

6.1040: Software Design

# Data Design

Arvind Satyanarayan & Max Goldman

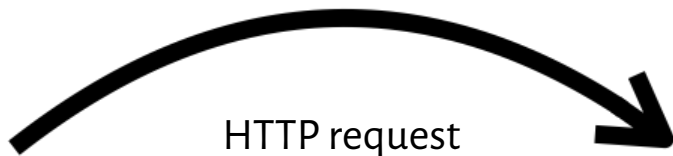
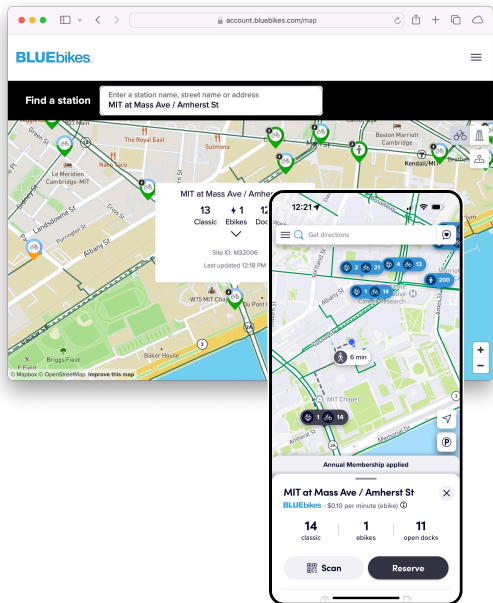
with material by Daniel Jackson

Fall '24

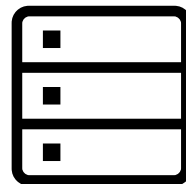
# Client

/

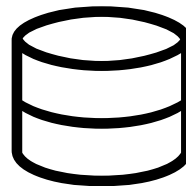
# Server



application server



process request  
& build response



database server



# Today

 Different database models

Classical data modeling

Concept data modeling

 Relational state

Implementation considerations



Crazy Rich Asians  
Drama/Come...



White Boy Rick  
Drama/Myste...



Peppermint  
Drama/Thrille...



Fahrenheit 11/9  
Political cine...



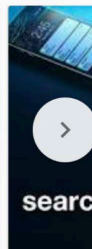
Life Itself  
Drama/Roma...



The Meg  
Thriller/Fanta...



Little Women  
Drama/Family

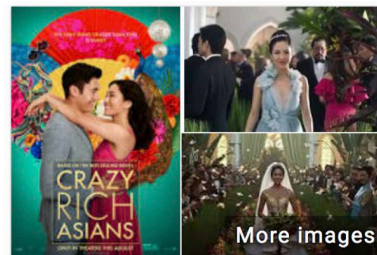


Searching  
Drama/TI

## Showtimes for Crazy Rich Asians

All times are in ET

	Today	Tomorrow	Tue, Oct 2	Wed, Oct 3		
<b>All times</b>	Morning	Afternoon	Evening	Night		
<b>AMC Loews Boston Common 19 - <a href="#">Map</a></b>	Standard	4:40pm	7:30pm	10:20pm		
<b>Regal Fenway Stadium 13 &amp; RPX - <a href="#">Map</a></b>	Standard	4:10pm	7:20pm	10:30pm		
<b>ShowPlace ICON at Seaport with ICON-X - <a href="#">Map</a></b>	Standard	4:45pm	6:10pm	7:45pm	9:10pm	10:30pm
<a href="#">More showtimes</a>						



