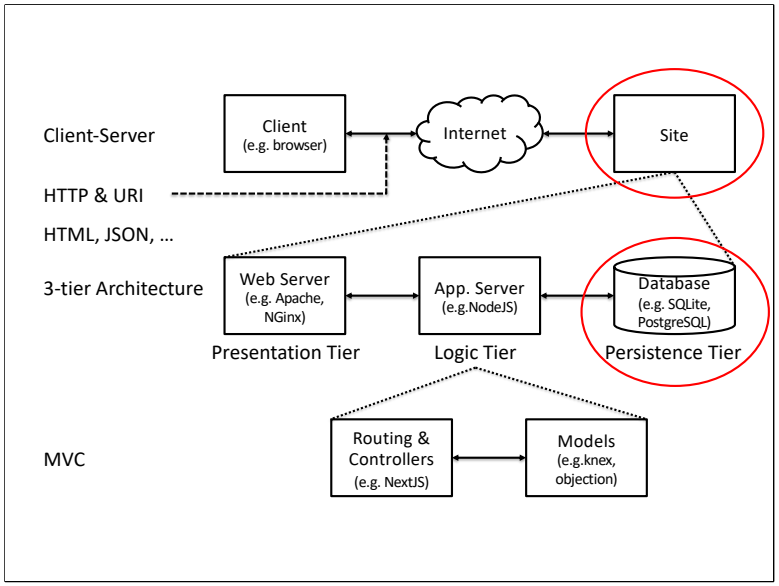


We previously introduced the role of the server: to provide persistence, a means for communicating between users and a secure environment (controlled by you) for operations that could/should not be performed by the untrusted client. Today we will focus of the first of those roles – persistence, that is saving data for future use. Although in doing so we will also touch on data integrity and related issues.

Why a database?

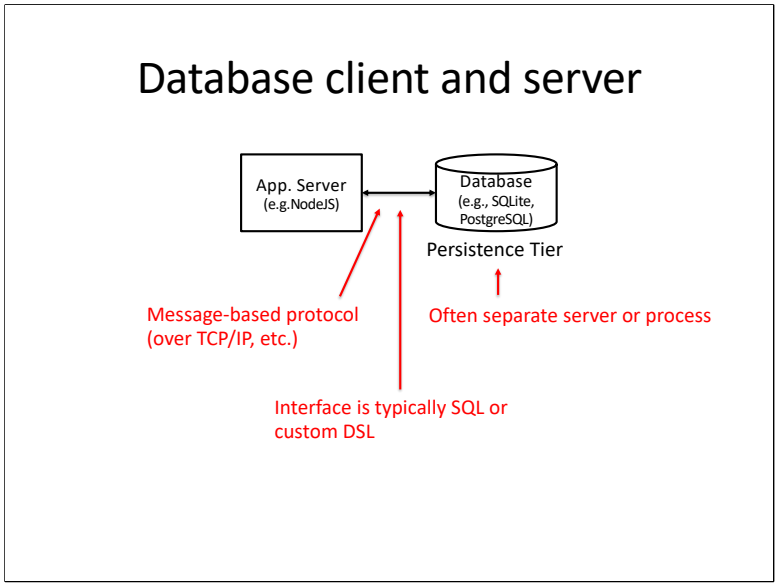
What are the limitations of the memory-backed server (or what missing features do we want)?

```
// pages/api/articles/[id].js
router.get((req, res) => {
  res.status(200).json(articles.get(req.query.id)); // not fully working code
});
```

- Efficient random access when total dataset is too large to fit in memory
- Fast *and* complex queries (not fast *or* complex)
- Model relationships within the data
- Transactions and other forms of fault tolerance
- Security (and management tools)

Our memory backed server worked (and was nice and simple) but had a major limitation, all modifications were lost when we reloaded the server. That is obviously not something we can tolerate in a real application. But the in-memory approach also has many other limitations that make it a poor choice for any real application.

What are those limitations, or perhaps from a more optimistic perspective, what are benefits of using a real database system as the persistence tier for our application (other than that when restart out server/the database the data is still there!) [clickl]



In the typical setup the database is its own process, and often running on a separate machine(s), and our application server (i.e., running on Node) communicates with the database server via TCP/IP or some other message-based protocol. That interaction occurs via SQL (a standardized language for querying relation databases) or a custom domain-specific language.

Although I should note that one of the databases we will use, SQLite, is not a server. It is a library that runs inside the "client" process, accessing a database stored entirely within a single file. As an aside, SQLite is one of the most widely-used pieces of SW ever written. It is embedded in basically everything, e.g., web browsers, iOS, etc.,.

