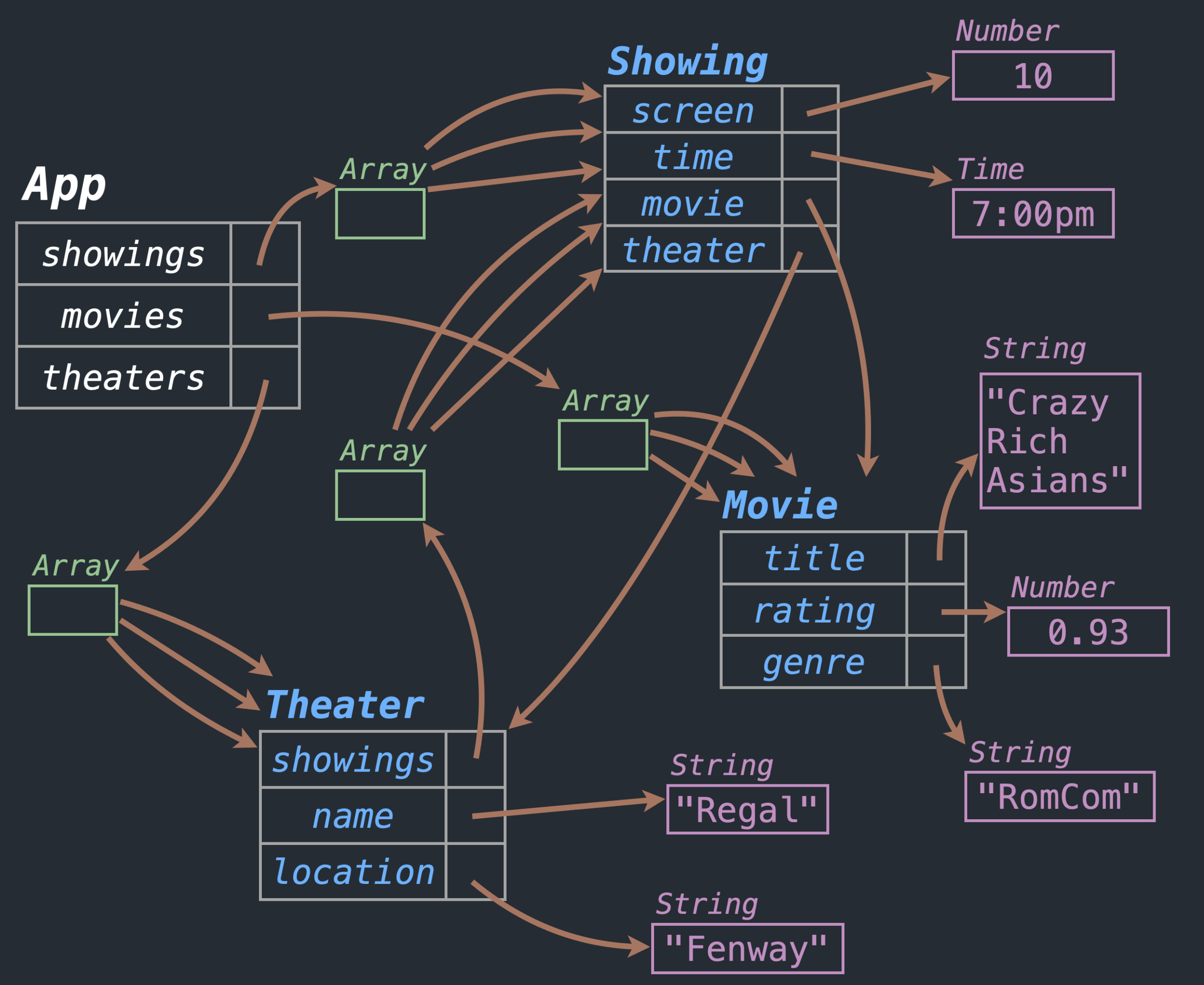
Little Women
Drama/Family



Searching
Drama/Th







Object model

Application root references collections of class instances that describe primitive data

- ✓ Quick to prototype
- ✓ Easy to experiment with arbitrary data structures
- ✕ Refactoring is difficult
- ✕ No advanced querying: only iterate over collections, follow references

Relational model

Relations (*a.k.a.* tables) of attributes (*a.k.a.* columns) and tuples (*a.k.a.* rows)

- ✓ Relational theory gives a clear path to separation of concerns with *normalization*
- ✓ Standardized query language (SQL) regardless of backend engine (MySQL, PostgreSQL, SQLite, ...)
- ✓ Many decades of research into performance and robustness (indexing, transactions, integrity, ...)
- ✗ Horizontal scaling can be a challenge
- ✗ If you like objects, there are no objects* here

Showings

id	*theater*	*screen*	*movie*	*time*
1	3	5	2	7:00pm
..	...	...	...	

Theaters

id	*name*	*location*
...	...	...
3	"Regal"	"Fenway"

Movies

id	*title*	*rating*	*genre*
...	...	...	
2	"Crazy Rich Asians"	"PG-13"	"RomCom"

Showings

id	theater	screen	movie	time
1	3	5	2	7:00pm
..	...	...	...	

Theaters

id	name	location
...	...	...
3	"Regal"	"Fenway"

Movies

id	title	rating	genre
...	...	...	
2	"Crazy Rich Asians"	"PG-13"	"RomCom"

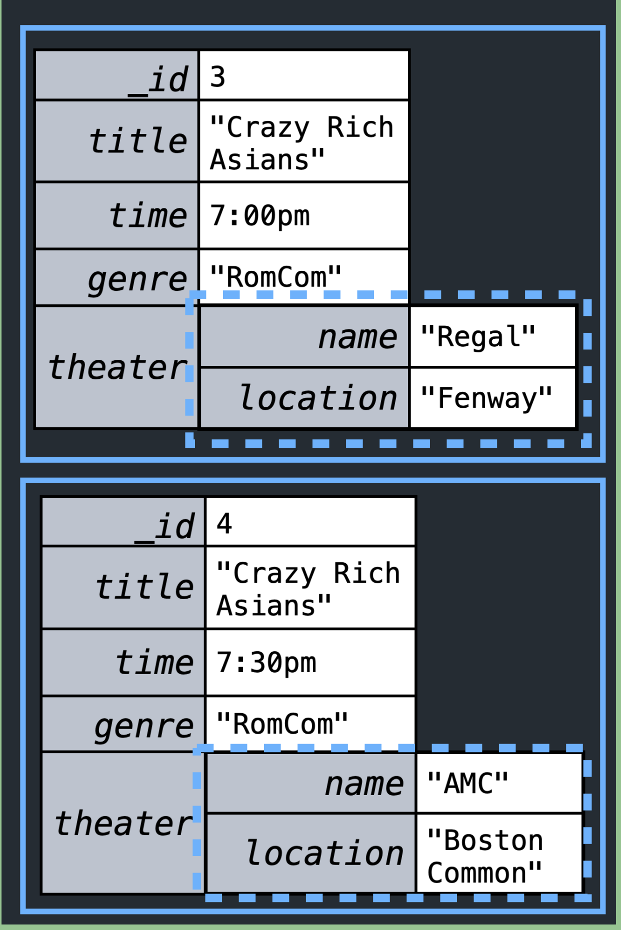
SQL: Structured Query Language

```
SELECT title, name, location, time
FROM showings
 JOIN theaters ON (showings.theater = theaters.id)
 JOIN movies ON (showings.movie = movies.id)
WHERE movies.genre = "RomCom";
```

JOIN is (a more flexible version of) the same relational join operator we discussed earlier