## Crazy Rich Asians

**[PG-13]** 2018 · Drama/Comedy-drama · 2h 1m

7.5/10  
IMDb

93%  
Rotten  
Tomatoes

74%  
Metacritic

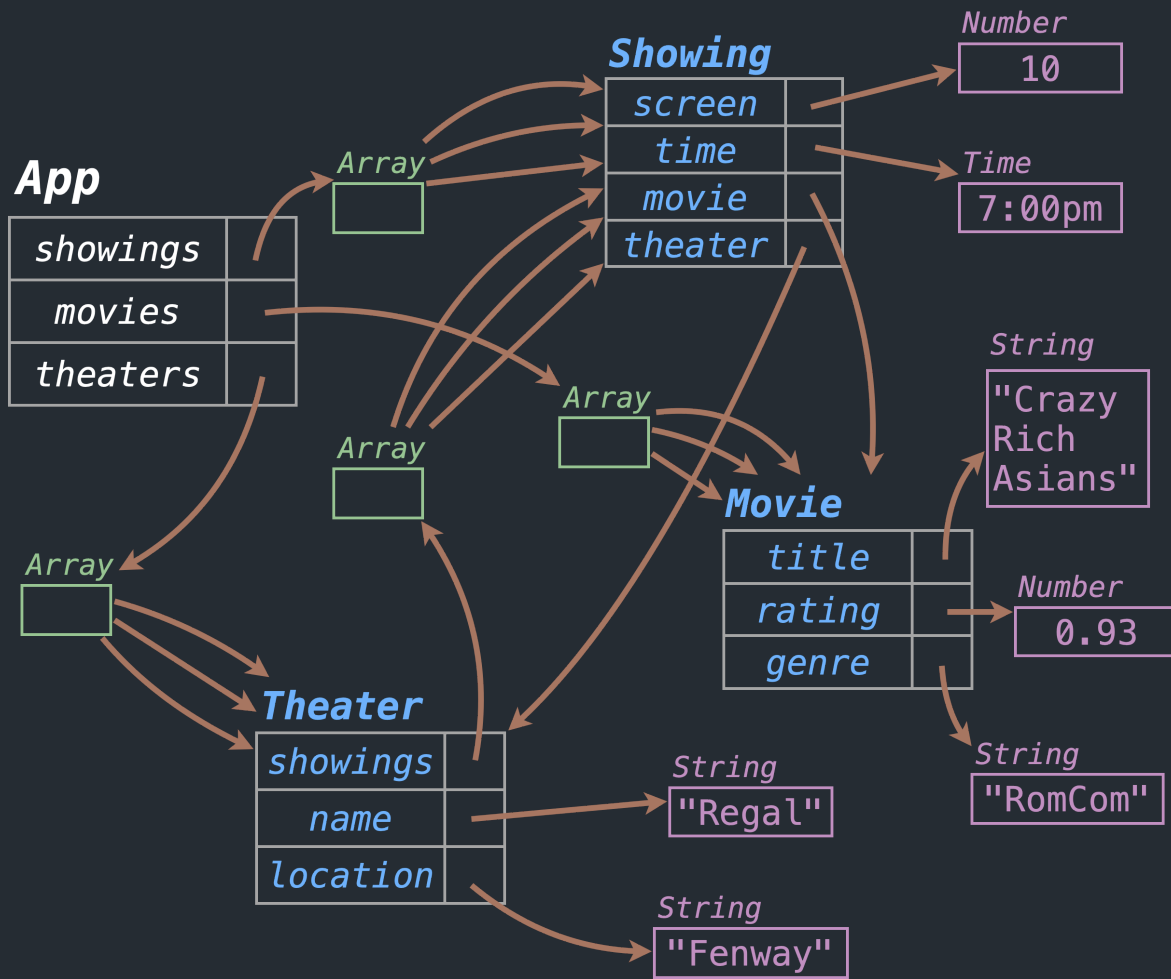
93% liked this movie  
Google users



# 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



## 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"

# Relational model

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

# 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*)

<i>_id</i>	3	
<i>title</i>	"Crazy Rich Asians"	
<i>time</i>	7:00pm	
<i>genre</i>	"RomCom"	
<i>theater</i>	<i>name</i>	"Regal"
	<i>location</i>	"Fenway"

<i>_id</i>	4	
<i>title</i>	"Crazy Rich Asians"	
<i>time</i>	7:30pm	
<i>genre</i>	"RomCom"	
<i>theater</i>	<i>name</i>	"AMC"
	<i>location</i>	"Boston Common"

# 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



<i>_id</i>	3	
<i>title</i>	"Crazy Rich Asians"	
<i>time</i>	7:00pm	
<i>genre</i>	"RomCom"	
<i>theater</i>	<i>name</i>	"Regal"
	<i>location</i>	"Fenway"

<i>_id</i>	4	
<i>title</i>	"Crazy Rich Asians"	
<i>time</i>	7:30pm	
<i>genre</i>	"RomCom"	
<i>theater</i>	<i>name</i>	"AMC"
	<i>location</i>	"Boston Common"

# 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

## Create

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

## Read

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

## Update

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

Updates specify fields to change

## 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

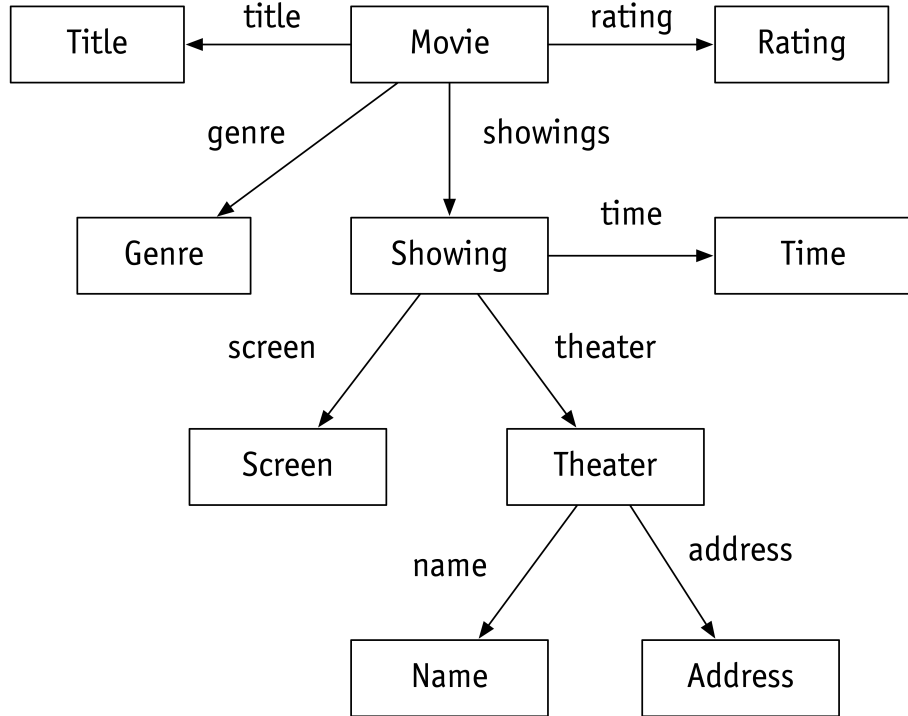
Relational model: tables of tuples

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

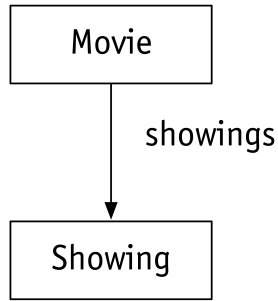
Representation-independent

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

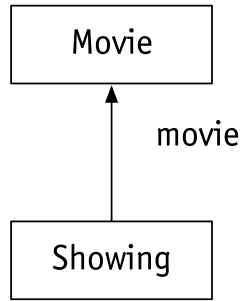
Diagram shows us possible navigations  
... but!

