Why a database?

```json
[{
  "adult": false,
  "backdrop_path": "/sEgULSEnywgdSesVHFHpPAbOijl.jpg",
  "genre_ids": [18, 12, 878],
  "id": 286217,
  "original_language": "en",
  "original_title": "The Martian",
  "overview": "During a manned mission to Mars, Astronaut Mark Watney is presumed dead and left behind by his crew. But Watney has survived and finds himself stranded and alone on the hostile planet. With only meager supplies, he must draw upon his ingenuity, wit and spirit to subsist and find a way to signal to Earth that he is alive."
  "release_date": "2015-10-02",
  "poster_path": "/AjbENYG3b8lhYSkdrWwlhVLRPKR.jpg",
  "popularity": 40.509541,
  "title": "The Martian",
  "video": false,
  "vote_average": 7.7,
  "vote_count": 447
}, ...
]```

What’s wrong with our approach of loading data from a JSON file and storing it in memory?

```javascript
app.get('/api/films/:id', (request, response) => {
  response.send(films[request.params.id]);
})
```

Database Management Systems (DBMS)

- 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)

Database client and server

- Interface is typically SQL or custom DSL
- Often separate server or process
- Message-based protocol (over TCP/IP, etc.)
SQL vs. NoSQL
Really: Relational vs. Non-Relational

<table>
<thead>
<tr>
<th>Relational (RDBMS)</th>
<th>Non-Relational</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>Table-oriented</td>
</tr>
<tr>
<td>Schema</td>
<td>Fixed schema</td>
</tr>
<tr>
<td>Joins</td>
<td>Used extensively</td>
</tr>
<tr>
<td>Interface</td>
<td>SQL</td>
</tr>
<tr>
<td>Transactions</td>
<td>ACID</td>
</tr>
</tbody>
</table>

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

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

RDBMS vocabulary

**DB instance** (e.g. PostreSQL)
- Has 0+

**Databases**
- Has 0+

**Tables**
- Each table has a schema with types, optional primary key, optional constraints
- Contains 0+

**Rows**
- With 1+

**Attributes/Columns**

**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

RDBMS mental model

Noun/Model, e.g. “Film” ⇔ Table
Model attributes, e.g., “title” ⇔ Columns

```
<table>
<thead>
<tr>
<th>id</th>
<th>title</th>
<th>overview</th>
<th>release_date</th>
<th>poster_path</th>
<th>vote_average</th>
<th>rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Star...</td>
<td>Princes...</td>
<td>1977-05-25</td>
<td>/tvSLB...</td>
<td>7.7</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>2001: A...</td>
<td>Huma...</td>
<td>1968-04-05</td>
<td>/90T7...</td>
<td>7.5</td>
<td>4</td>
</tr>
</tbody>
</table>
```

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

SQL statements

```
SELECT columns FROM table WHERE conditions;
```

```
INSERT INTO table(columns) VALUES (values);
```

```
UPDATE table SET column=value, … WHERE conditions;
```

```
DELETE FROM table WHERE conditions;
```

```
CREATE TABLE table (column Type, …);
```

```
DROP TABLE table;
```

Example

```
SELECT title FROM Film WHERE rating >= 4;
```
SQLite SELECT Statement grammar

When in doubt, abstract!

Object-Relational Mapping (ORM)
(telegram.js)
Associations, Validation

Migrations, Queries

PostgreSQL
MySQL
sqlite

Example: Film posters

<table>
<thead>
<tr>
<th>id</th>
<th>title</th>
<th>overview</th>
<th>poster</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>Star Wars</td>
<td>&quot;Princess Leia is captured...&quot;</td>
<td>star-wars.jpg</td>
</tr>
<tr>
<td>105</td>
<td>Back to the Future</td>
<td>&quot;Eighties teenager Marty McFly...&quot;</td>
<td>back.jpg</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
**Example: Film posters**

<table>
<thead>
<tr>
<th>id</th>
<th>title</th>
<th>overview</th>
<th>poster</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>Star Wars</td>
<td>&quot;Princess Leia is captured...&quot;</td>
<td>star-wars1.jpg</td>
</tr>
<tr>
<td>11</td>
<td>Star Wars</td>
<td>&quot;Princess Leia is captured...&quot;</td>
<td>star-wars2.jpg</td>
</tr>
<tr>
<td>11</td>
<td>Star Wars</td>
<td>&quot;Princess Leia is captured...&quot;</td>
<td>star-wars3.jpg</td>
</tr>
<tr>
<td>11</td>
<td>Star Wars</td>
<td>&quot;Princess Leia is captured...&quot;</td>
<td>star-wars4.jpg</td>
</tr>
<tr>
<td>11</td>
<td>Star Wars</td>
<td>&quot;Princess Leia is captured...&quot;</td>
<td>star-wars5.jpg</td>
</tr>
<tr>
<td>105</td>
<td>Back to the Future</td>
<td>&quot;Eighties teenager Marty McFly...&quot;</td>
<td>back.jpg</td>
</tr>
</tbody>
</table>

**Normalizing tables**

<table>
<thead>
<tr>
<th>id</th>
<th>filmId</th>
<th>path</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>11</td>
<td>star-wars1.jpg</td>
<td>1977</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td>star-wars5.jpg</td>
<td>1992</td>
</tr>
<tr>
<td>23</td>
<td>105</td>
<td>back.jpg</td>
<td>1985</td>
</tr>
<tr>
<td>67</td>
<td>11</td>
<td>star-wars2.jpg</td>
<td>2002</td>
</tr>
</tbody>
</table>

**Primary key**

<table>
<thead>
<tr>
<th>id</th>
<th>filmId</th>
<th>path</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>11</td>
<td>star-wars1.jpg</td>
<td>1977</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td>star-wars5.jpg</td>
<td>1992</td>
</tr>
<tr>
<td>23</td>
<td>105</td>
<td>back.jpg</td>
<td>1985</td>
</tr>
<tr>
<td>67</td>
<td>11</td>
<td>star-wars2.jpg</td>
<td>2002</td>
</tr>
</tbody>
</table>

**Foreign key**

<table>
<thead>
<tr>
<th>id</th>
<th>filmId</th>
<th>path</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>11</td>
<td>star-wars1.jpg</td>
<td>1977</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td>star-wars5.jpg</td>
<td>1992</td>
</tr>
<tr>
<td>23</td>
<td>105</td>
<td>back.jpg</td>
<td>1985</td>
</tr>
<tr>
<td>67</td>
<td>11</td>
<td>star-wars2.jpg</td>
<td>2002</td>
</tr>
</tbody>
</table>
### Example: Film posters

Database **Joins** are formed by **Cartesian Products**

