ECE464 Databases:
Movie Reservation System

Andy Jeong, Kevin Lin, Ali Rahman

December 17, 2019

Abstract

In this project, we design a movie reservation system on a non-relational database management system (MongoDB). This system resembles a typical movie theater website, such as AMC, allowing users to query for showing movies at chosen theaters and make reservations for movies and food. The data structure for the database is document-based in order to allow for various types of attribute values, which are of boolean, string and array types for our data. Upon entering the application, users can create accounts, login, search for movies at specific theater locations, and make seating and food reservations for the selected movies. We exploit MongoDB's built-in sharding support, which allows to distribute read and write workloads across the sharded clusters and efficiently process operations.

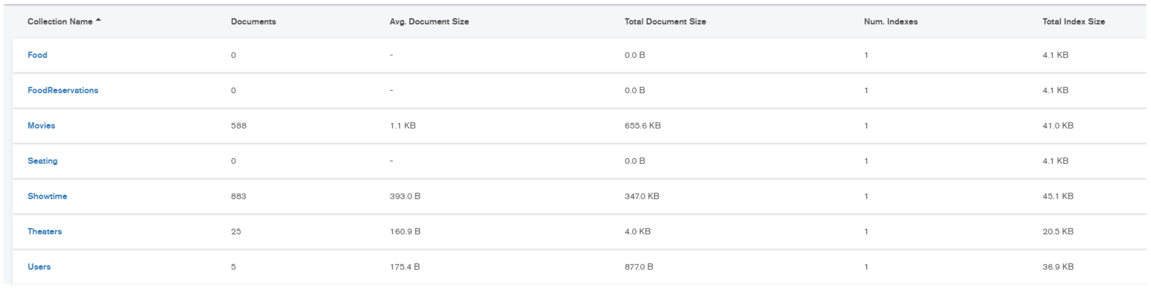
Contents

1 Overview

The movie reservation system is designed from data scraped from a number of existent movie websites, such as IMDB, AMC and RottenTomatoes (see Table 1). Each movie information (e.g. ‘movies’ table) contains title, consensus, critic rating, critic count, audience count, text description, rating, genre, director, writer, air date, runtime and studio; each theater information (e.g. ‘theater’ table) contains the address, phone number and theater name; each movie showtime information (e.g. ‘showtimes’ table) contains references to the movie and theater, time of the show, type of theater (e.g. Dolby, 2D) and available seats for the particular movie at the theater. All web-scraped data are populated, cleaned on MongoDB Compass for selecting useful attributes, and queried by filtering multiple collections by criteria. In the database named ‘AKA,’ we have collections for food, food reservations, movies, seating, showtime, theaters, and users (see Figure 1 and 3 in Appendix A.1). The application side interface is designed as shown in Figure 2.

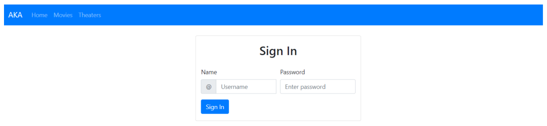
	Website	**Web Address**
1	IMDB	https://www.imdb.com
2	Rotten Tomatoes	https://www.rottentomatoes.com
3	AMC	https://www.amctheatres.com

Table 1: Web-Scraped Sites

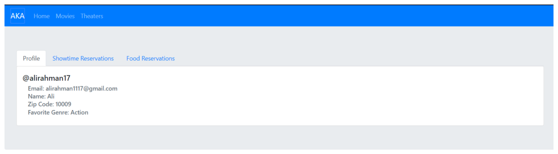


Collection Name ▲	Documents	Avg. Document Size	Total Document Size	Num. Indexes	Total Index Size
Food	0	-	0.0 B	1	4.1 KB
FoodReservations	0	-	0.0 B	1	4.1 KB
Movies	588	1.1 KB	655.6 KB	1	41.0 KB
Seating	0	-	0.0 B	1	4.1 KB
Showtime	883	393.0 B	347.0 KB	1	45.1 KB
Theaters	25	160.9 B	4.0 KB	1	20.5 KB
Users	5	175.4 B	877.0 B	1	36.9 KB

Figure 1: Database Collections



(a) Homepage



@alirahman17
Email: alirahman1117@gmail.com
Name: Ali
Zip Code: 10009
Favorite Genre: Action

(b) User page

Figure 2: Front-end User Interface

1.1 Tech Stack

- MongoDB

- **Schema-less** MongoDB allows for collections to hold documents containing different field types and sizes
- **Clear Object Structure**
- **No complex joins**
- **Deep query-ability:** dynamic query support on documents using a document-based query language as powerful as SQL
- **Scalability** MongoDB is relatively easier to scale than MySQL
- **No Conversion/mapping of application objects to database objects**
- **Internal distributed clusters** for efficient data store and retrieval

- Spark Framework (Java)

- **Speed:** Spark is a thin wrapper around Java EE's Servlet API, ranked highly on industry benchmarks where not only Java is tested but many programming languages and toolkits
- **Productivity:** Spark allows developers to be more productive by providing a simple DSL for routing your API's endpoints to handlers
- **Built for Routing:** Spark is designed to handle routing very well
- **Cloud-Ready:** Spark is a great, lightweight alternative to other, heavier frameworks, making it perfect for applications throughout the entire cloud stack

- ReactJS

- **Easy to learn, adapt, and use**
- **Reusable components**
- **Ample resources for development**
- **Virtual DOM:** React provides a virtual DOM, which is a more effective way to

make these changes and then apply only those changes to the real DOM. This guarantees a minimum update time to the real DOM, providing higher performance and a cleaner user experience all around.