# Designing a database: the classic approach

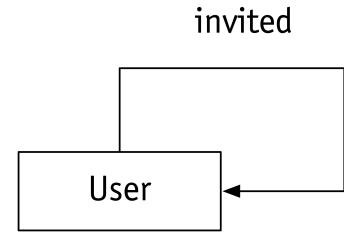
A common confusion: arrow direction



Arrow direction is  
*NOT* navigation and  
*NOT* containment



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



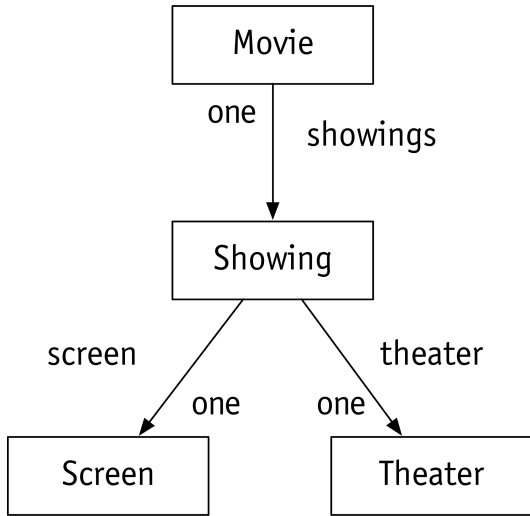
Matters for homogeneous  
relations where we can  
easily interpret wrong:

(alice, bob) in invited

*Did Alice invite Bob,  
or did Bob invite Alice?*

# Designing a database: the classic approach

## Step 2: add multiplicities



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

$\geq 0$  **set**, *the default*

$\geq 1$  **some**, **+**

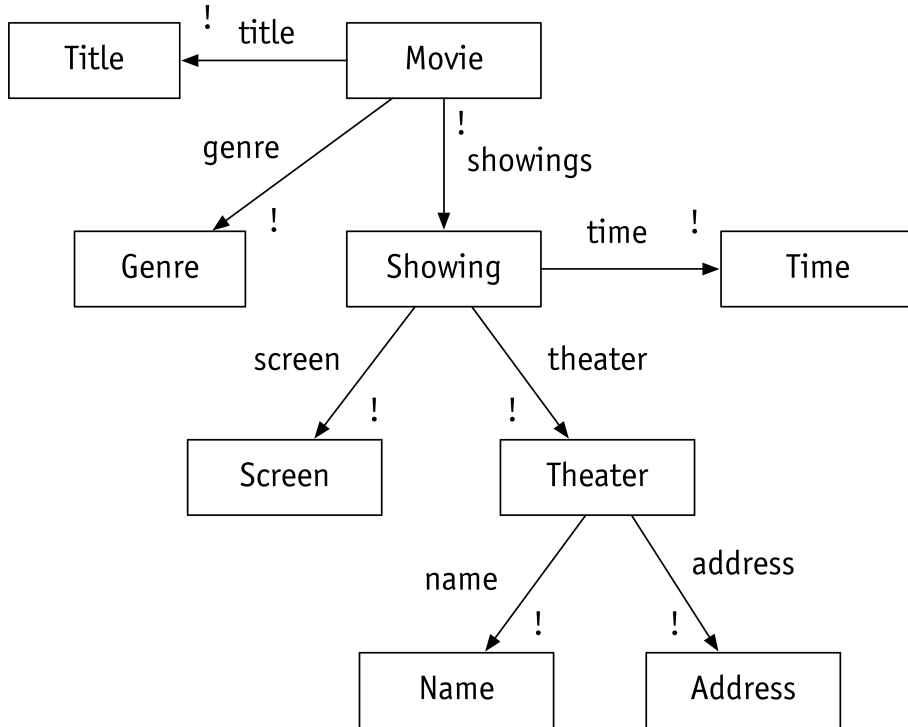
$\leq 1$  **opt, lone**, **?**

$= 1$  **one**, **!**

(Many other notations, see [xkcd.com/927](http://xkcd.com/927))

# Designing a database: the classic approach

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



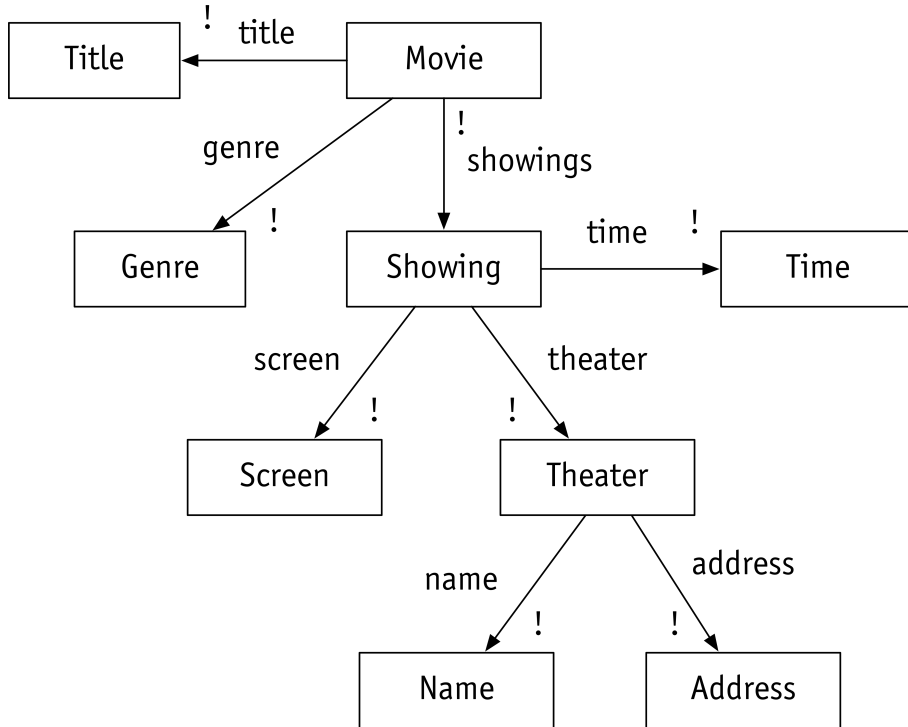
movies?