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

	Relational (RDBMS)	Non-Relational
Data	Table-oriented	Document-oriented, key-value, graph-based, column-oriented, …
Schema	Fixed schema	Dynamic schema
Joins	Used extensively	Used infrequently
Interface	SQL	Custom query language
Transactions	ACID	CAP

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

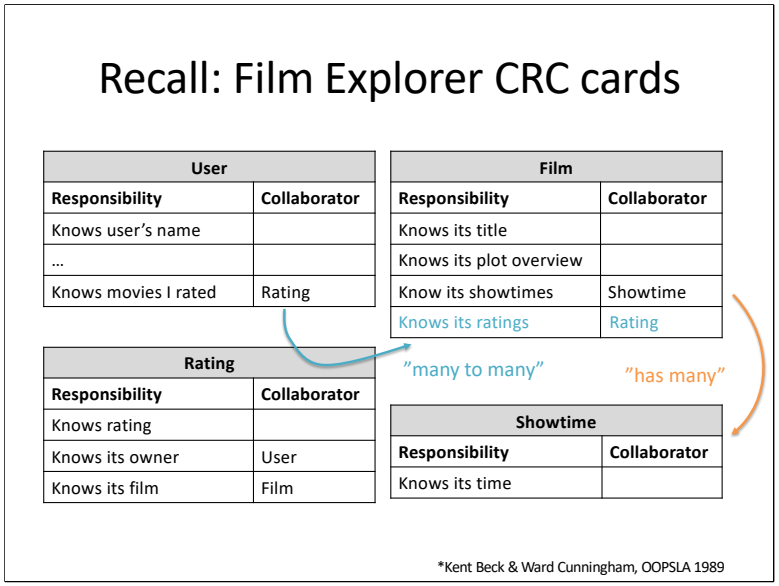
```
db.people.find(
  { age: { $gt: 25 } }
)
```

I just mentioned the term “relational database”. That is a term for a particular class of database management tools, it also sometimes referred to via “SQL”, which is really the name of the query language used by relational DBs. The alternative is often called “NoSQL”, or more appropriately “non-relational” databases. An example of the latter would be Google’s Firebase.

One of the decisions we will need to make in our project is what kind of database to use. Like many decisions we encounter in class there is no right answer – although the entire Internet will have an opinion – just tradeoffs. From my perspective, NoSQL is more flexible, and perhaps easier to get started, but the flexibility begets challenges in managing your data. In contrast, relational databases have a slightly steeper learning curve but force us to organize our data in helpful ways. A relevant analogy might be a statically-typed languages like Java (relational) vs. dynamic languages like Python (non-relational). The latter is quick to get going but is susceptible to type errors that are not possible in Java…

Before we can pick a database, however, we need to figure out how the data in our application is structured (i.e., determine the data model as discussed previously) and only then can we pick a database that make sense for our application. Again, from my perspective, a well-designed data model is more important than the choice of database system.

Glossary:
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.
CAP: Two of consistency (most recent data), availability, and partition tolerance.



Recall CRC cards are like user stories, but for classes. Each index card contains:

- On top of the card, the class name
- On the left, the responsibilities of the class, i.e., what this class "knows" and "does". For example, a "car" class may know how many seats and doors it has and could "do" things like stop and go.
- On the right, the collaborators (other classes) with which this class interacts to fulfill its responsibilities

The CRC cards help guide the design of our models and database schema. The "knows" are going to become the fields that we store in our database for each model and the knows/collaborators define the relationships or associations between those models. Recall that:

- A film has a one-to-many relationship with showtimes (i.e., film "has many" showtimes)
- There is a many-to-many relationship between Users and Films via the ratings, i.e.,, a film has been rated by many users, and a user has rated many films. This type of relationship is often called a "has many-through" association.

These terms are semi-formal (note different tools use slightly different names, but the concepts are the same), that is they map directly to the design of the database schema. We are effectively designing database tables as we work out these relations.

That said, I encourage you to approach the data modeling from this “direction”, that is start by modeling the nouns in your application (and their relationships) then choose and design your database instead of starting with the database design then developing the data model.

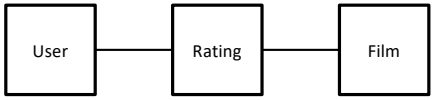
Thinking in relations/associations

- "HasOne" / "BelongsToOne"

  One-to-one relationship, e.g., Supplier and Account
- "HasMany" / "BelongsToOne"

  One-to-many relationship, e.g., Film and Showtime
- "ManyToMany"

  Many-to-many relationship (often called "has many through"), e.g., User and Film through Rating



The relations/associations you will typically encounter are listed here (again a quick reminder that different tools will use slightly different terminology, but the concepts are the same). We can think of these associations as design patterns that will enable us to utilized libraries/frameworks for the "parts that are the same every time", i.e., automatic validations, optimized queries and more.