WHERE is a *filter*

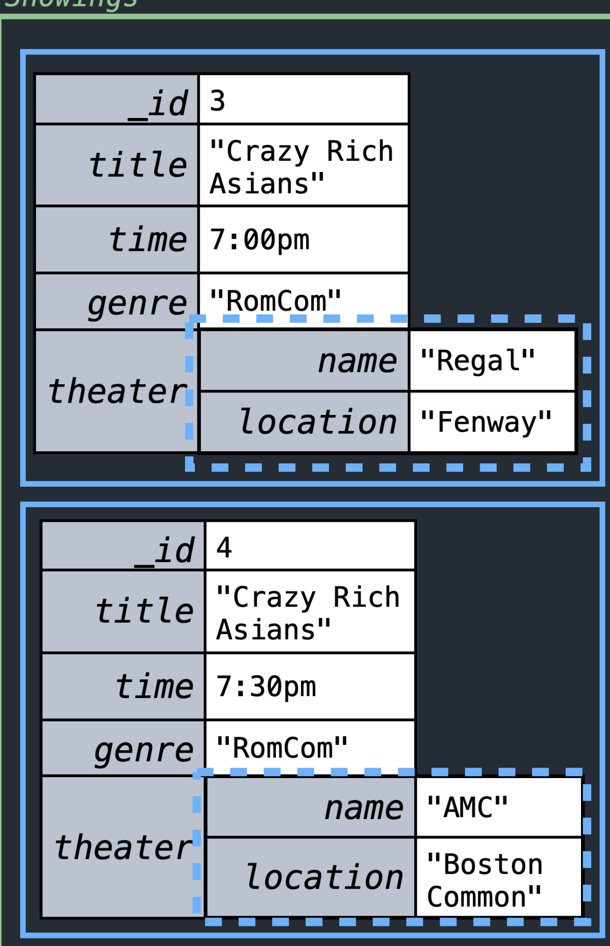
SELECT is a *map* (in this case, a *projection*)



Document model

Collections of nested documents

- ✓ Quick to prototype with JSON objects
- ✓ Easy to experiment with arbitrary data structures
- ✓ Pattern matching by document structure
- ✓ Horizontal performance (many less-powerful servers instead of one very powerful server)
- ✕ No standardized query language
- ✕ To query across collections, either: use a DB-specific API, or write code at the application level
- ✕ With embedded documents, easy to make poor design decisions



NoSQL:
Not SQL -or- Not *Only* SQL

Document databases like MongoDB

also

Graph databases

Key-value databases

and others

MongoDB

a NoSQL document database

MongoDB "CRUD" operations

Create

```
db.showings.insertOne({...})
db.showings.insertMany([{...}, {...}, ...])
```

```
{
  "_id": new ObjectId(),
  "title": "Crazy Rich Asians",
  "genre": "RomCom",
  "showtime": new Date("2022-10-07 15:30"),
  "theater": {
    "name": "AMC",
    "location": "Boston Common"
  }
}
```

Documents are JSON-like structures (BSON) that support some additional datatypes, *e.g.* `Date`

Every document has a unique `_id` of type `ObjectId` generated by MongoDB

MongoDB "CRUD" operations

Create

```
insertOne({...})
insertMany([{...}, {...}, ...])
```

Read

```
db.showings.findOne({...})
db.showings.find({...})
```

```
{ "title": "Oppenheimer" }

{ "theater.name": "AMC" }

{
  "title": "Oppenheimer",
  "theater.name": "AMC"
}
```

Query filters specify the document-matching predicate for a read, update, or delete

```
{ "$or": [
    { "title": "Oppenheimer" },
    { "theater.name": "AMC" }
] }

{ "theater.name": {
  "$in": [ "AMC", "Regal" ]
} }

{ "showtime": {
  "$lte": new Date("2024-09-25")
} }
```

MongoDB “CRUD” operations

Updates specify fields to change

Create

```
insertOne({...})
insertMany([{...}, {...}, ...])
```

Read

```
findOne({...})
find({...})
```

Update

```
updateOne({...}, {"$set": {...}})
updateMany({...}, {"$set": {...}})
replaceOne({...}, {...})
```

Delete

```
deleteOne({...})
deleteMany({...})
```

Multiple collections *vs.* embedded documents

```
db.theaters.insertOne({
  "_id": <1>,
  "name": "AMC", ...
});

db.movies.insertOne({
  "_id": <3>,
  "title": "Oppenheimer", ...
});

db.showings.insertOne({
  "_id": <5>,
  "theater": <1>,
  "movie": <3>,
  "showtime": new Date(...)
});
```

vs.

```
db.movies.insertOne({
  "_id": <7>,
  "title": "Oppenheimer",
  "showings": [
    {
      "theater": {"name":"AMC", ...},
      "showtime": new Date(...)
    }
  ]
});
```

Multiple collections *vs.* embedded documents

```
const amcs = db.theaters.find({
  "name": "AMC"
});
const oids = amcs.map(t => t._id);
const movies = db.movies.find({
  "theater": { "$in": oids }
});
```

-or-

Write a MongoDB *aggregation pipeline*…

stages in the pipeline can perform *map-* and *filter-*like operations

a `$lookup` stage performs a *join!* (recall "not only SQL")

vs.

```
const movies = db.movies.find({
  "theater.name": "AMC"
});
```

Some questions to ask:

1. How many documents are you embedding in a single parent?
2. Does the embedded document relate to other documents?
3. Will you have a need for the embedded document without the parent, or *vice versa*?

Three database models

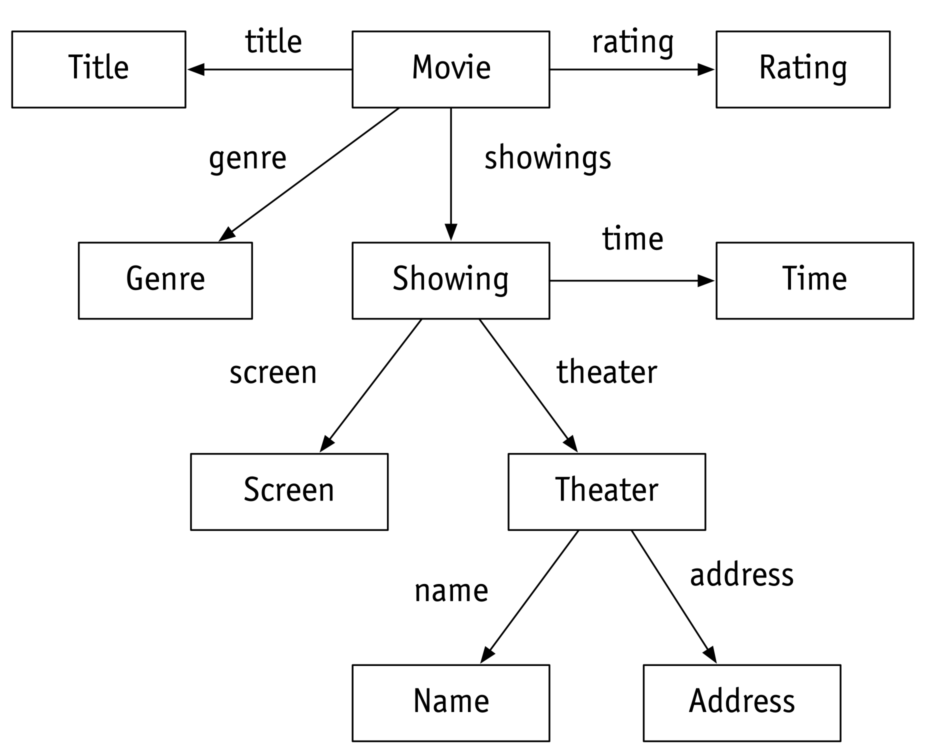
Object oriented model: references

Relational model: tables of tuples

Document collection model: (nested) documents

Designing a database: the classic approach

Step 1: identify entities and relationships



... and draw a graph

Boxes are **sets**, arrows are **relations**

Simple semantics

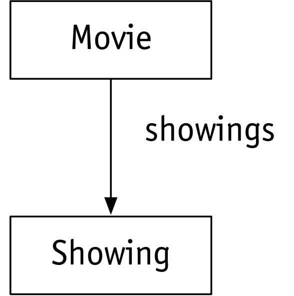
Representation-independent

Relations are predicates on 2-tuples
e.g. (Barbie, Fantasy) **in** genre

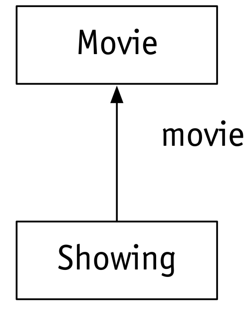
Diagram shows us possible navigations
... but*!*