<i>id</i>	<i>title</i>	<i>showings?</i>
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

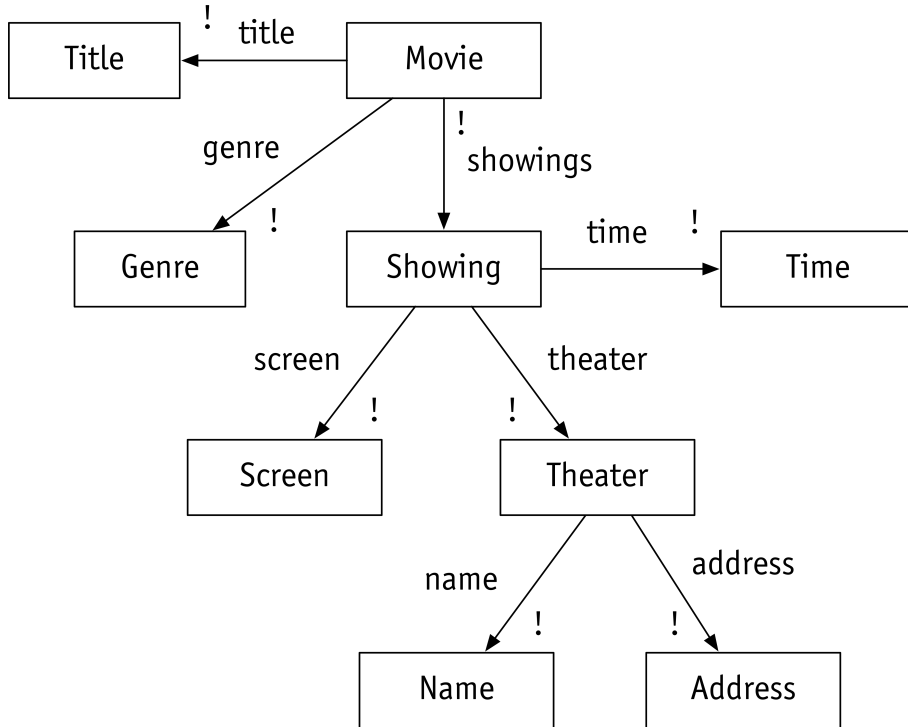


```
CREATE TABLE movies
  (id int, title text, genre text);
CREATE TABLE showings
  (id int, movie int, screen int,
  theater int, time timestamp);
```

**Constraint:** no set-valued columns\*

# Designing a database: the classic approach

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



movies

<i>id</i>	<i>title</i>	<i>genre</i>
1	Crazy Rich Asians	RomCom
2	Barbie	Fantasy

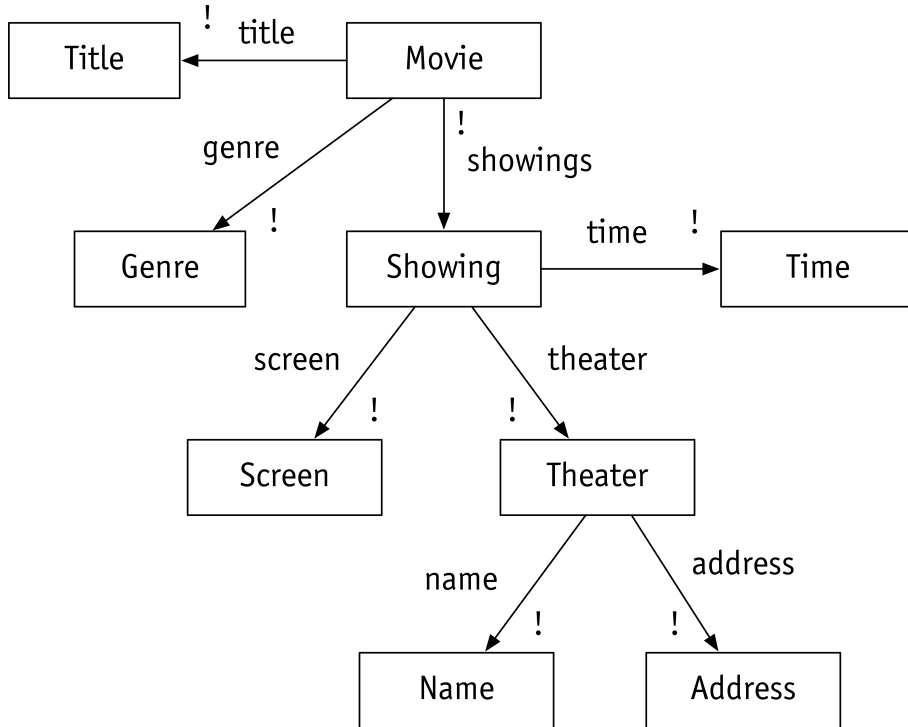
showings

<i>id</i>	<i>movie</i>	<i>screen</i>	<i>theater</i>	<i>time</i>
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



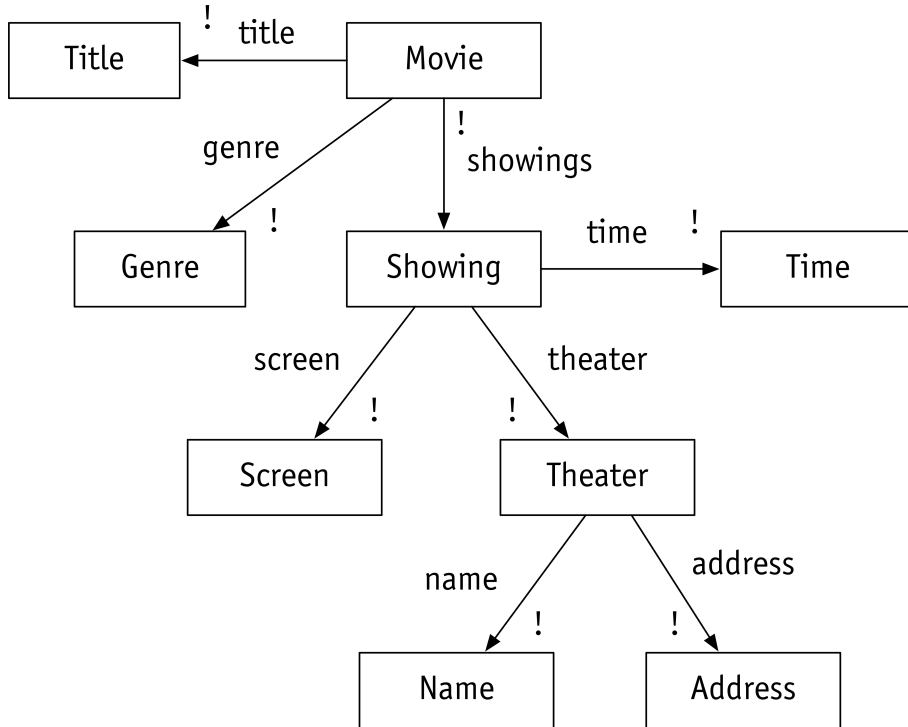
```
class Movie {  
    title: string;  
    genre: Genre;  
    showings: Array<Showing>;  
}
```

```
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>>;  
}
```

