Task 2 – Movie Database

Final Report

Federico Fregosi, Mirko Laruina,

Riccardo Mancini, Gianmarco Petrelli

June 4, 2020

Contents

1 Specifications

1.1 Application Overview

The application is an aggregator of movies and movie ratings with the purpose of providing logged users statistics and informations about a large set of movies. Logged-in user can also rate movies they have watched while not logged-in users may still use the service to browse movie rankings and statistics but they are not able to give their rate. Only movies released in Italy are considered.

All users can search a movie and view its details (e.g., title, original title, duration, cast, ...) along with its average rating from users and from external sources.

In addition, all users can browse the list of movies sorting and filtering it by many parameters (e.g. year, genre, country, actors, ...).

System administrators can view all user profile pages and ban users. In order to do that, he can check the full history of ratings. Once a user is banned, he can no longer log in and his username and email cannot be used by new users.

The movie database will be built upon the publicly available IMDb dataset.

The ratings will be gathered also by periodically scraping external websites (e.g., Rotten Tomatoes, Coming Soon, MyMovies).

1.2 Actors

Anonymous user, registered user, administrator and updater “bot”.

1.3 Requirement Analysis

1.3.1 Functional Requirements

An **anonymous user** must be able to register in order to become a *registered user*. Login is carried out using username and password selected by the user when registering. Username must be unique. A valid email address is also required in order to register. An email cannot be used more than once.

Both **anonymous user** and **registered user** must be able to:

- view details and average rating of a specific movie
- view a list of movies and filter it by many parameters. Combined filters are also allowed
- view aggregated statistics about movies: the user can choose on which field to aggregate movies (year, country, actor, director, genre) and additional filters (like the movie browsing feature). E.g. the user might want to see the ranking of the countries with the best movies in the last 10 years.

A **registered user** must be able to rate a movie, in addition to what anonymous user can do. A registered user must also be able to manage his profile. In the profile a registered user can:

- check, add and modify his personal data
- browse the history of his rates

- view aggregated statistics about his profile (i.e. most viewed genre, most recurrent actor, etc...) based on his rated movies
- delete the account

Finally, a registered user can logout in any moment.

An **administrator** is a special registered user who must be able to ban users. In order to do that, an administrator can check a global rating history to retrieve information about all the application's activity, and to check every user's profile. Banned user's rating are automatically removed from the database. Email and username of banned users cannot be used again.

The **updater "bot"** is not a real user but an entity used to periodically update the database in order to add new movies and update external ratings.

1.3.2 Non-Functional Requirements

- **Availability**: the Database must be replicated in order to be always available. Write operations on the Database can be eventually consistent.
- **Scalability**: the application must be able to scale to an arbitrary number of servers.
- **Security**: passwords must be stored in a secure way.
- **Responsive UI**: Client-side application must provide a responsive view both for pc, laptops and mobile devices.

2 Design

2.1 Use-case diagram

The use-case diagram is shown in Figure 1. Different colors are used to highlight cases that are exclusive of some actors: white cases are referred to all users; blue cases are referred to registered user and admin; yellow cases are exclusive of the admin.

2.2 Class diagram

The class diagram is shown in Figure 2. It was decided to show *Country*, *Year* and *Genre* as separate entities as they are some of the fields over which aggregate statistics are calculated.