Designing a database: the classic approach

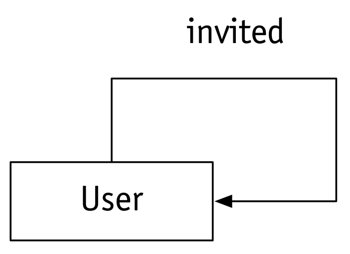
A common confusion: arrow direction



Arrow direction is
NOT navigation and
NOT containment



Can switch direction,
so long as we interpret
the relation consistently



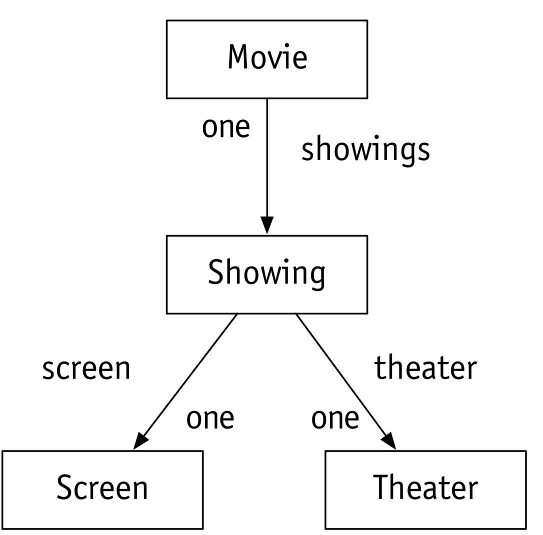
Matters for homogeneous
relations where we can
easily interpret wrong:

(alice, bob) in invited

Did Alice invite Bob,
or did Bob invite Alice?

Designing a database: the classic approach

Step 2: add multiplicities



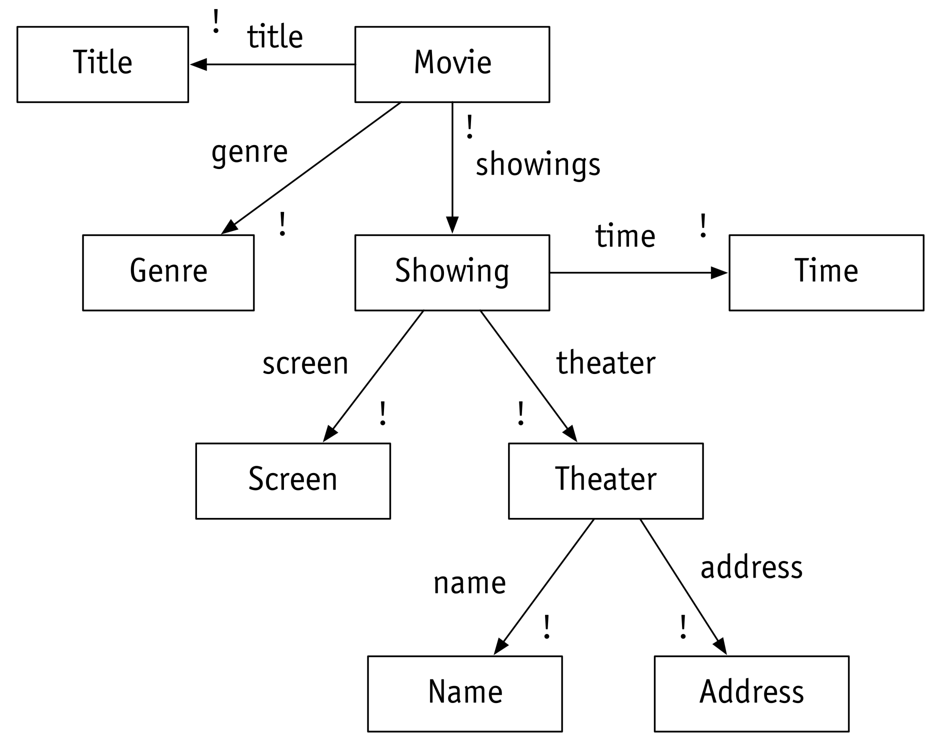
Mutiplicities tell you:
how many on that end of the arrow?

≥ 0 **set**, *the default*
≥ 1 **some**, **+**
≤ 1 **opt**, **lone**, **?**
= 1 **one**, **!**

(Many other notations, see xkcd.com/927)

Designing a database: the classic approach

Step 3: transform to... a relational database schema



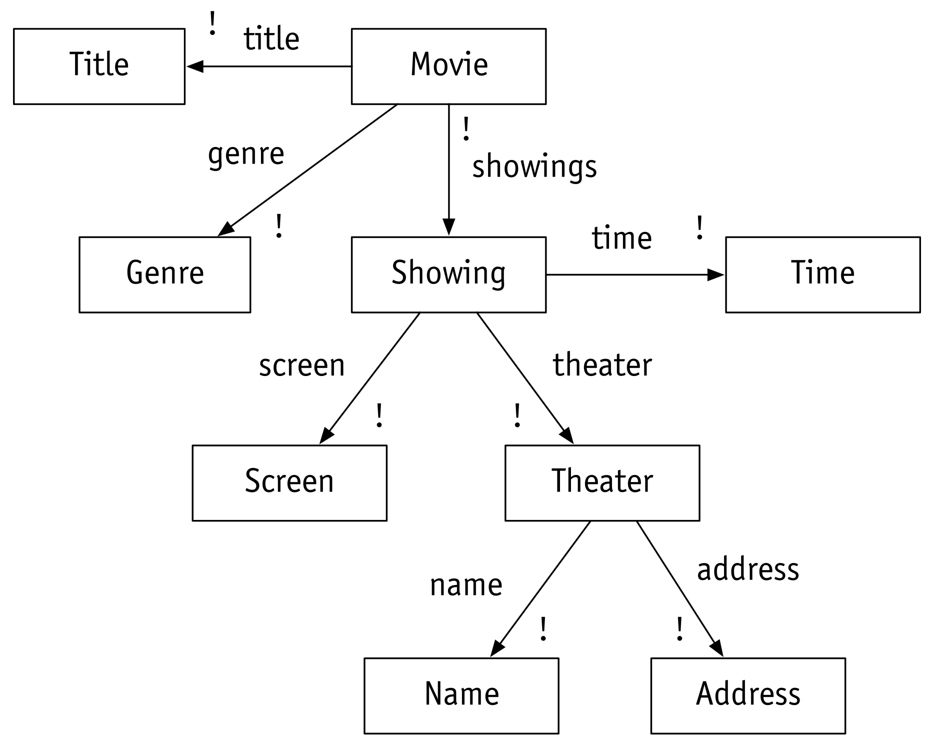
movies?