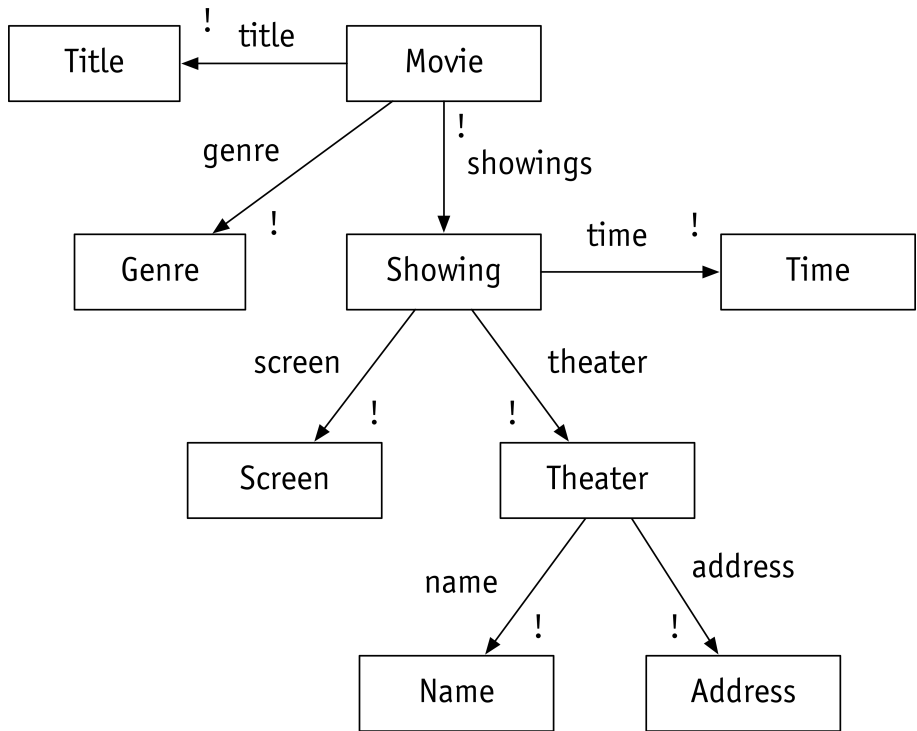
```
class Showing {  
  screen: number;  
  theater: Theater;  
  time: Date;  
}
```