2 Changes in Functionalities

The initial proposed approach for data collection via web scraping was taken on the websites listed on Table 1. From each site, we scraped information about details of movies from the past five years, theater information for currently showing movies, and current available showtimes. The general query functionality for a movie and a theater meets our initial goal; we were able to set up a seating chart from which users can select and reserve and update as requests for reservations arrive at the server. Also, the schema has been changed in order to accommodate for the collected data in a more efficient manner in a NoSQL. The updated schematic representation are shown below.

Schematic Representation

```
- users(user_id, name, username, password, email, admin, created_at,
        favorite_genre, zip_code)
- seating(username, showtime_id, seat_num)
- theater(theater_id, theater_address, theater_phone, admin_id)
- food_item(food_id, food_desc)
- showtimes(showtime_id, movie_id, date, time, type, theater_id,
            seats)
- food_reservation(username, showtime_id, food_id, count)
- movies(movie_id, title, consensus, critic_rating,critic_count,
        audience_count, description, rating, genre, director,
        writer, air_date, runtime, studio)
```

2.1 Challenges

For web scraping data, one difficulty we countered was the limited number of HTTP GET requests to certain websites when requests are made from the same address (e.g. Rotten Tomatoes). In addition, since working with a too large database can potentially give overheads and slow access speeds in query execution, we limited the size of scraped data to 25 theaters (near zip code 10003), 883 current movie showtimes and 588 movies from past five years (see Figure 1). From these separately collected data, we find and return overlapped information and/or available information. With this large, but manageable size of our data designed to demonstrate a user-oriented database system, we thought this would be sufficient to demonstrate the core functionalities. For modes of access, we limited the permissions for a user to read-only, without options to work in the administrative mode for each individual theater, although we maintain a central database ‘admin.’ This administrator would be able to perform CRUD operations on any movie showtimes, theaters, or reservations. In terms of multiple screens within a theater, we decided to assume that one unique screen is available for each movie at its theater. Allocating a screen for each movie show would require avoiding conflicting screens by time, which would need additional runtime and start/end time comparisons for each theater, thereby causing additional computational overheads. We would consider this screen conflict avoidance one of the next steps. As we were building the front- and back-end, another challenge we needed to resolve was the format of the response body content that is to be delivered to the application side. While it would be ideal to send an object payload, we decided to send JSON-formatted text and have the front-end parse according to the type of object, considering there are only a limited number of objects we are sending.

3 Conclusion

The most important aspect of this reservation system would certainly be the data collected across the three websites, as the database and the application is structured around the scraped data entirely. With this in mind, we spent enough time to understand what kinds of data we have available from our web scrapers, and think about how to merge, clean, retrieve from the collections efficiently. Using MongoDB in place of a type of RDBMS allowed us to manipulate with data structures in a more flexible manner, and on the backend it was a good learning experience using the Spark framework. We also realized the limitations in our available data and considered how much overhead each query would result in throughout the development process.

Appendix A Schema Representation

A.1 Schema Diagram

Although this schema structure no longer applies to our application using MongoDB, we show it for the purpose of understanding the data types and relationships. We modified our initial proposed schema to match our current structure.