```
SELECT * from Film, Poster WHERE Film.id = Poster.filmId
```

<table>
<thead>
<tr>
<th>Film x Poster</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Film.id</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Poster.id</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Poster.filmId</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>5</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>9</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>23</td>
<td>105</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>67</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>105</td>
<td>5</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>105</td>
<td>9</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>105</td>
<td>23</td>
<td>105</td>
<td></td>
</tr>
<tr>
<td>105</td>
<td>67</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

### Example: Film posters

![Database Joins formed by Cartesian Products](example.png)

**SELECT** * from **Film**, **Poster** WHERE **Film.id** = **Poster.filmId**

<table>
<thead>
<tr>
<th>Film</th>
<th>Responsibility</th>
<th>Collaborator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Knows its title</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Knows its plot overview</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Know which genres it is</td>
<td>Genre</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Poster</th>
<th>Responsibility</th>
<th>Collaborator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Knows its path</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Knows its date</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Knows its artist</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Knows its film</td>
<td>Film</td>
<td></td>
</tr>
</tbody>
</table>

### Example: Film Ratings

<table>
<thead>
<tr>
<th>User</th>
<th>Responsibility</th>
<th>Collaborator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Knows user's name</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Knows films I rated</td>
<td>Rating</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rating</th>
<th>Responsibility</th>
<th>Collaborator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Knows rating</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Knows its owner</td>
<td>User</td>
<td></td>
</tr>
<tr>
<td>Knows its film</td>
<td>Film</td>
<td></td>
</tr>
</tbody>
</table>

### 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 Poster

- **“ManyToMany”**
  Many-to-many relationship (often called “has many through”), e.g. User and Film through Rating

---

*Example: Film posters with returned rows and one-to-many relationship.*
Where do the foreign keys go?

• “HasOne” / “BelongsToOne”  
  Foreign key typically in the “BelongsToOne” side (although could be reversed)

• “HasMany” / “BelongsToOne”  
  Foreign key in “BelongsToOne” side (the “many” model)

• “ManyToMany”  
  Foreign keys in join model, e.g. Rating in “User and Film through Rating”

True or False? There can only be one relationship between two models.

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”

Specifying 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
Example Migration

```javascript
exports.up = function(knex, Promise) {
    return knex.schema.createTable('Film', table => {
        table.integer('id').unsigned().primary();
        table.text('overview');
        table.string('release_date');
        table.string('poster_path');
        table.string('title');
        table.float('vote_average');
        table.integer('rating');
    });
}
exports.down = function(knex, Promise) {
    return knex.schema.dropTableIfExists('Film');
};
```

Object Relational Mapping (ORM)

![Object Relational Mapping (ORM) Diagram](image-url)

```
// ORM Model
class Film extends Model {
    static get tableName() {
        return 'film';
    }
    static get jsonSchema() {
        return {
            type: 'object',
            required: ['id', 'overview', 'release_date', 'poster_path', 'title', 'vote_average'],
            properties: {
                id: { type: 'integer' },
                overview: { type: 'text' },
                release_date: { type: 'string' },
                poster_path: { type: 'string' },
                title: { type: 'string' },
                vote_average: { type: 'number' },
                rating: { type: 'integer', 'null': true },
            },
        };
    }
    static get idColumn() {
        return ['filmId', 'genreId'];
    }
    static get relationMappings() {
        return {
            film: {
                relation: Model.BelongsToOneRelation,
                modelClass: path.join(__dirname, 'Film'),
                join: {
                    from: 'Genre.filmId',
                    to: 'Film.id'}},
        };
    }
}
// ORM Model
class Genre extends Model {
    static get tableName() {
        return 'genre';
    }
    static get idColumn() {
        return ['filmId', 'genreId'];
    }
    static get relationMappings() {
        return {
            film: {
                relation: Model.BelongsToOneRelation,
                modelClass: path.join(__dirname, 'Film'),
                join: {
                    from: 'Genre.filmId',
                    to: 'Film.id'}},
        };
    }
}```