id	*title*	*showings?*
1	Crazy Rich Asians	42, 43

Constraint: no set-valued columns*

Designing a database: the classic approach

Step 3: transform to… a relational database schema

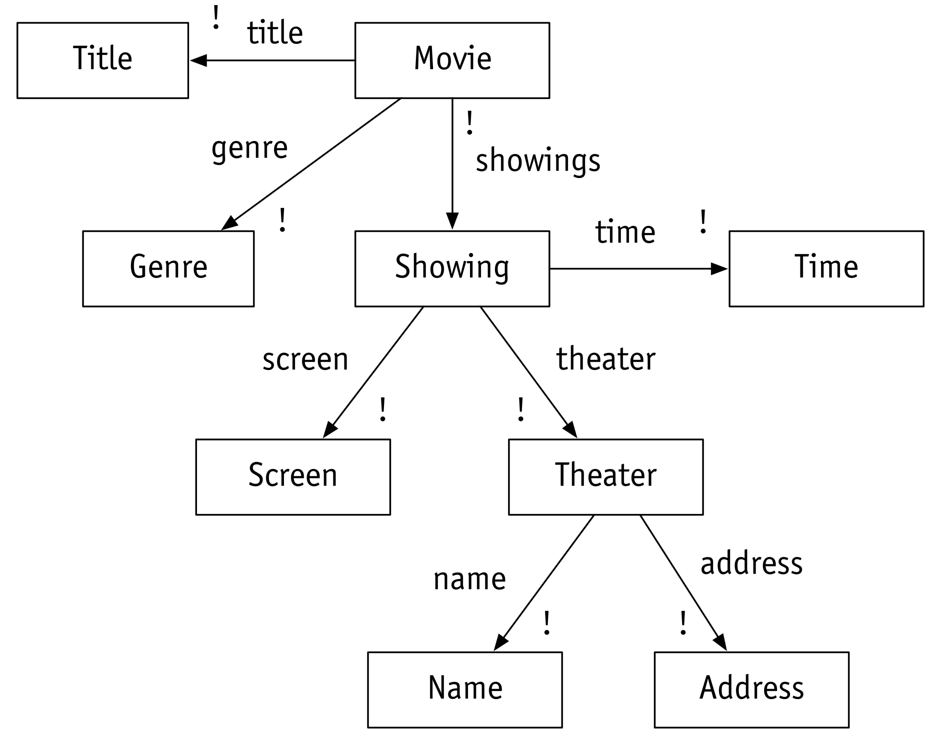


CREATE TABLE movies
(id int, title text, genre text);
CREATE TABLE showings
(id int, movie int, screen int,
theater int, time timestamp);

Constraint: no set-valued columns*

Designing a database: the classic approach

Step 3: transform to... a relational database schema



movies

id	*title*	*genre*
1	Crazy Rich Asians	RomCom
2	Barbie	Fantasy

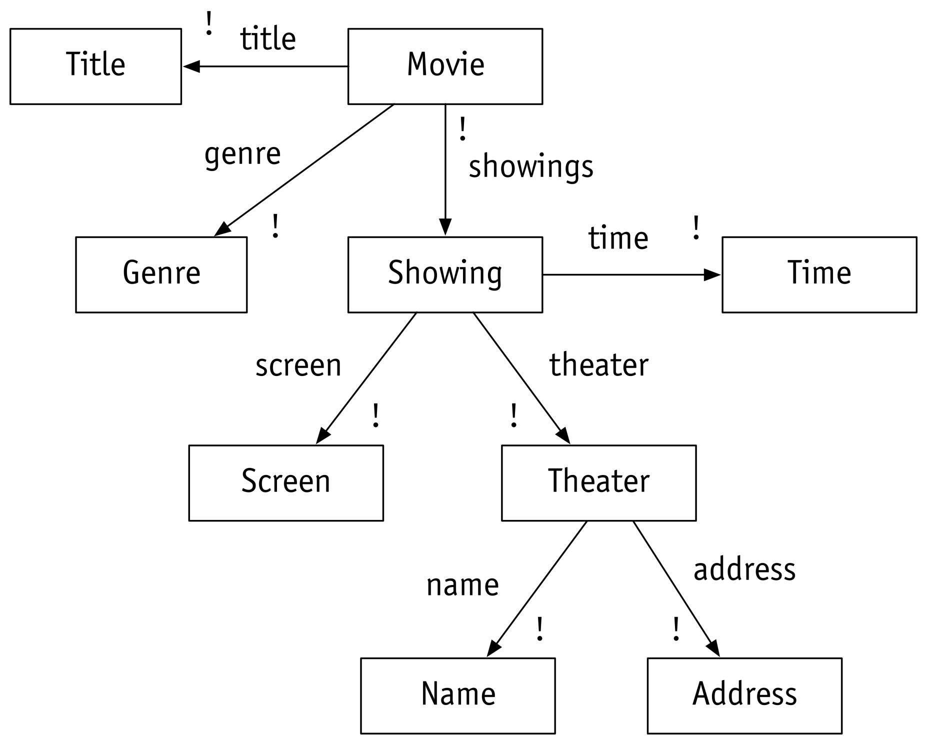
showings

id	*movie*	*screen*	*theater*	*time*
42	1	2	35	3:00pm
43	1	1	23	7:00pm

Constraint: no set-valued columns*

Designing a database: the classic approach

Step 3: transform to... an object oriented schema



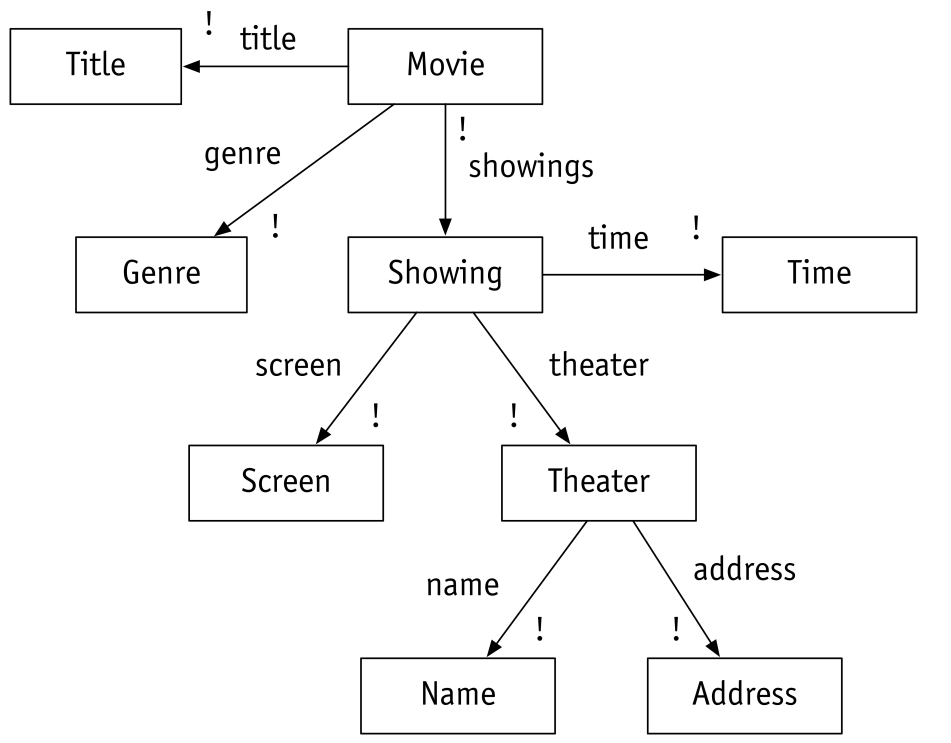

```
class Movie {
  title: string;
  genre: Genre;
  showings: Array<Showing>;
}

class Showing {
  screen: number;
  theater: Theater;
  time: Date;
}
```