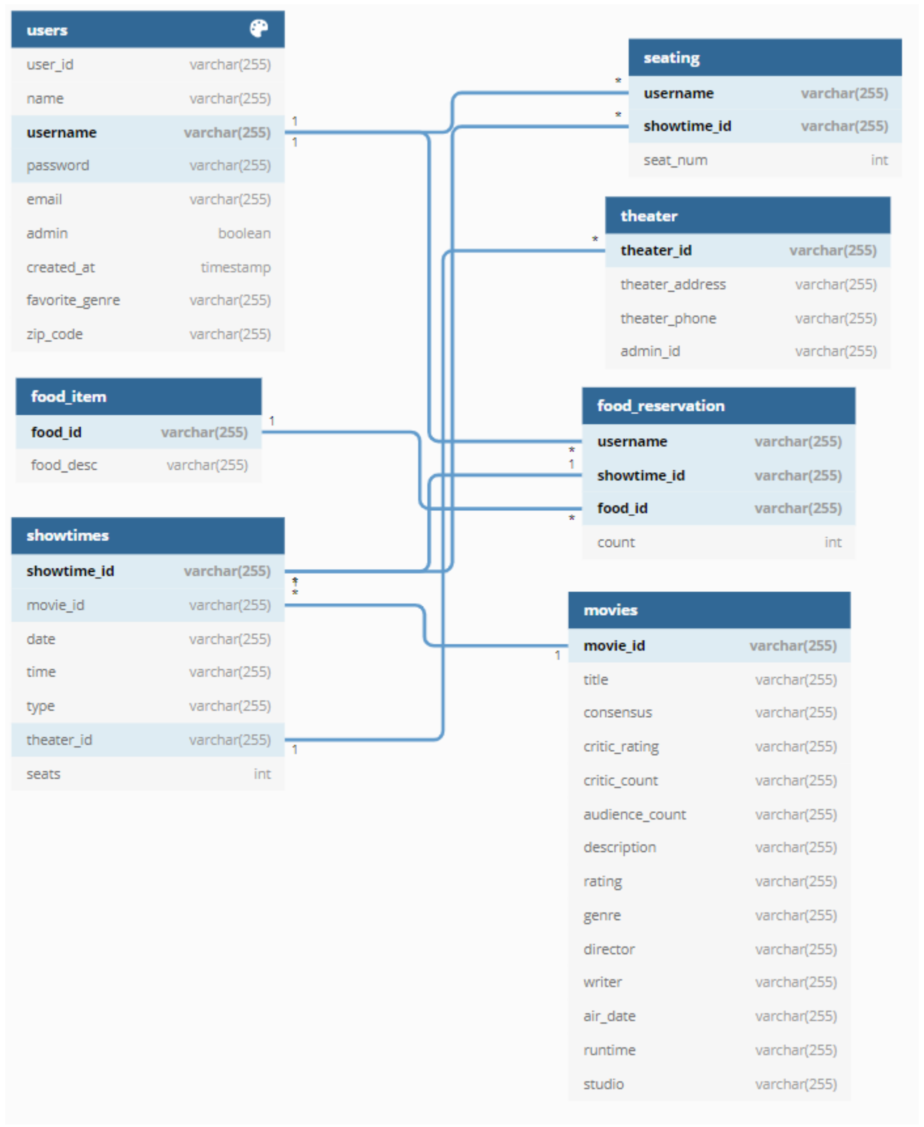


Figure 3: Relational Schema Diagram

A.2 SQL queries for schema

```
CREATE TABLE `users` (
  `user_id` varchar(255),
  `name` varchar(255),
  `username` varchar(255) PRIMARY KEY AUTO_INCREMENT,
  `password` varchar(255),
  `email` varchar(255),
  `admin` boolean,
  `created_at` timestamp,
  `favorite_genre` varchar(255),
  `zip_code` varchar(255)
);

CREATE TABLE `theater` (
  `theater_id` varchar(255) PRIMARY KEY,
  `theater_address` varchar(255),
  `theater_phone` varchar(255),
  `admin_id` varchar(255)
);

CREATE TABLE `showtimes` (
  `showtime_id` varchar(255) PRIMARY KEY,
  `movie_id` varchar(255),
  `date` varchar(255),
  `time` varchar(255),
  `type` varchar(255),
  `theater_id` varchar(255),
  `seats` int
);

CREATE TABLE `seating` (
  `username` varchar(255),
  `showtime_id` varchar(255),
  `seat_num` int,
  PRIMARY KEY (`username`, `showtime_id`)
);
```

```
CREATE TABLE `movies` (
  `movie_id` varchar(255) PRIMARY KEY,
  `title` varchar(255),
  `consensus` varchar(255),
  `critic_rating` varchar(255),
  `critic_count` varchar(255),
  `audience_count` varchar(255),
  `description` varchar(255),
  `rating` varchar(255),
  `genre` varchar(255),
  `director` varchar(255),
  `writer` varchar(255),
  `air_date` varchar(255),
  `runtime` varchar(255),
  `studio` varchar(255)
);

CREATE TABLE `food_reservation` (
  `username` varchar(255),
  `showtime_id` varchar(255),
  `food_id` varchar(255),
  `count` int,
 PRIMARY KEY (`username`, `showtime_id`, `food_id`)
);

CREATE TABLE `food_item` (
  `food_id` varchar(255) PRIMARY KEY,
  `food_desc` varchar(255)
);

ALTER TABLE `seating`
ADD FOREIGN KEY (`username`)
REFERENCES `users` (`username`);

ALTER TABLE `food_reservation`
ADD FOREIGN KEY (`username`)
REFERENCES `users` (`username`);
```

```
ALTER TABLE `showtimes`
ADD FOREIGN KEY (`movie_id`)
REFERENCES `movies` (`movie_id`);

ALTER TABLE `theater`
ADD FOREIGN KEY (`theater_id`)
REFERENCES `showtimes` (`theater_id`);

ALTER TABLE `seating`
ADD FOREIGN KEY (`showtime_id`)
REFERENCES `showtimes` (`showtime_id`);

ALTER TABLE `showtimes`
ADD FOREIGN KEY (`showtime_id`)
REFERENCES `food_reservation` (`showtime_id`);

ALTER TABLE `food_reservation`
ADD FOREIGN KEY (`food_id`)
REFERENCES `food_item` (`food_id`);
```