**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!

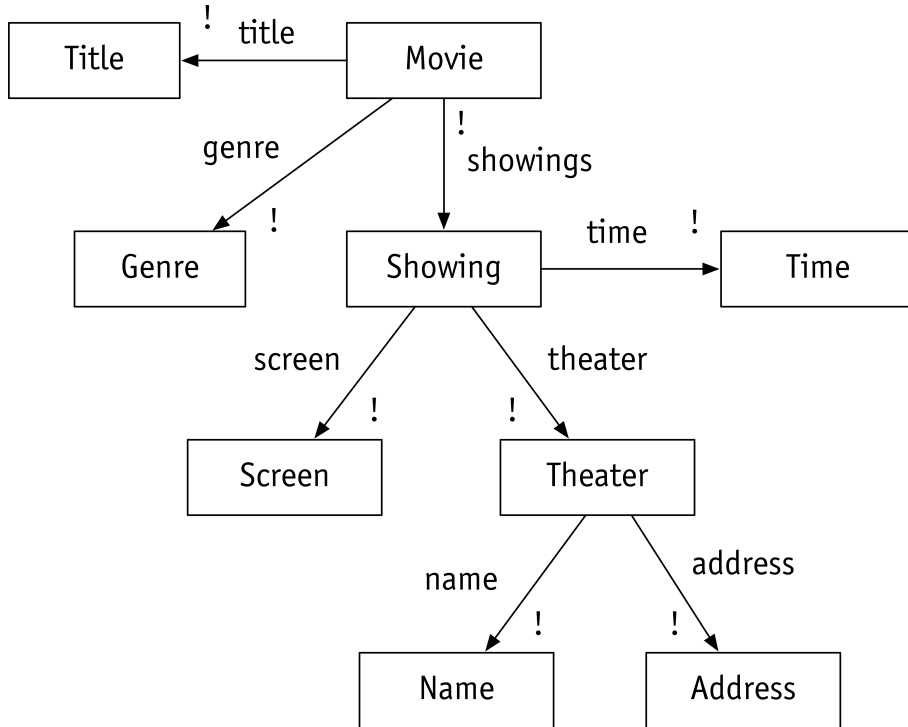
We will use TypeScript interfaces

```
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

<i>id</i>	1				
<i>title</i>	"Crazy Rich Asians"				
<i>time</i>	7:00pm				
<i>genre</i>	"RomCom"				
<i>screen</i>	2				
<i>theater</i>	<table border="1"><tr><td><i>name</i></td><td>"AMC"</td></tr><tr><td><i>address</i></td><td>"401 Park Drive"</td></tr></table>	<i>name</i>	"AMC"	<i>address</i>	"401 Park Drive"
<i>name</i>	"AMC"				
<i>address</i>	"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?

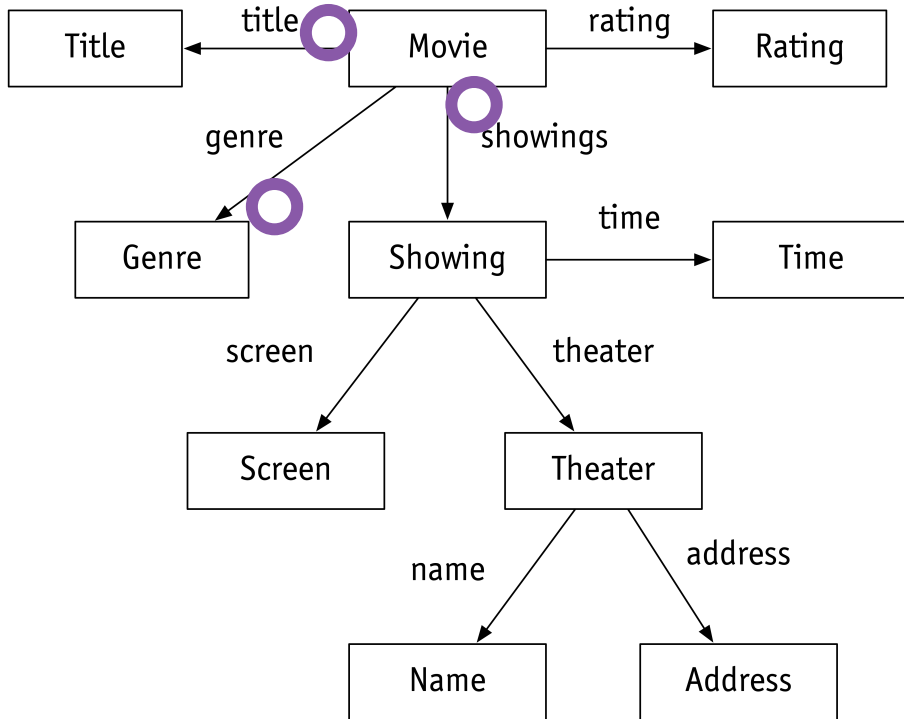
*e.g.* cost of relational joins mitigated by indexes

What is the cost of updates?

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

# Multiplicities matter

What are *these* multiplicities?



$\geq 0$  **set**, *the default*

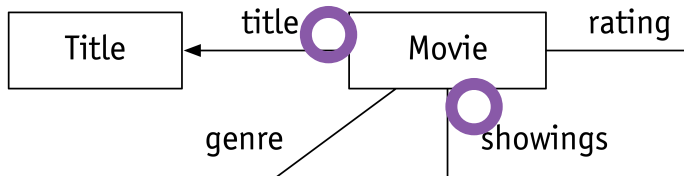
$\geq 1$  **some**, **+**

$\leq 1$  **opt**, **lone**, **?**

$= 1$  **one**, **!**

# Multiplicities matter

What are **these** multiplicities?



Genre

*Edge of Tomorrow*<sup>[a]</sup> is a 2014 American [science fiction action film](#) and written by [Christopher McQuarrie](#) and the writing team of [Jez](#) and [Butterworth](#), loosely based on the [Japanese light novel](#) *All You Need Is Kill* by [Sakurazaka](#). Starring [Tom Cruise](#) and [Emily Blunt](#), the film takes place in a [future](#) where [Europe](#) is occupied by an alien race. Major William Cage (Cruise) with no combat experience, is forced by his superiors to join a [fight](#) against the aliens, only to find himself experiencing a [time loop](#) as he tries to find a way to stop the invaders. [Bill Paxton](#) and [Brendan Gleeson](#) also appear in supporting

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. <sup>^</sup> Later marketed as *Live Die Repeat: Edge of Tomorrow*<sup>[6]</sup>

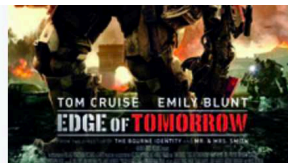
Barbie / Genres

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

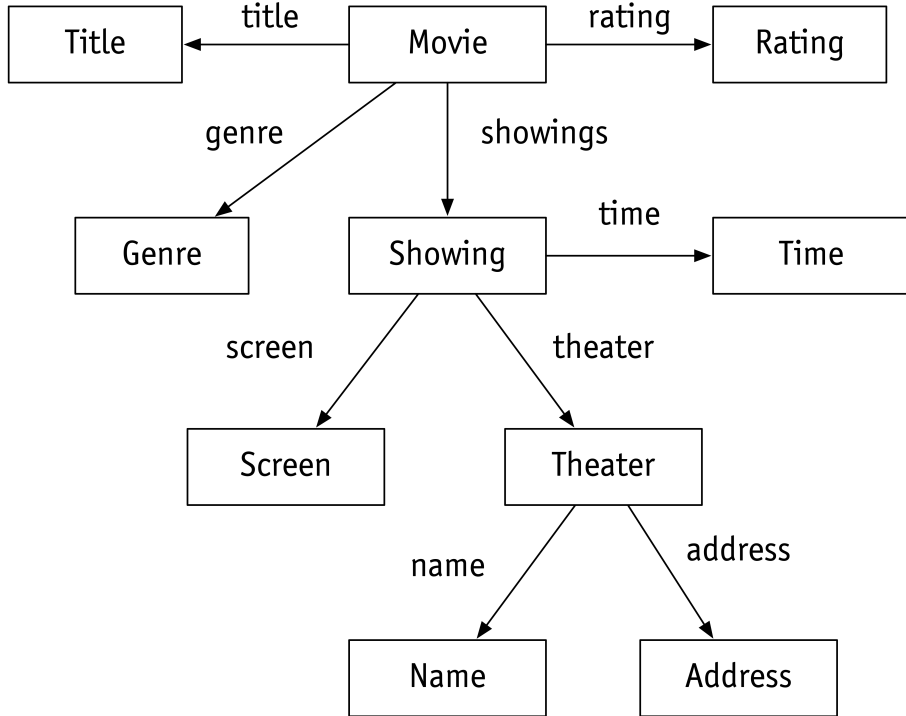
- 'Night Moves' (1975) and (2013) ...
- 'Missing' (1982) and (2023) ...
- 'Twilight' (1998) and (2008) ...
- 'Possession' (2014) and (2023) ...
