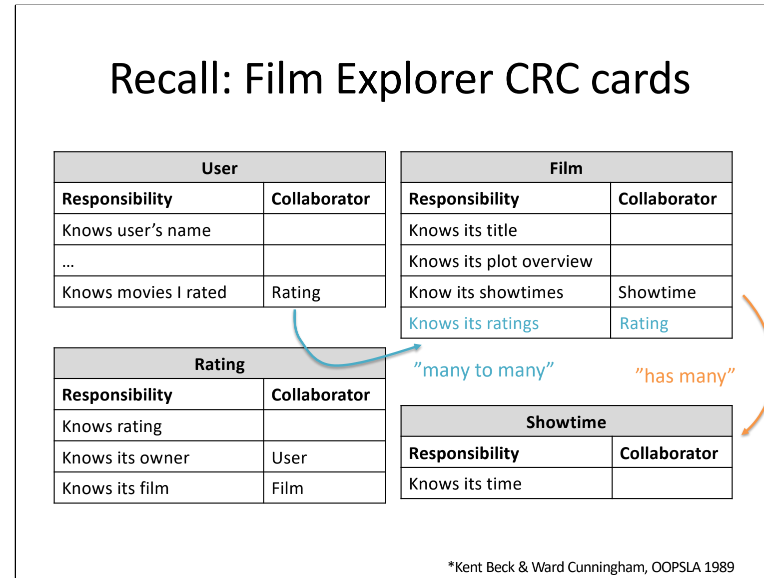
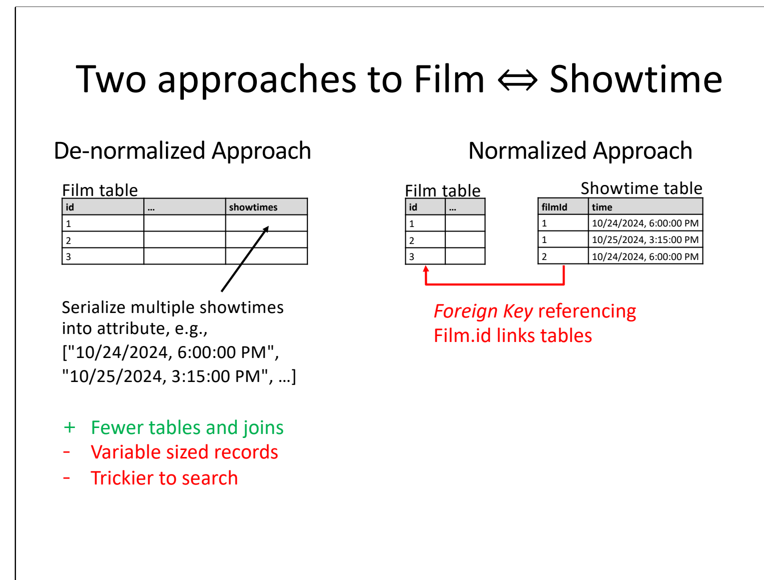


These associations have specific schema associated with them. That is the association will determine what columns we need in our database. Specifically …

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.



The first approach is what we would implement in a memory backed server (and most NoSQL DBs). All the data for film, including its one or more showtimes are packed together in a single, albeit variable sized, object. No additional data is required. In contrast, a normalized approach breaks the film into two fixed size parts, the film itself and separate showtime entries, that are linked together via foreign keys, that is we need to include additional columns/attributes - the filmId column - to create the connection between the two tables. The second, normalized, approach is typically used with RDBMS (these are the relations in the name).

We could describe normalization as eliminating repeated information in a table by decomposing repeating entries into separate linked tables. The data is reconstructed via join operations (to come…)

More specifically, “first normal form” enforces these criteria: 1) Eliminate repeating groups in individual tables, 2) Create a separate table for each set of related data, 3) Identify each set of related data with a primary key

Foreign keys enforce the connection between two tables. A foreign key is a constraint not a type. To insert an entry into Showtime, there must be a corresponding entry in the Film (it will fail if there is no Film). Ensuring that all foreign key references are valid is termed maintaining “referential integrity” and is one of the benefits of RDBMs.