A note: the "through" modifier can be applied to one-to-many relationships as well, and typically implies that the you might want to work with the "through" noun independently of the two sides of the relationship. We still would want to identify the relationship as one-to-many to take advantage of built-in validations.

You are developing an application for a veterinarian's office. How would you model the relation between Customer and Animal?

A. One-to-one, e.g., "HasOne"
B. One-to-many, e.g., "HasMany"
C. Many-to-many, e.g., "HasManyThrough"

Answer: B

A customer can have many animals (pets), but each animal is presumably owned by a single customer. Although we could imagine situations though where C might be needed… What would such an example be? Multiple customers were the owners/responsible parties for a pet.

We could imagine there is data associated with the specific Customer-Animal relation (e.g., insurance), however that each association may be its own entity doesn't itself change that it is a one-to-many relation.

True or False? Two models can only have one relation.

A. True
B. False

Answer: False

Consider an application where a user can make and like comments. A user has many comments via posting (a has-many relation), and user also has many comments via liking (a many-to-many or has-many-through relation).

Interlude: RESTful URLs for associations

What association is implied by this URL (from Gradescope)?

```
/courses/848148/assignments/5010276
```

A Course “has many” Assignments

Route	Controller Action
GET /courses/:course_id/assignments	Retrieve all assignments in associated course
POST /courses/:course_id/assignments	Create new assignment in associated course
…	

[click] The associations can be directly translated to URLs. That is a ”has many” relationship is typically expressed through nested URLs, and thus we infer ”has many” relationship. In this context we might describe assignments as a subordinate resource of a course. For viewing a single assignment this may not seem very compelling, as we could also retrieve the assignment (presumably) with just that id, e.g., //assignments/5010276. Where it might be more relevant is for POST, etc.

[click] Here the URL embeds the associated resource, i.e., we are creating a new assignment in the course indicated by the URL.

In theory we can go infinitely deep with this nesting, in practice we shouldn’t go more than one or two levels, otherwise it gets unwieldy.

RDBMS mental model

Noun/Model, e.g., "Film" ⇔ Table

Model attributes ⇔ Columns

Schema (name and type)

Film table

id	title	overview	release_date	poster_path	vote_average	rating
int	string	text	string	string	float	int
1	Star…	Princes…	1977-05-25	/tvSLB…	7.7	3
2	2001: A…	Huma…	1968-04-05	/90T7…	7.5	4

Primary key: Unique identifier for record (can be 1+ columns)

With our CRC cards we focused on modeling our data independent of how it is stored. We will now implement those models using a relational database. Our mental model is a table (e.g., a spreadsheet table). The attributes/columns are typically the "knows" in your CRC cards, that is the schema is a nearly direct translation of the CRC card. And the rows are specific entries, e.g., specific films.

The Primary Key is a unique identifier for a record (that should be not be reused). Often it is an arbitrary (auto-incrementing) integer, e.g., the "id" in Simplepedia, but does not need to be (and it can even be a composite of multiple columns) as long as it is unique. The schema includes type (storage size) and can further include indexes (think hash tables or trees) to speed up queries and other constraints, like not null.

RDBMS vocabulary

DB instance (e.g., PostgreSQL)

Has 0+

Databases

Has 0+

Tables

Contains 0+

Rows

With 1+

Attributes/Columns

Each table has a schema with types, optional primary key, optional constraints

Index
Optimized lookup tables (e.g., tree) for specific columns

Cursor
Iterator into the result set that can obtain a few documents at a time

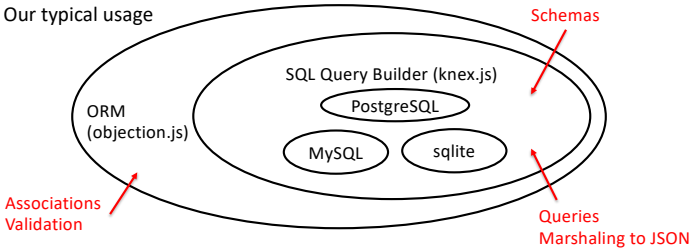
[click] Index, Cursor
[click] Each table has its own schema

Writing SQL queries

"Raw" queries

```
SELECT columns FROM table WHERE conditions;
INSERT INTO table(columns) VALUES (values);
UPDATE table SET column=value, … WHERE conditions;
CREATE TABLE table (column Type, …);
```

Our typical usage



We generally won't write "raw" queries, instead we will use the knex.js query builder to abstract DB-specific differences, handle "safe" parameters substitution, etc.. We will further wrap knex with an ORM library (Object Relational Mapping) that provides a more object-oriented interface to our database (stay tuned).