- 'Rush' (1999) and (2013) ...

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 Villeneuve 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!



# 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?








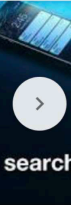
Where is the modularity?

How can we reuse modeling within or across systems?

# Designing a database: the concept approach

## Step 1: identify the concepts

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/TI

### Showtimes for Crazy Rich Asians

All times are in ET


Today Tomorrow Tue, Oct 2 Wed, Oct 3



All times Morning Afternoon Evening Night


AMC Loews Boston Common 19 - [Map](#)  
Standard 4:40pm 7:30pm 10:20pm

Regal Fenway Stadium 13 & RPX - [Map](#)  
Standard 4:10pm 7:20pm 10:30pm

ShowPlace ICON at Seaport with ICON-X - [Map](#)  
Standard 4:45pm 6:10pm 7:45pm 9:10pm 10:30pm



 More showtimes



**Crazy Rich Asians** 

PG-13 2018 · Drama/Comedy-drama · 2h 1m

7.5/10 IMDb	93% Rotten Tomatoes	74% Metacritic
----------------	------------------------	-------------------

93% liked this movie   31

Google users

# Designing a database: the concept approach

Reviewing comparables can help...

The screenshot shows the TMDB API Reference interface. The top navigation bar includes 'TMDB', 'OAS', 'Service Status', and 'Support'. Below this, there are links for 'v3', 'Guides', 'API Reference', and 'Changelog'. The left sidebar lists various API endpoints under categories like 'FIND', 'GENRES', 'TV List', 'GUEST SESSIONS', 'KEYWORDS', and 'LISTS'. The main content area is titled 'Movie List' and shows the endpoint URL 'https://api.themoviedb.org/3/genre/movie/list' with a 'GET' button. Below the URL, it says 'Get the list of official genres for movies.' There is also a 'YOUR REQUEST HISTORY' section showing 0 calls and a 'QUERY PARAMS' section with 'language string'. At the bottom, the 'RESPONSE' section shows a count of 200 items.