Associations: “One-to-many” example

```
GET /api/films/1
```

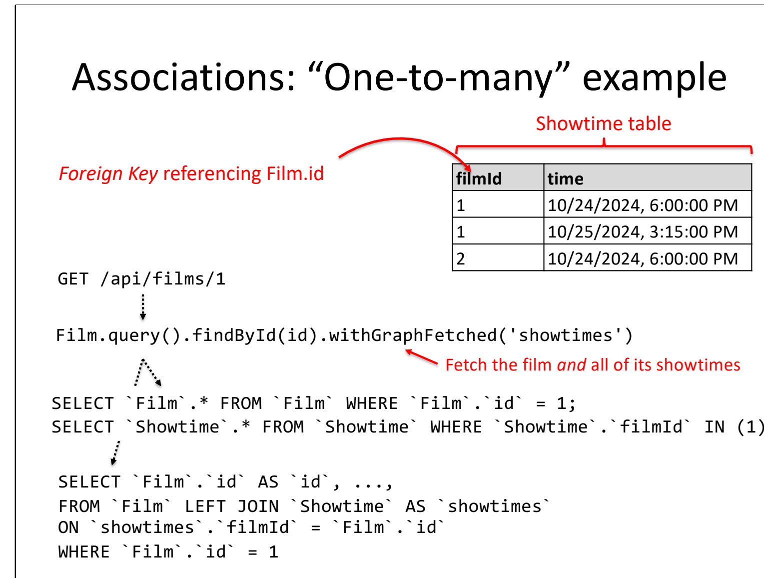
Desired response

```
{
   "id":1,
   "title":"Smile 2",
   ...
   "showtimes":[
      {
         "filmId":1,
         "time":"10/18/2024, 3:15:00 PM"
      }, ...]
}
```

From Film table

From Showtime table

When we implement this route to on the film explorer server we want to obtain all the data for this film, including its showtimes. But now that data is separate. How do we combine the data from the two tables?



Here we see two approaches to construct the entire object returned by this route:

1. The multiple queries approach first gets gets the film, then the associated showtimes.
2. The join approach conceptually builds a table with a row for each combination of movie and showtime (with both sets of columns and duplicated movie entries) and then filters that table according to the join conditions and where clause, etc.

The former requires more queries (more latency), but it is simpler to parse into tree of Objects. The latter is one (complex) query but parsing results into objects will be a little more involved.

In practice we will not implement these queries directly. Instead, we will use the (Objection.js) ORM to create the queries for us. Here we are telling Objection to eagerly, as opposed to lazily, fetch and populate the showtimes. The ORM uses the associations defined in the model class to generate the appropriate query and construct the final object.

Implementing "one-to-many" with Objection.js

```
Film
 .query()
 .findById(id)
 .withGraphFetched(
  'showtimes'
 )

SELECT `Film`.* FROM `Film`
  WHERE `Film`.`id` = 1;
SELECT `Showtime`.* FROM `Showtime`
  WHERE `Showtime`.`filmId` IN (1);
```

```
class Film extends Model {
  static get tableName() { return 'Film'; }
  …
  static get relationMappings() {
    return {
      showtimes: {
        relation: Model.HasManyRelation,
         modelClass: Showtime,
        join: {
          from: 'Film.id',
          to: 'Showtime.filmId',
        },
      },
    };
  }
}
```

Relevant columns in DB schema

How did Objection.js know to generate that queries from that terse set of calls? We specified the relation in the Model. Here is the relevant portion of Film Model. We specify a one-to-many relation named "genres" (the key in the object return by relationMappings) and the relevant columns in the DB that implement that relation. It is that name (and relation) that we are referencing in withGraphFetched. Using that name, and the corresponding association information in the model, Objection can generate the relevant query.