Constraint: queries must follow fields

Designing a database: the classic approach

Step 3: transform to… an object oriented schema



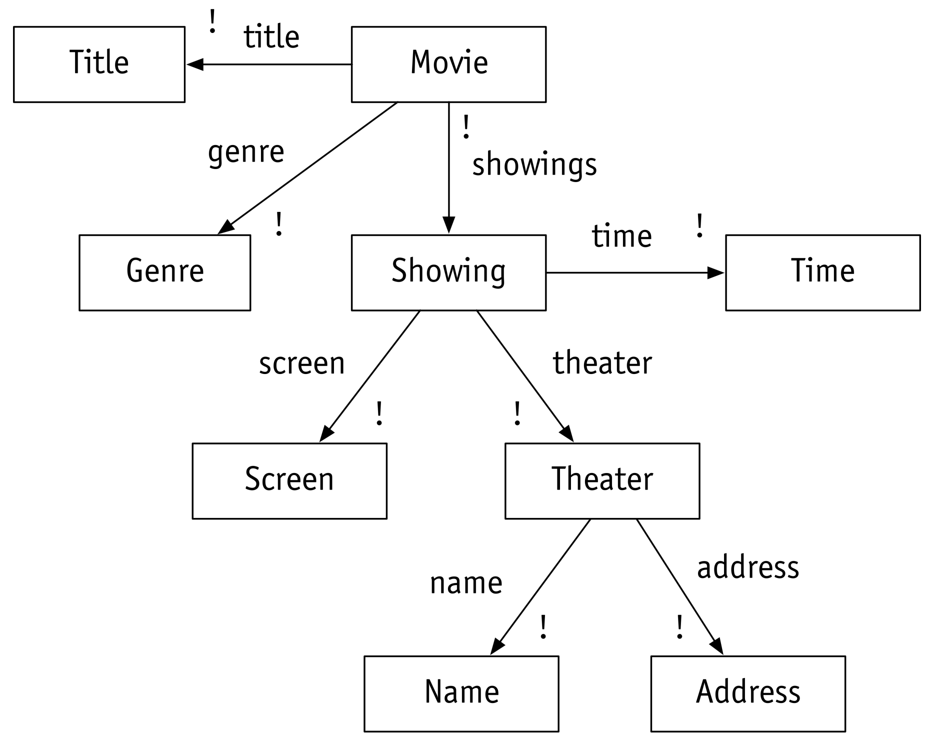

```
class Movie {
  title: string;
  genre: Genre;
  showings: Map<number,
              Array<Showing>>;
}

class Showing {
  screen: number;
  theater: Theater;
  time: Date;
}
```

Constraint: queries must follow fields

Designing a database: the classic approach

Step 3: transform to… a document collection schema



In most NoSQL databases, collections do not have a fixed schema!

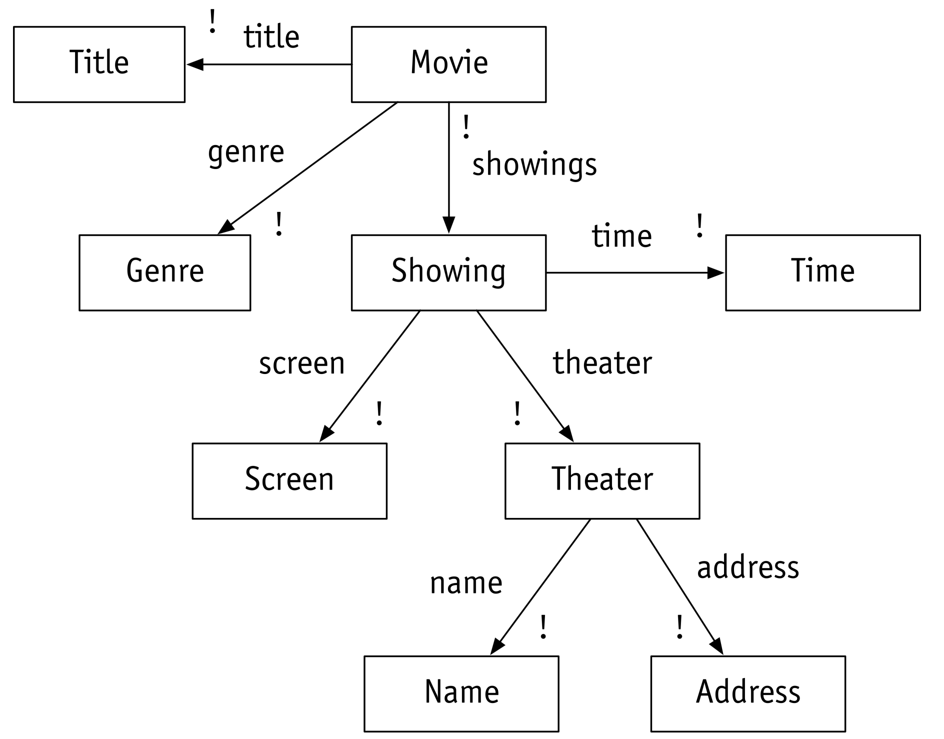
We will use TypeScript interfaces

```
interface Showing {
  _id: ObjectId,
  title: string,
  time: Date,
  ...
}
```

Constraint: if mutable, application must keep embedded docs consistent

Designing a database: the classic approach

Step 3: transform to… a document collection schema



showings

id	1
title	"Crazy Rich Asians"
time	7:00pm
genre	"RomCom"
screen	2
theater	*name*: "AMC" *address*: "401 Park Drive"

Constraint: if mutable, application must keep embedded docs consistent

Designing a database

Schema design considerations

What is possible to represent?

e.g. in a relational database with scalar fields, is the multiplicity correct?

What is the cost of queries?

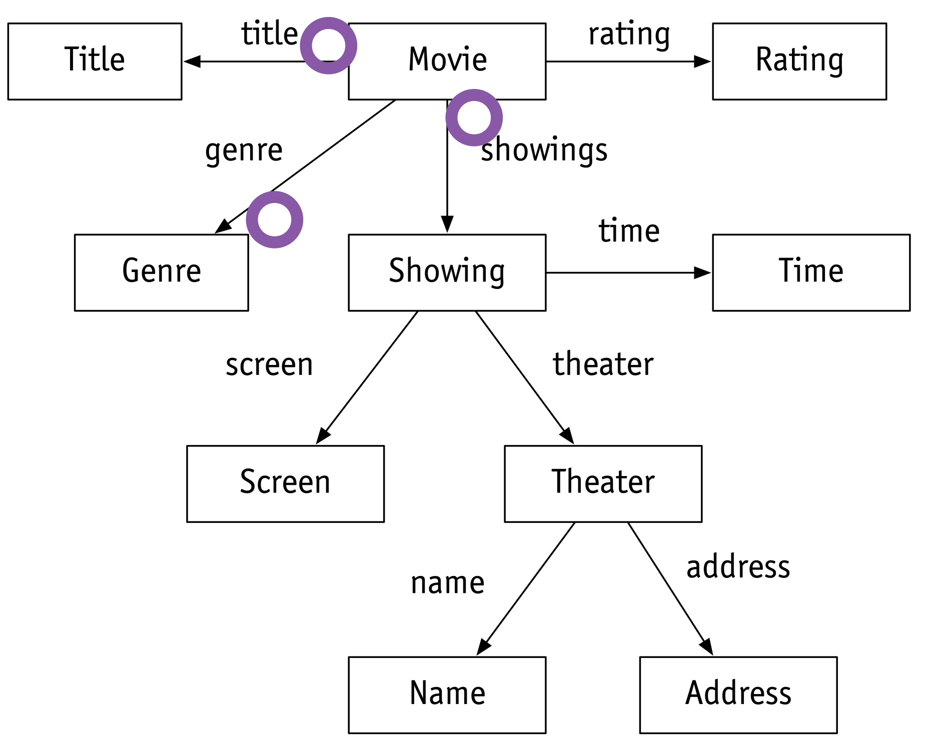
e.g. cost of relational joins mitigated by indexes

What is the cost of updates?

e.g. locking a table/object/document to prevent race conditions,
or keeping embedded documents consistent

Mutiplicities matter

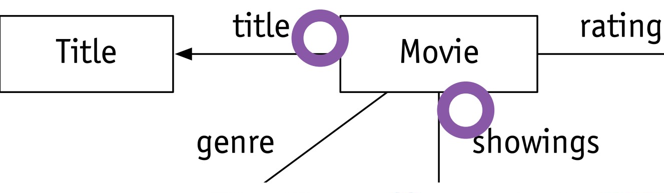
What are these multiplicities?



- ≥ 0 **set**, *the default*
- ≥ 1 **some**, **+**
- ≤ 1 **opt**, **lone**, **?**
- $= 1$ **one**, **!**

Mutiplicities matter

What are these multiplicities?



Barbie / Genres

10 Films With The Same Title That Are Not The Same Movie

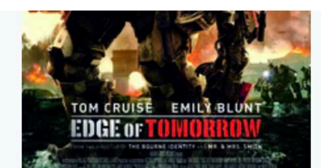
- 'Night Moves' (1975) and (2013) ...
- 'Missing' (1982) and (2023) ...
- 'Twilight' (1998) and (2008) ...
- 'Possess
- 'Rush' (1

Two chilling, bold, mesmerizing, futuristic detective thrillers.