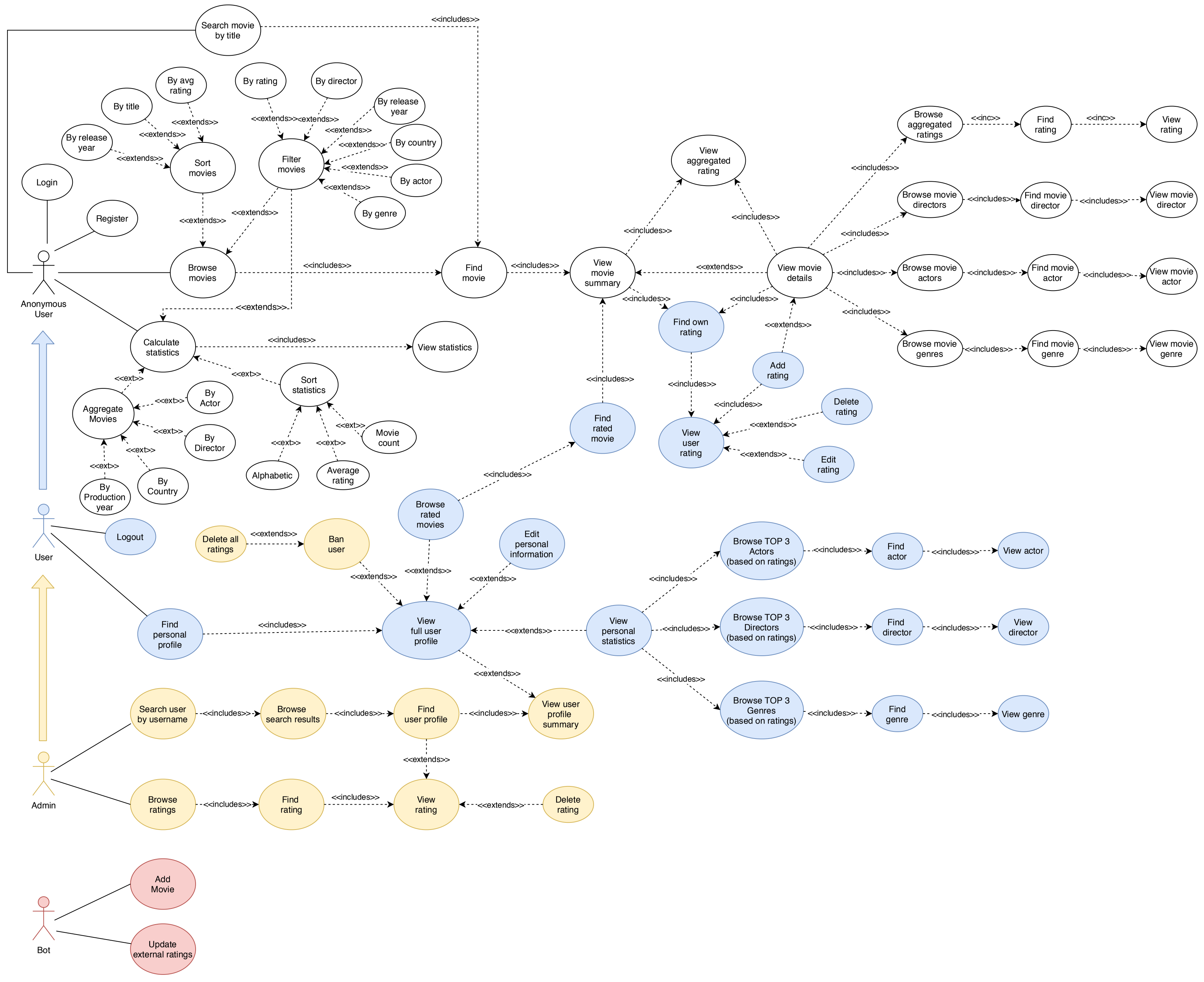


Figure 1: Use-case diagram

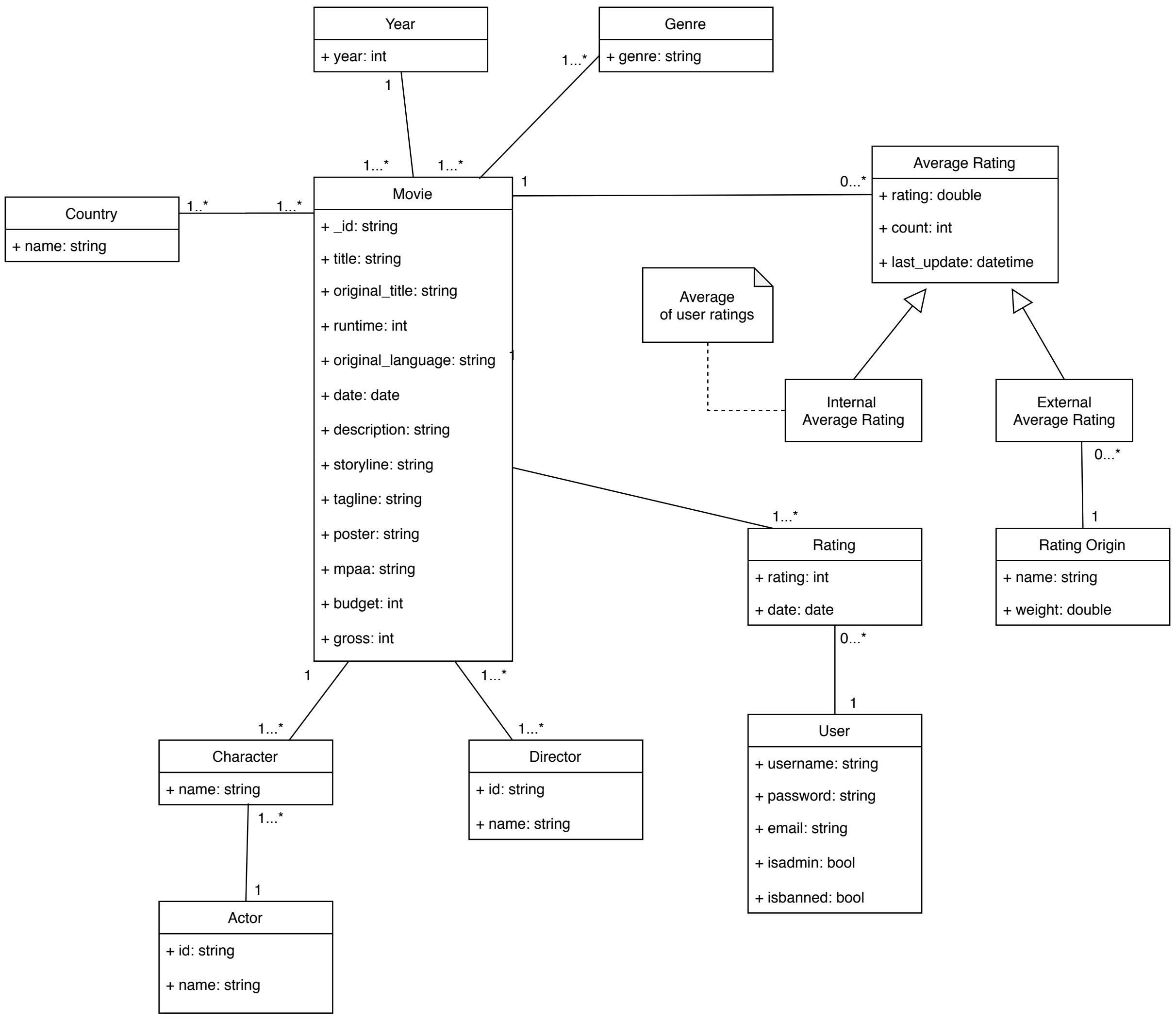


Figure 2: Class diagram for the identified entities

2.3 Data model

The data model is split in 3 different collections:

- Movies (Section 2.3.1)
- Users (Section 2.3.3)
- Ratings (Section 2.3.2)

In each following subsection, an example document is shown for every collection.

2.3.1 Movies

```
{
    "_id": "tt7286456",
    "title": "Joker",
    "original_title": "Joker",
    "runtime": 122,
    "countries": ["USA", "Canada"],
    "original_language": "English",
    "year": 2019,
    "date": "2019-10-04",
    "description": "In Gotham City, mentally troubled comedian [...]",
    "storyline": "Joker centers around an origin of the iconic arch [...]",
    "tagline": "Put on a happy face.",
    "poster": "https://m.media-amazon.com/images/M/[...].jpg",
    "mpaa": "Rated R for strong bloody violence, disturbing behavior, [...]",
    "budget": 55000000,
    "gross": 1074251311,
    "characters": [
        {
            "name": "Joker",
            "actor_name": "Joaquin Phoenix",
            "actor_id": "nm0001618"
        },
        ...
    ],
    "directors": [
        {
            "id": "nm0680846",
            "name": "Todd Phillips",
        }
    ],
    "genres": ["Crime", "Drama", "Thriller"],
    "ratings": [
        {
            "source": "internal",
            "sum": 450,
            "count": 100,
            "weight": 1,
            "last_update": "2019-11-28 12:34:56"
        },
        {
            "source": "IMDb",
            "avgrating": 8.6,
            "count": 628981,
            "weight": 0.5,
            "last_update": "2019-11-28 12:34:56"
        },
        ...
    ],
    "total_rating": 4.43,
    "last_scraped": "2019-11-28 12:34:56"
}
```

Notes Character and director nested documents contain redundant data (person's name) that is introduced to reduce the number of seeks necessary to return the movie details to the user.