Joins as filtered cartesian product

Film × Showtime cartesian product

Film.id	...	Showtime.filmId	Showtime.time
1		1	10/24/2024, 6:00:00 PM
2		1	10/24/2024, 6:00:00 PM
3		1	10/24/2024, 6:00:00 PM
1		1	10/25/2024, 3:15:00 PM
2		1	10/25/2024, 3:15:00 PM
3		1	10/25/2024, 3:15:00 PM
1		2	10/24/2024, 6:00:00 PM
2		2	10/24/2024, 6:00:00 PM
3		2	10/24/2024, 6:00:00 PM

Film.id == Showtime.filmId

Joins are such a key feature of an RDBMS I want to briefly expand on what is going on behind the scenes. Our mental model for joins is a filtered cartesian product. That is the database system is creating all combinations of entries from the Film table and the Showtime table and then only keeping those where the join criteria, in this case that Film.id == Showtime.filmId, is true (the actual implementation is more efficient than that though!).

Interlude: Refining server responses

```
GET /api/films/1
```

Desired response

```
{
   "id":1,
   "title":"Smile 2",
   ...
   "showtimes":[ {"filmId":1,
        "time":"10/18/2024, 3:15:00 PM"}, ...]
}
```

Duplicated, many times if there are multiple showtimes!

This is better

```
{
  "id":1,
  "title":"Smile 2",
  …
  "showtimes":[
    "10/18/2024, 3:15:00 PM", …]
}
```

And can be achieved through JS!

```
const { showtimes, ...film } = ...;
res.status(200).json({
  ...film,
  showtimes: showtimes.map(s => s.time)
});
```

Our previous response was the direct output produced by Objection.js as it joined the Film and its showtimes. In many cases that is exactly what we want. But we notice it contains lots of extraneous data (i.e., repeats of the filmId). We could imagine want to strip that out, or otherwise modify the response. One place to do is in the API handler. Keep in mind that is just JavaScript code and so we can execute other operations, like transforming the showtimes. If we wanted to do this every time with your model, we are best off integrating that transformation into the model itself..

Ratings: A "many-to-many" association

filmId	userId	rating
int	int	int
12	4	2
53	4	3

Rating
"Join Table"

Get a movie with its ratings?

```
GET /api/films/12
Film.query().findById(id).withGraphFetched('ratings')
```

Create a new rating for a movie?

```
POST /api/ratings
Rating.query().insert({…})
```


Or from a movie

```
POST /api/films/12/ratings
movie.$relatedQuery('ratings').insert({…})
```


The last assumes we have obtained the movie and the user already in the handler, e.g. via fetchById.

Where do the foreign keys go?

- One-to-One or “HasOne”/“BelongsToOne”
  Foreign key typically in the “BelongsToOne” side (although could be reversed)
- One-to-Many or “HasMany”/“BelongsToOne”
  Foreign key in “BelongsToOne” side (the “many” side of relation)
- Many-to-Many
  Foreign keys in join model, e.g., Rating in “User and Film through Rating”

These are established designs for these relations, that is once you define the relation we know where the keys need to go. We don’t need to figure that out every time.
[end]
Why does the foreign key need to go in the “many” side of the relation? Recall we want the records to be fixed size. If went it in the ”one” side, we would potentially have multiple entries, i.e., a variable length array of keys.

Which of the following is the best migration (schema) for the Showtime table in the Film Explorer? Note that `onDelete('CASCADE')` specifies that rows are deleted from the table if that corresponding row is deleted from the parent table.

A.
```
table.increments('id');
table.integer('filmId');
table.string('time');
```

B.
```
table.integer('filmId');
table.string('time');
table.primary(
  ['filmId', 'time']);
```

C.
```
table.increments('id');
table.integer('filmId')
  .references('id')
  .inTable('Film');
  .onDelete('CASCADE');
table.string('time');
```

D.
```
table.integer('filmId')
  .references('id')
  .inTable('Film')
  .onDelete('CASCADE');
table.string('time');
table.primary(
  ['filmId', 'time']);
```

Answer: D

Answers A & B are missing foreign key constraints and thus will not enforce that a showtime entry must be associated with a valid film. When a Film is a deleted, we want to delete all of its showtime entries, onDelete('CASCADE') does that. Answer C/3 could work but would require an additional attribute, i.e., require more space than D.