Ridley Scott's visually stunning *Blade Runner* set a new benchmark for science fiction upon its release in 1982. In 2017, director Denis Villenueve did the unthinkable with *Blade Runner 2049*, crafting an atmospheric and riveting sequel that is not only worthy of the original, but may actually surpass it. See them back to back in this special double-feature and decide for yourself!

Edge of Tomorrow[a] is a 2014 American science fiction action film and written by Christopher McQuarrie and the writing team of Jez ar Butterworth, loosely based on the Japanese light novel *All You Need* Sakurazaka. Starring Tom Cruise and Emily Blunt, the film takes pla of Europe is occupied by an alien race. Major William Cage (Cruise) with no combat experience, is forced by his superiors to join a landir aliens, only to find himself experiencing a time loop as he tries to fin invaders. Bill Paxton and Brendan Gleeson also appear in supportin

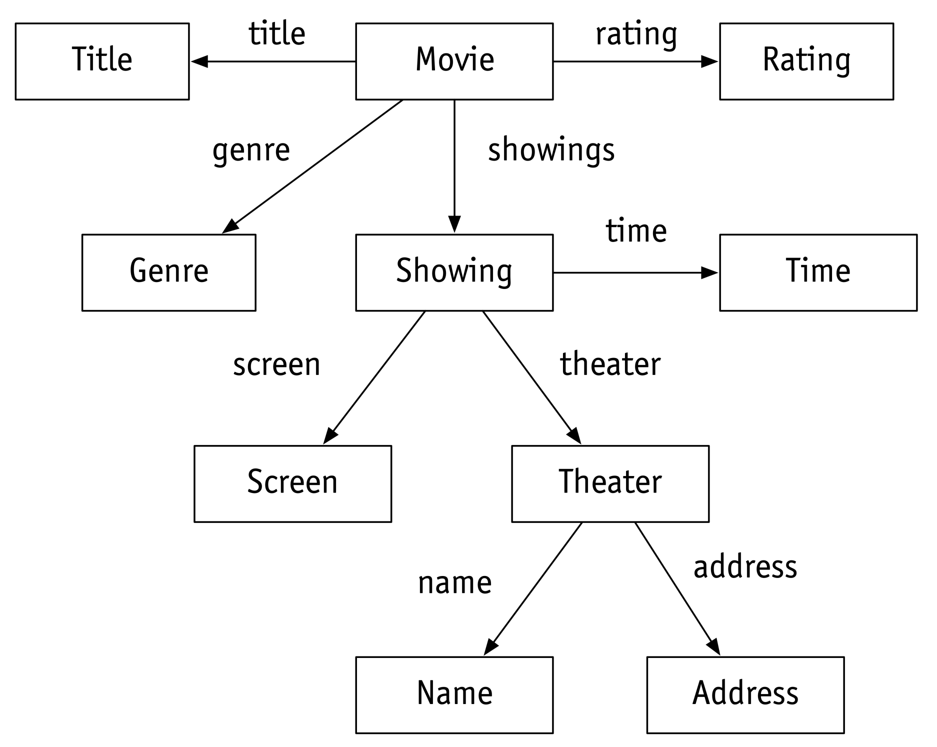
In late 2009, 3 Arts Entertainment purchased the rights to *All You Need Is Kill* and sold a spec script to the American studio Warner Bros. The studio produced *Edge of Tomorrow* with the involvement of 3 Arts, the novel's publisher Viz Media, and Australian production company Village Roadshow. Filming began in late 2012, taking place in England: at WB Studios in Leavesden, outside London, and other locations, such as London's Trafalgar Square and the coastal Saunton



Notes [edit]

a. ^ Later marketed as ***Live Die Repeat: Edge of Tomorrow***[6]

Challenges in the classic approach



What data go in the model?

Step 1 was to draw this entire graph

Where do we start?

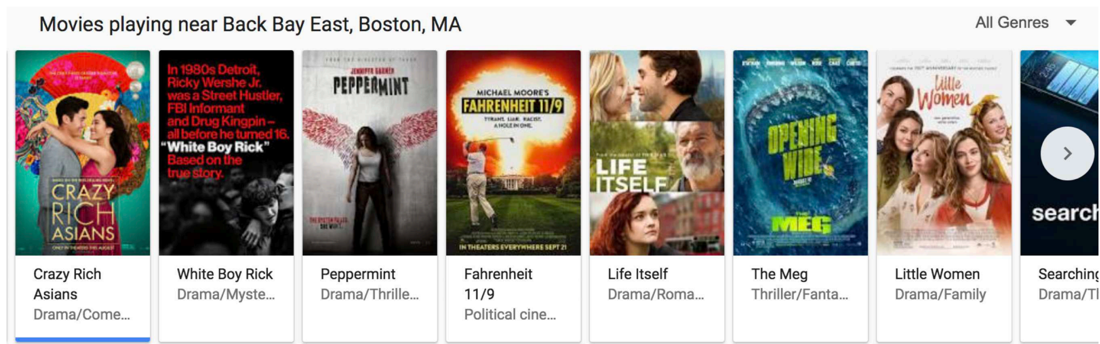
Where do we stop?

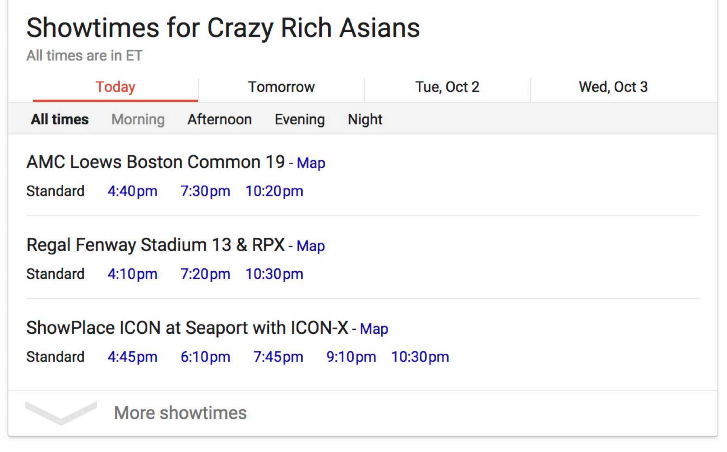
Where is the modularity?

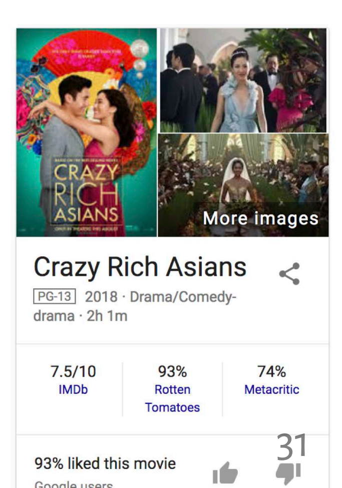
How can we reuse modeling within or across systems?