Managing Schema: Migrations

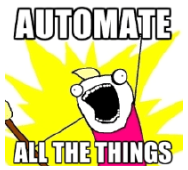
Customer data is critical! How do you evolve your application without destroying any data?

- Maintain multiple databases (e.g., test, development, production, …)
- Change schema/data with scripted *migrations*

Migrations create/delete tables, add/remove/modify columns, modify data, etc.

Advantage of migrations:

+ Track all changes made to DB
+ Manage with VCS
+ Repeatable



Migrations are the answer to how we smartly evolve our database schema at at all stages in our application lifecycle, from creating the initial database schema to safely evolving the database to add features to our production application (which presumably has customer data in it). While we could modify our database manually. We won't. Instead, we define a series of migrations scripts that evolve the schema from an empty database to the desired state.

Each migration has two parts, an "up" function that makes the desired changes, e.g., creating a table, adding column, etc. and the a "down" function that reverts those changes. Performing the up function and then the down function should return the database to its prior state. Each migration is incremental, that is it makes the "next" set of changes to the prior database/schema data. For example, if you add a feature that needs a new column in an existing table, we create a migration that adds that column (and sets an appropriate value for existing entries).

Migrations are a key part of our "overall" DevOps approach.

Frequent error: Migrations are tracked by date-time

```
migrations/20190424165216_users_and_articles.js
```

Date and time for this migration

Knex et al. apply migrations in *date-time order* and *track the last migration applied*

- Applying migrations multiple times won't have any effect
- Modifying and re-applying a migration won't have any effect

If you modify a migration, rollback then reapply

Why does it work this way? The goal for migrations is to enable to you to evolve a database that is in use and has customer data that you don't want to lose. Thus, we don't want to "double apply" the changes in a migration. And there isn't the expectation that we would go back and modify already implied migrations because that would invalidate the data.

Modifying a migration is common during development, however. If you do so, either delete the database and rebuild from scratch (easy with SQLite – just delete the file - less so with other RDBMSs) or more robustly, rollback the relevant migrations (invoking the "down" operations) then reapply the migrations after the making the edit.

Recall: Film Explorer CRC cards

User	
Responsibility	**Collaborator**
Knows user's name	
…	
Knows movies I rated	Rating

Film	
Responsibility	**Collaborator**
Knows its title	
Knows its plot overview	
Know its showtimes	Showtime
Knows its ratings	Rating

Rating	
Responsibility	**Collaborator**
Knows rating	
Knows its owner	User
Knows its film	Film

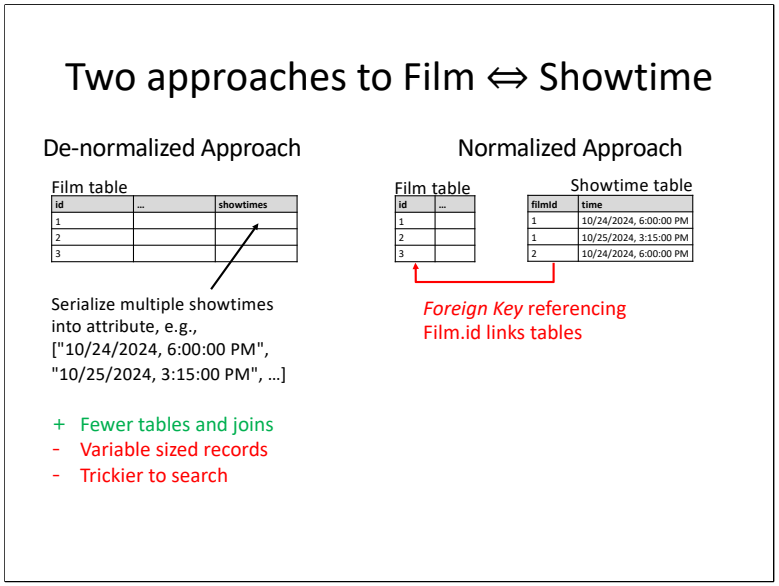
Showtime	
Responsibility	**Collaborator**
Knows its time	

"many to many"

"has many"

*Kent Beck & Ward Cunningham, OOPSLA 1989

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



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.