Film model (M in MVC)

Film "resource" starts as a simple object (POJO), later transitions to ORM model

- Express associations between models
- Validate user rating is 0-5
- Provide "virtual" attributes that transform data

```
class Film extends Model {
  static get tableName() }
    return 'Film';
  }
  ...
  static get relationMappings() {
    ...
  }
}
```

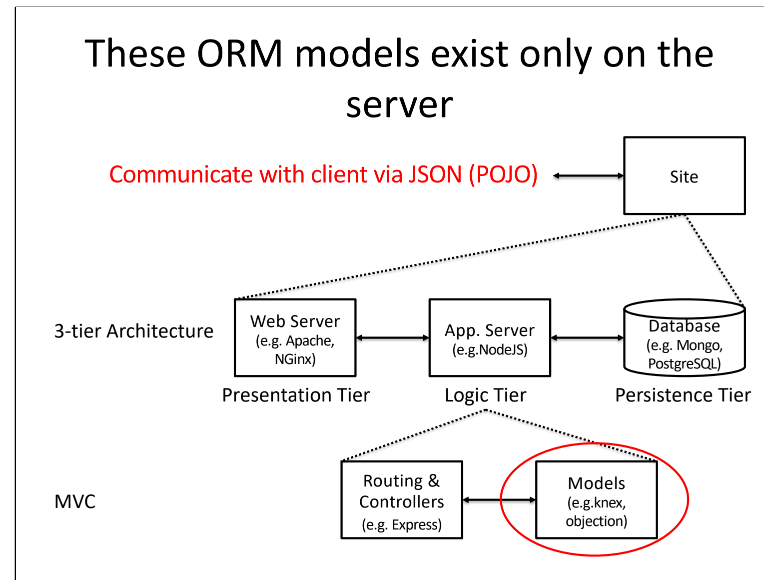
Film table in database

ORM is a design pattern for mapping database schema to object

In the Film Explorer application, the model is a Film. In our practicals, there is no explicit model class, just a plain old JavaScript object (POJO) representing the records in the table. Depending on the application we might not need much more. But as we saw already, Film Explorer could and does benefits from established design patterns and built-in functionality offered by an ORM library (Object Relational Mapping). ORMs are a design pattern for mapping database schema to an object whose methods/properties correspond to attributes in DB, DB queries, etc.

We already saw use of the ORM model to express and implement associations between models (the eager withGraphFetched), some other features are:
Validations: Exactly what they sound like, example of Aspect-oriented programming (i.e., these validations are relevant everywhere the model is created/used. Instead re-implementing that code, we do it once).
Virtual attributes: Convenience "attributes" derived from actual attributes/columns in DB.

Just how different of a database can a given ORM support? Some… Across different RDMSs, e.g., sqlite, MySQL and PostgreSQL. Yes. Relational vs. Non-relational? No.


These ORM models exist only on the server
Communicate with client via JSON (POJO)
Site
3-tier Architecture
Web Server
(e.g. Apache, NGinx)
App. Server
(e.g.NodeJS)
Database
(e.g. Mongo, PostgreSQL)
Presentation Tier
Logic Tier
Persistence Tier
MVC
Routing & Controllers
(e.g. Express)
Models
(e.g.knex, objection)

Validation (recall aspects & AOP)

Mechanisms for validating model data?
- Schema itself (unique, not null, etc.)
- Requirements specified in ORM model

```
Film.query().patchAndFetchById(…, { rating: 10 })
        ↓
Film.fromJson ◄·············· properties: {          Model Schema
        ↓                       ...
throw ValidationError           rating: {
        ↓                         type: ['integer', 'null'],
                                  minimum: 0,
400 Bad Request                   maximum: 5
                                },
                              }
```

The schema alone, i.e., type, is insufficient to enforce the range. So here we leverage the ORMs additional validation tools. This is an example of where AOP can be useful, as these validations should be and are enforced everywhere a model is created/modified, i.e., we want to implement that cross-cutting concern once, not repeatedly…

Note that any constraint could be implemented in this way (not just range).