Designing a database: the concept approach

Step 1: identify the concepts

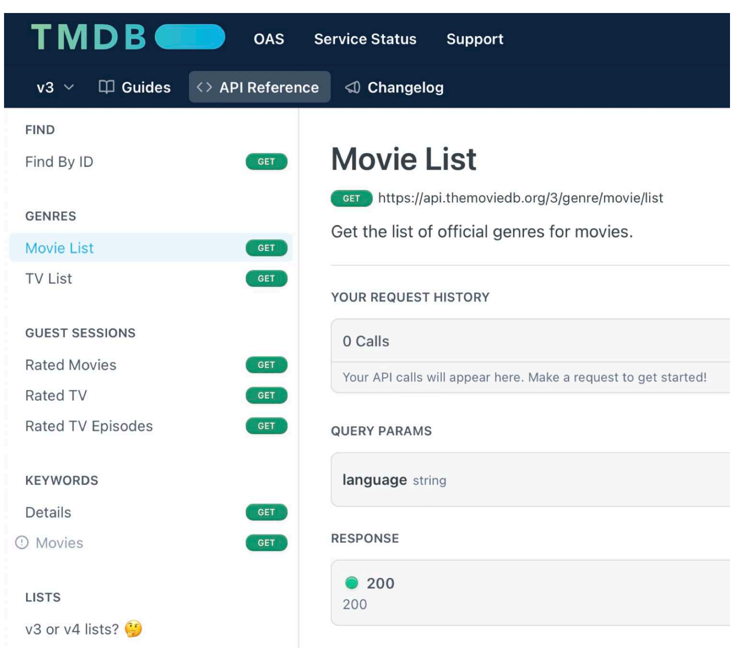


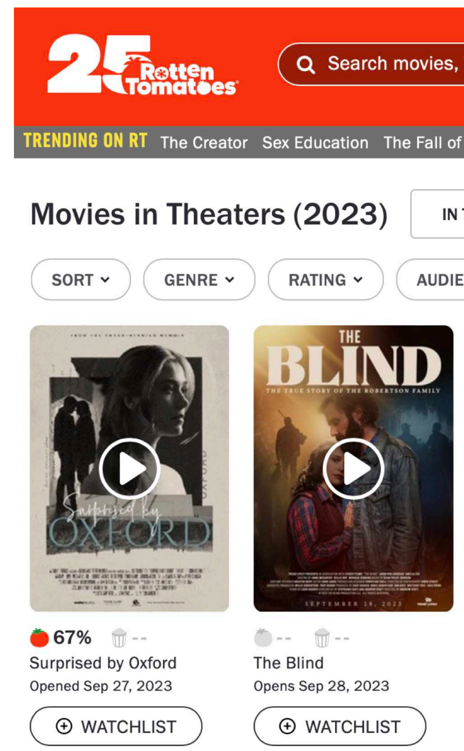


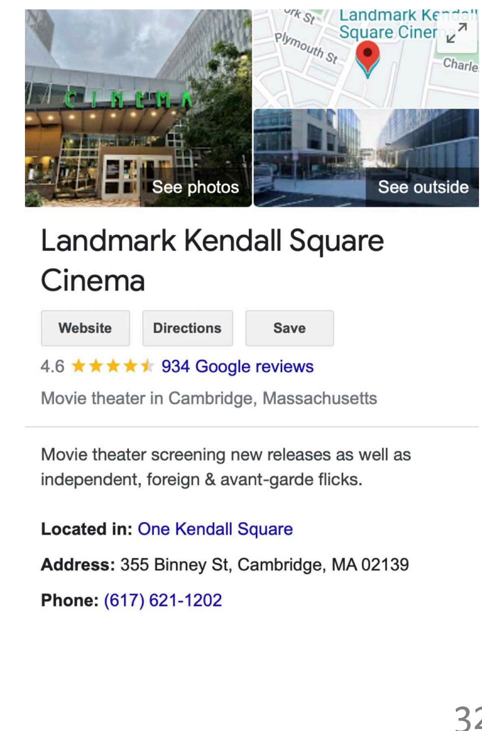


Designing a database: the concept approach

Reviewing comparables can help...







Designing a database: the concept approach

reminder: generic parameters

Starting simple -but- modular

concept Movies
purpose provide info about all movies
state
genres: Movie → **set** Genre
title: Movie → **one** String
year: Movie → **one** Year
remakeOf, sequelTo: Movie → **opt** Movie

concept Showings [Movie, Theater]
purpose provide info on current movie showings
state
movie: Showing → **one** Movie
theater: Showing → **one** Theater
time: Showing → **one** Date
screen: Showing → **one** String

concept Businesses [Location]
purpose provide info on places of business
state
name: Business → **one** String
address: Business → **one** Address
website: Business → **one** URL
location: Business → **one** Location

Designing a database: the concept approach

Identifying stakeholders and thinking about operational principles & actions

concept Posting
principle after making a post,
that post is available to other users

concept Movies
principle after a movie… exists?,
users can find it,
and related movies

At order one new movie per day, handled internally?
What **actions** and **state**?

concept Showings

At order ten thousand per day (US), …?

concept Businesses

Maybe only one update per day, but
across order one thousand theaters, …?

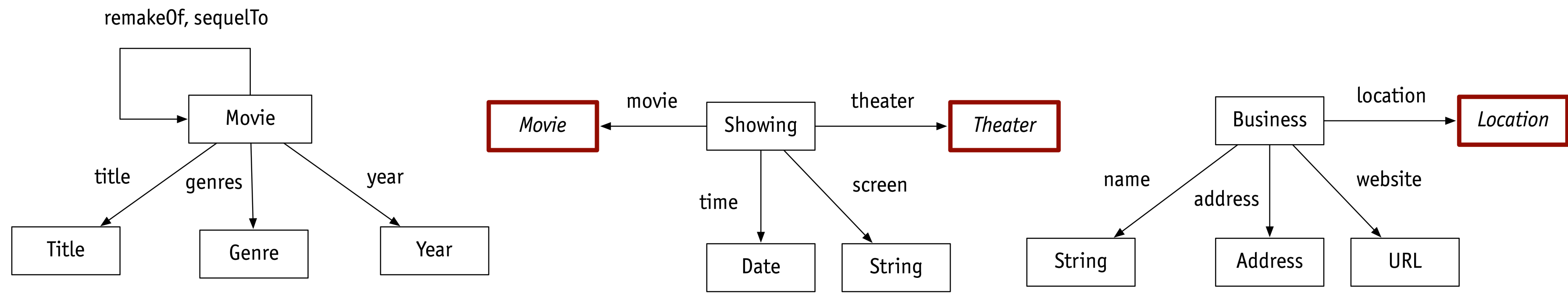
concept Verifying

concept Crowdsourcing

concept Scraping

Designing a database: the concept approach

Step 1: we now have entities and relationships for each concept



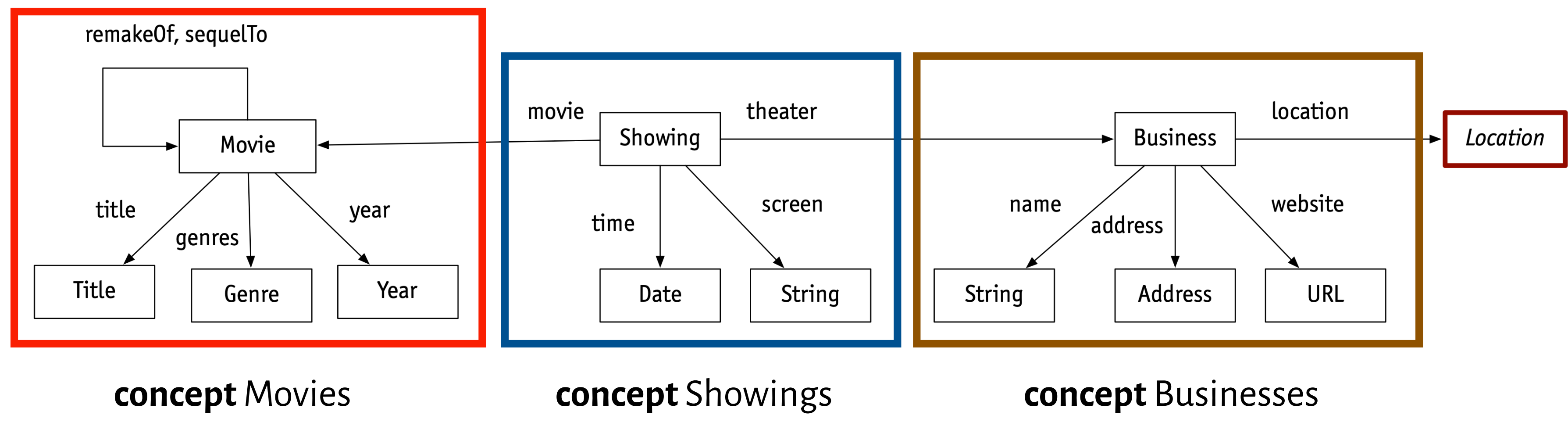
concept Movies
state
genres: Movie → **set** Genre
title: Movie → **one** String
year: Movie → **one** Year
remakeOf, sequelTo: Movie → **opt** Movie

concept Showings [Movie, Theater]
state
movie: Showing → **one** Movie
theater: Showing → **one** Theater
time: Showing → **one** Date
screen: Showing → **one** String

concept Businesses [Location]
state
name: Business → **one** String
address: Business → **one** Address
website: Business → **one** URL
location: Business → **one** Location

Designing a database: the concept approach

Step 2: compose a global data model



app *M104vies*

include Movies, Showings [Movies.Movie, Businesses.Business], Businesses [...]