The screenshot shows the Rotten Tomatoes 'Movies in Theaters (2023)' page. The top banner features the '25 Rotten Tomatoes' logo and a search bar. Below the banner, there are filters for 'TRENDING ON RT', 'The Creator', 'Sex Education', and 'The Fall of'. The main heading is 'Movies in Theaters (2023)' with a 'IN 1' indicator. Below the heading, there are filter buttons for 'SORT', 'GENRE', 'RATING', and 'AUDIENCE'. Two movie posters are displayed: 'Surprised by Oxford' and 'The Blind'. Below each poster, there is a play button icon, a Rotten Tomatoes score (67% for 'Surprised by Oxford'), and the release date ('Opened Sep 27, 2023'). At the bottom of each poster, there is a 'WATCHLIST' button.

The screenshot shows the website for Landmark Kendall Square Cinema. The top part of the page features a large image of the cinema building with the word 'CINEMA' in green letters. To the right, there is a map showing the location at the intersection of Kendall Square and Plymouth St. Below the main image, there are two buttons: 'See photos' and 'See outside'. The main heading is 'Landmark Kendall Square Cinema'. Below the heading, there are three buttons: 'Website', 'Directions', and 'Save'. The rating is 4.6 stars with 934 Google reviews. Below the rating, it says 'Movie theater in Cambridge, Massachusetts'.

Movie theater screening new releases as well as independent, foreign & avant-garde flicks.

**Located in:** One Kendall Square

**Address:** 355 Binney St, Cambridge, MA 02139

**Phone:** (617) 621-1202



# 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, ...?

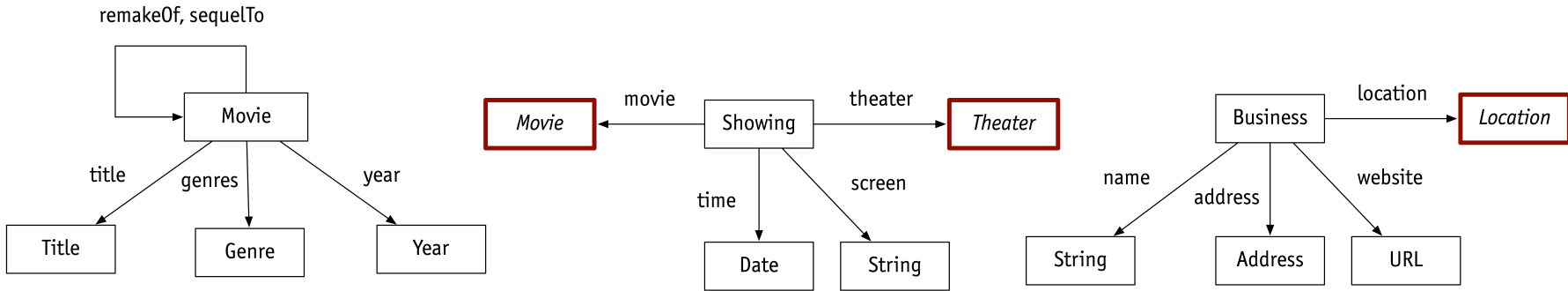
**concept** Verifying

**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

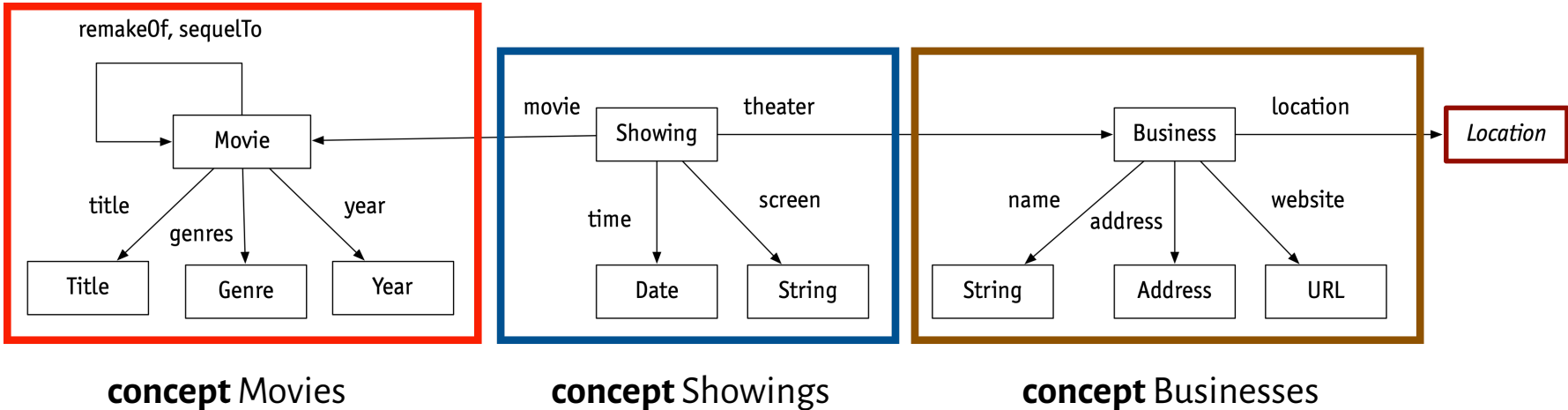
## concept Businesses [Location]

### 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)

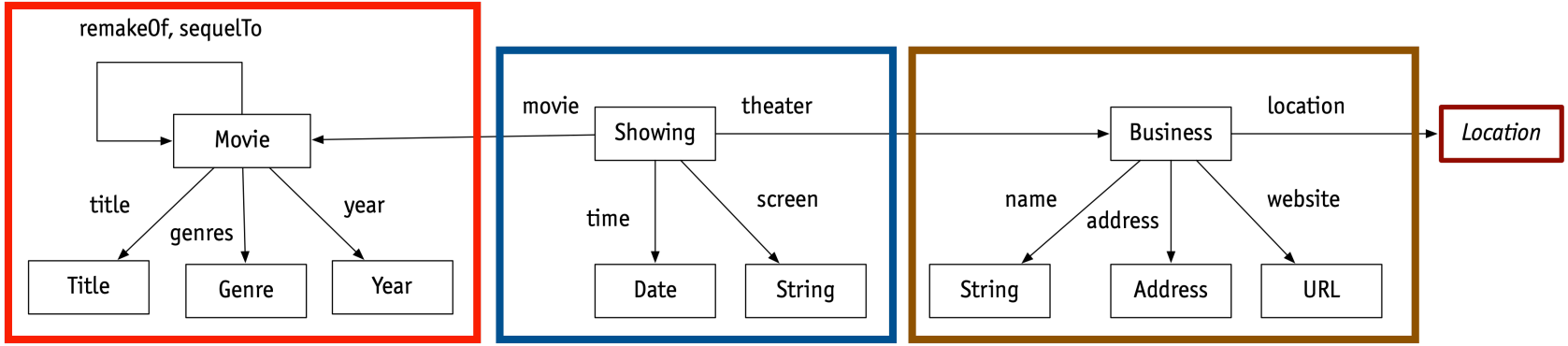
add (point: POI, loc: Location)

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

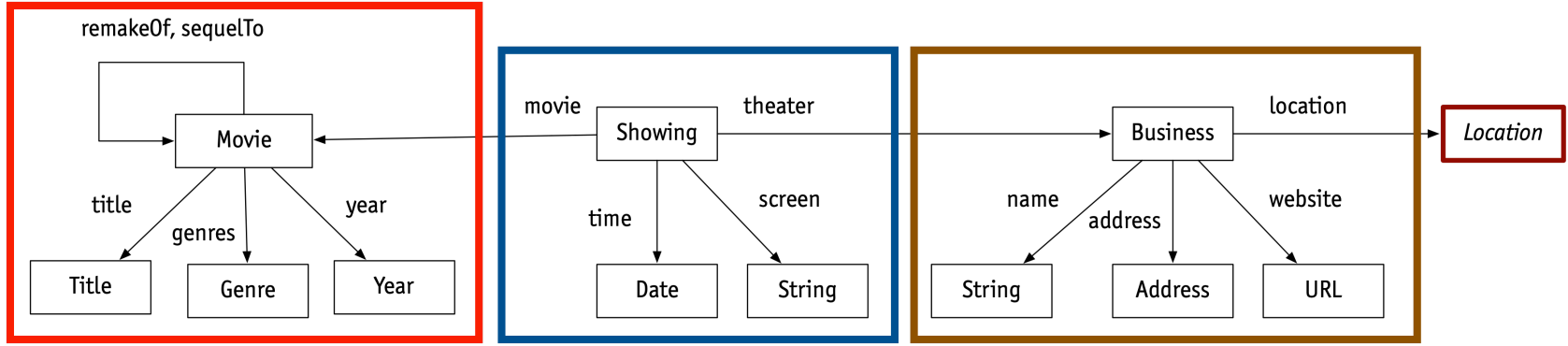


How many collections?

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  
}
```

benefits and drawbacks?

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

```
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  
}
```

benefits and drawbacks?

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

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

# 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!