Remember, AOP is a Design pattern for implementing "cross-cutting" concerns

“in-class” data model

Room	
Responsibility	**Collaborator**
Knows name	
Knows polls	Poll
Knows members	User

Poll	
Responsibility	**Collaborator**
Knows start & end	
Knows results	
Knows room	Room

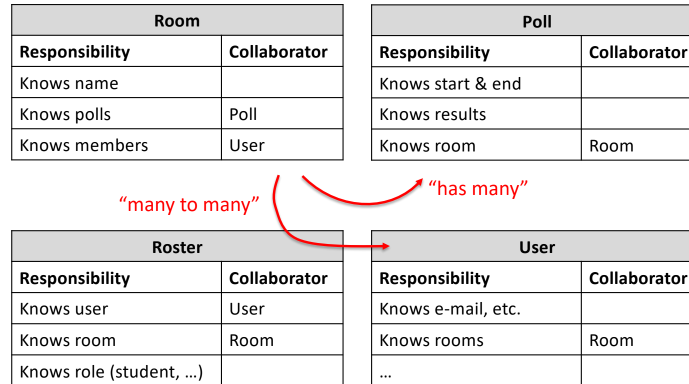
- What relationships are there between room and poll?
- What relationships are there between room and user?

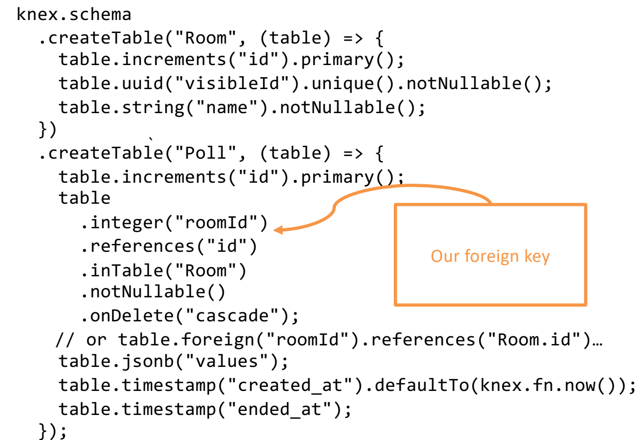
User	
Responsibility	**Collaborator**
Knows e-mail, etc.	
Knows rooms	Room
…	

When you were looking through the in-class backlog, you likely saw these CRC cards. Let’s turn this high-level representation into a design for a relational database.

- We will Let’s start by translating the knows/collaborators into the formal associations we saw last time.
    - What are the relationships between Room and Poll? One-to-many, i.e., a room has many polls, but a poll belongs to a single room.
    - What about relationship between Room and User? This is a many-to-relationship, as each room has many users and a user could be a member (student or instructor) of many rooms (i.e., of many classes). That implies the need for a join table that connects these two. Further in this case we will likely want to encode additional information about this relationship, e.g., is the user a student or an instructor. Let’s call that join Model, Roster, Thus we would express this as a many-to-many through Roster.

Roster	
Knows user	User
Knows room	Room
Knows role { student, …}	


"in-class" data model
Room
Responsibility
Collaborator
Knows name
Knows polls
Poll
Knows members
User
Poll
Responsibility
Collaborator
Knows start & end
Knows results
Knows room
Room
"many to many"
"has many"
Roster
Responsibility
Collaborator
Knows user
User
Knows room
Room
Knows role (student, …)
User
Responsibility
Collaborator
Knows e-mail, etc.
Knows rooms
Room
…



What are some things that you now notice (especially as it relates to the associations we just defined)?

- Recall that in a one-to-many relationship the foreign key goes in the "many" side (so our records are of a fixed size). We see that here, the roomId is an integer that references the id in Room (recall foreign keys are a constraint, not a type). Further, we specify that polls should be deleted if the room is…
- The other attributes you see, e.g., values, created_at, and ended_at, correspond to the knows in the CRC card (the values and the start/end times).
- They have specific types (doing so helps with performance, validation, etc. "out-of-the-box"). We want to look first to specialized types that might be relevant to our attributes before defaulting to something like string, text etc.

https://github.com/csci312-common-v2/class-interactor/blob/main/src/knex/migrations/20230116140145_rooms.ts

How would we design the User/Roster tables?

We would need to create the User table. Does the model schema have any references to Room, etc. No. That is all contained within the Roster join table…
What does the Roster table need? roomId and userId columns that foreign keys (linked the to the ids in those respective tables), and an additional column that stores the role for this individual (here is an enumerated value, i.e., it can only take on certain values, e.g., someone who can administer the room and someone or is a participant (e.g., maybe they can only view, not control/administer a room).