What about locations?

concept Geo Locations [POI]
purpose find points-of-interest by location

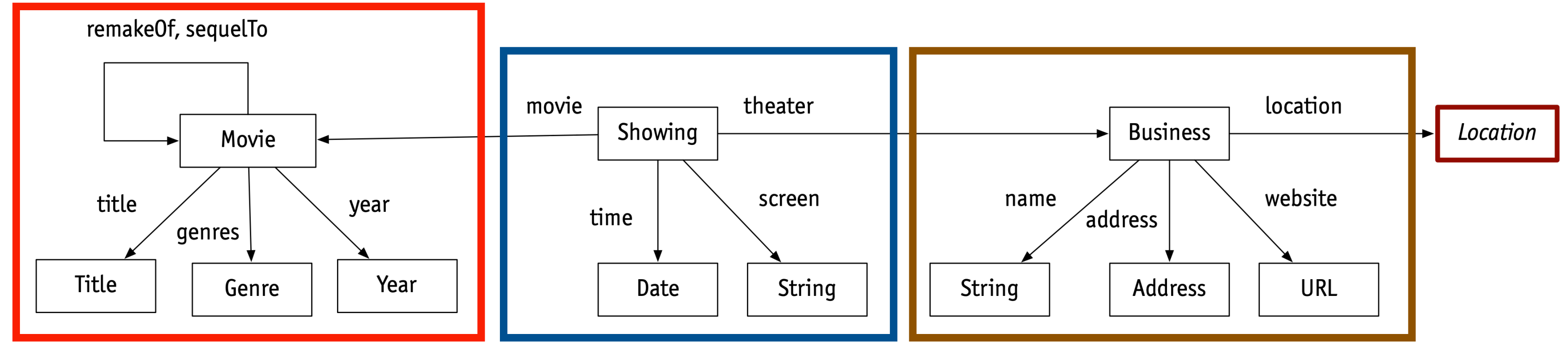
state
  location: POI → **opt** Location

actions
  locate (addr: String, **out** loc: Location)
  add (point: POI, loc: Location)
  findNearby (loc: Location, **out** points: **set** POI)

Actual representation will be a data structure that enables an efficient algorithm for findNearby

Designing a database: the concept approach

Step 3: transform to… *e.g.* a MongoDB schema

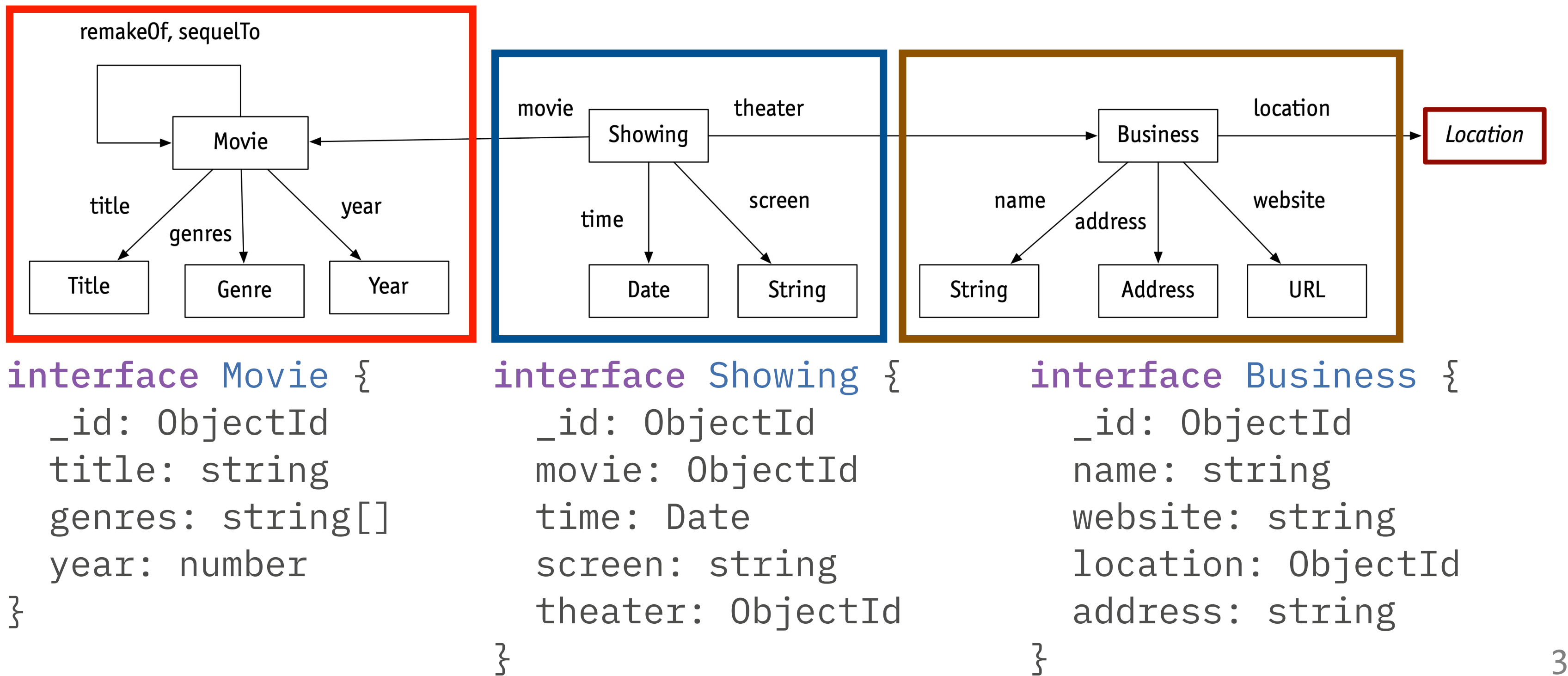


How many collections?

What primitive types?

Designing a database: the concept approach

Step 3: transform to… *e.g.* a MongoDB schema




```
interface Movie {
  _id: ObjectId
  title: string
  genres: string[]
  year: number
}
```

```
interface Showing {
  _id: ObjectId
  movie: ObjectId
  time: Date
  screen: string
  theater: ObjectId
}
```

```
interface Business {
  _id: ObjectId
  name: string
  website: string
  location: ObjectId
  address: string
}
```

Designing a database: the concept approach

Step 3: transform to… *e.g.* a MongoDB schema

```
interface Movie {
  _id: ObjectId
  title: string
  genres: string[]
  year: number
}
```

```
interface Business {
  _id: ObjectId
  name: string
  website: string
  location: ObjectId
  address: string
}
```

benefits and drawbacks?

```
interface MovieShowings {
  _id: ObjectId
  movie: ObjectId
  showings: [
    { theater:ObjectId, screen:string, time:Date }, ...
  ]
}
```

Designing a database: the concept approach

Step 3: transform to… *e.g.* a MongoDB schema

```
interface Movie {
  _id: ObjectId
  title: string
  genres: string[]
  year: number
}
```

```
interface Business {
  _id: ObjectId
  name: string
  website: string
  location: ObjectId
  address: string
}
```

benefits and drawbacks?

```
interface TheaterMovieShowings {
  _id: ObjectId
  theater: ObjectId
  movie: ObjectId
  showings: [
    { screen:string, time:Date }, ...
  ]
}
```

Today

Database models

object-oriented relational document collection

Abstract data models

relations & sets in one global model

Concept-driven data models

data for separable concepts in a composed model

Implementation in MongoDB

Looking ahead

Designing services
Reactive frameworks*!*