This collection is mainly read-heavy since most fields will be written only once. The only fields subject to change are the ones related to ratings. For these reasons, we can afford many indices to speed-up the aggregations.

Indices Indices on the following fields will be set for this collection:

- `title`: speed-up sorting by title
- `title`, `original_title` (text index): movie lookup by name
- `countries` (multi-key index): filter and aggregation by country
- `genres` (multi-key index): filter and aggregation by genre
- `year`: filter and aggregation by year
- `date`: sorting by release date
- `characters.actor_id` (multi-key index): filter and aggregation by actor
- `directors.id` (multi-key index): filter and aggregation by director
- `total_rating`: filter and sorting by rating
- `last_scraped`: fast lookup for next-to-be-scraped movie (oldest movie).

Among this fields, *total_rating* is the only one that will be updated frequently (all other fields with index are updated with the periodic scraper whose period is of some days). However it is necessary to efficiently fulfill many queries.

2.3.2 Ratings

```
{
    "_id": {
        "user_id": <ObjectId>,
        "movie_id": "tt7286456"
    },
    "date": "2020-01-20",
    "rating": 5
}
```

Notes The raw ratings are kept separated from the movie they refer for three reasons:

1. single ratings are never shown to the user, except when the user looks his own ratings or when the admin looks all ratings
2. they may be accessed per-user, per-movie (for calculating statistics) and globally (by the administrator). The optimization for one use-case would influence the performance of another.
3. new ratings are expected to be frequently added and, thus, the use of a nested document inside the *movies* collection would be infeasible because it would grow indefinitely.

However, this choice mandates the introduction of indices for efficiently obtaining ratings of a single user and for getting the most recent ratings (admin use-case).

Indices

- `_id.user_id`: speed up user ratings browse.
- `date`: speed up browsing user and global ratings.

2.3.3 Users

```
{
    "_id": <ObjectId>,
    "username": "joker",
    "password": "<HASHED PASSWORD>",
    "email": "joker@dccomics.com",
    "isadmin": True (optional),
    "isbanned": False (optional),
    "sessions": [
        {
            "session_id": "<session_id>",
            "expiry": "2020-01-28"
        },
        ...
    ]
}
```

Notes *Username*, *email* and *session_id* must be unique across the whole collection.

Indices

- `username` (text index): find user by username
- `username`: find user by exact username
- `email`: find user by email
- `sessions.session_id` (multi-key index): find user by session id

2.4 Software Architecture

The application will be made of the following 4 components:

- **MongoDB**: a MongoDB cluster will be deployed with replication.
- **React Front-end**: web-based UI.
- **Java Back-end**: using *Spring*, the *Java back-end* will provide REST APIs to the *React front-end*.
- **Updater "bot"**: two Python scripts are needed in order to nightly update the DB with the latest movies and ratings:
    1. the **IMDb parser** periodically parses the IMDb dataset to add the latest movies.
    2. the **scraper** continuously parses the rating sources to update the ratings.

  These scripts will executes asynchronously from the *Java back-end*.

3 Database

3.1 Creation of the database

The database is created from scratch starting from the public IMDb dataset available at the link https://www.imdb.com/interfaces/.

We developed a Python script, in order to obtain the data in the form described in the Data Model. However, some information where missing. So we integrated missing information scraping them from the external rating sources while we were scraping for ratings.

Movies are saved into Json files and then "upserted" in MongoDB database with another simple python script with the use of the **UpdateOne** module. Update requests are appended to an array and served through the **bulk_write()** function.

By enabling the *upsert* functionality, we can reuse the same script to update the movie collection since movies that are not in the DB will be inserted and movies that are already present will be updated. Note that it is rare that a movie information will change so most of the time the *upsert* will do nothing to the stored movies.

3.2 Scraping bot

The aim of this bot, developed in Python (3.7) language, is to solve the problem of lack of data, indeed Movienator app needs a lot of movie information and details to work sensibly, so the scraper works providing from two main Web sources, MyMovies and IMDb, all types of data required, according to the ETL ("Extract-Transform-Load") paradigm.