If we assume a user and room can only have one relationship, then the combination of userId and roomId will be unique and could be used as a composite primary key

```
return knex.schema.createTable("User", (table) => {
table.increments("id").primary();
table.string("googleId");
table.string("name");
table.text("email");
})
.createTable("Roster", (table) => {
table.foreign("userId").references("User.id").notNullable
().onDelete(”CASCADE");
table.foreign("roomId").references("Room.id").notNullable
().onDelete(”CASCADE");
```

```
table.enu("role", ["administrator", "student"], {
useNative: true, enumName: "roster_role_type"
}).notNullable();
table.primary(['userId', 'roomId']);
});
```

How would we use this association?

```
// relation mapping in objection.js Room model
users: {
  relation: Model.ManyToManyRelation,
  modelClass: User,
  join: {
    from: "Room.id",
    through: {
      from: "Roster.roomId",
      to: "Roster.userId",
      extra: ["role"]
    },
    to: "User.id",
  }
}
```

Ensures that "role" is included from the join table

Model relation enables concise query…

```
Room
  .query()
   .where(…)
   .withGraphFetched("users");
```

Which produces…

```
[ Room {
    id: 1,
    visibleId: …,
    name: 'TestClass',
    users: [ User {
       id: 1,
       name: …,
       role: "administrator",
    }]
}]
```

```
 // When fetching users make sure to include "role" from the join table
```

Standup Meeting Prompts

Each team member should discuss:

- What they did since the last class to help the team meet the Sprint Goal
- What they plan to do between now and the next class
- Any impediments that will prevent the team from meeting the Sprint Goal

5-10 minutes

Exam Details

- Administered on gradescope
  - Practice exam we've been going through in class is available today through Friday as an example!
  - Start (the real exam) between 12:00AM on Wednesday and 9:45PM on Friday
- Print exam, write your answers **in the space given**, and scan
  - Recommendation: use the CamScanner app and *test creating a PDF with multiple pages before the exam!*

Exam Details

- All questions will be graded credit/no-credit
- There is some leeway:
  - You need to get 7/8 questions correct for an A in the course
  - You will have the opportunity to take a retest later in the semester if you so choose

Question 7 Spring 2024 Midterm #1

Assume you are developing a web application for supporting student clubs at a college, e.g., membership lists, calendars, announcements, etc. You will be using a relational database to store the data for this application.

(a) Identify the minimum set of models you would define in your server backend to implement the following user story:

> As a student, I want to view a consolidated a list of announcements for all the clubs I am a member of, so that I can stay informed about all club activities.

10 minutes to work on practice questions, 10 minutes to review

Question 7 Spring 2024 Midterm #1

Assume you are developing a web application for supporting student clubs at a college, e.g., membership lists, calendars, announcements, etc. You will be using a relational database to store the data for this application.

(b) Which of the following best describe the relations between the following pairs of entities:

Student and Club
- ◯ One-to-One
- ◯ One-to-Many
- ◯ Many-to-Many
- ◯ No relation

Club and Announcement
- ◯ One-to-One
- ◯ One-to-Many
- ◯ Many-to-Many
- ◯ No relation

Question 7 Spring 2024 Midterm #1

Assume you are developing a web application for supporting student clubs at a college, e.g., membership lists, calendars, announcements, etc. You will be using a relational database to store the data for this application.

(c) In a normalized schema designed for a relational database (RDBMS), what schema would be needed to support club membership. Assume club members can have different roles, e.g., "member", "president", etc. You do not need to provide SQL, just the attributes, their types, the primary key, and any foreign key constraints.

Question 8 Spring 2024 Midterm #1

For each of the following, indicate whether the action would be consistent with the best practices for software development as described in class or not consistent

(a) Each team member uses a separate "personal" branch throughout the sprint, e.g., mlinderman_sprint1, to implement their tasks before merging with main at end of the sprint.

(b) Assign a responsible developer for all the tasks in the sprint backlog during the sprint planning meeting.

(c) Update the main branch and rebase a newly created feature branch before pushing that feature branch to GitHub for the first time.