We can identify two main uses of the scraper, as follow:

1. Find missing sensible informations for stored movies.
2. Update movies with most recent data, this operation is carried out periodically on each movie.

Below are listed all the attributes collected by the scraper for each movie:

- movie title;
- movie genre;
- URL to movie poster;

- movie description;
- public release date;
- aggregate rate of external source;
- extra attributes (storyline, tag-line, production country, budget ..)

3.2.1 Software design according to ETL model

Data extraction The very first step is data extraction, made possible by sequential HTTP requests (performed by Requests module), followed by the utilization of Beautiful Soup, a Python library for pulling data out of HTML and XML files. Data, among different http responses, are retrieved by a JSON-LD tag, contained within html body of every single movie page.

```python
def get_ld_json(self, url: str) -> dict:
    '''
    This method return parsed info as a dict by scraped page
    '''

    parser = "html.parser"
    req = requests.get(url)
    print(req.encoding)
    if self.source == "mymovies":
        text = str(req.content, 'UTF-8', errors='replace')
    else:
        text = req.text
    soup = BeautifulSoup(text, parser)
    print(url)

    if self.source == "imdb":
        ld_json = soup.find("script", {"type":"application/ld+json"})
        if ld_json is not None:
            json_dict = json.loads(normalize_json_string("".join(ld_json.contents))
    )
            json_dict.update(self.find_additional_info_from_imdb(soup))
            return json_dict
        else:
            return None
    elif self.source == "mymovies":
        json_scripts = soup.findAll("script", {"type":"application/ld+json"})
        for script in json_scripts:
            obj = json.loads(normalize_json_string(script.getText()))
            if ("name" in obj.keys() and "genre" in obj.keys() ):
                return obj
            else:
                continue
        return None
```

Web indexing is made agile through two separate techniques:

1. Across IMDb pages, every movie page is uniquely identified by imdb_id, exploiting a standard URL format, "*https://www.imdb.com/title/ttxxxxxxx/*", where *"xxxxxxx"* is the integer id to locate the source. Since our database was built from IMDb, our DB uses the IMDB id, therefore we can just fetch the correct page.

2. Exploiting MyMovies Web API, it's possible to find movies URL starting from their titles; "*https://www.mymovies.it/ricerca/ricerca.php?limit=true&q=yyyyyyy*", this is the query format, where the real argument (passed by GET method) is actually just one, "*q*" and "*yyyyyyy*" represents movie title (in Italian).

3.2.2 Data transformation

All the input data, retrieved from Web, are parsed into Python dictionaries, exploiting json library methods applied to de-serialized data content, which is obtained by the *"soup"* in the previous phase.

```python
def LoadMovie(self,movieId):
    """valid format only for imdb,mymovies
    ---req_movie_path = self.source + str(movieId)
    ---req_movie_path = https://www.mymovies.it/film/yyyy/title/

    movieId can be an integer ->IMDB, a string like
    'https://www.mymovies.it/film/yyyy/title/'->mymovies
    """
    if movieId.startswith("https://www.mymovies.it/"):
        self.source = "mymovies"
        """add tomato"""
    else:
        self.source = "imdb"
        movieId = web_sources[self.source] + str(movieId)
    #getting json results
    req_movie = self.get_ld_json(movieId)
    if req_movie == None:
        return
    #print(req_movie)

    #miss values manager
    for key in MovieScraper.Attributes + MovieScraper.EXTRA_ATTRIBUTES:
        if key not in req_movie.keys():
            req_movie[key]= None

    nt = {"source":str(self.source), "movie":{} } #new dict element

    #listing attributes separately
    for key in MovieScraper.Attributes + MovieScraper.EXTRA_ATTRIBUTES:
        # gestire il name nel formato opportuno per ricavare una lista di
        # nomi compatibile rottent e mymovies
        exec("self."+key+".append(req_movie[key])" ) in locals()
        nt["movie"][key] = req_movie[key]
    #listing attributes in tuples
    """
    new_tuple= tuple(nt)
    self.moviesInfo.append(new_tuple)
    """
    #getting attributes movie matrice
    #print(nt)
    new_movie_doc = nt
    self.moviesInfo.append(new_movie_doc)
    return new_movie_doc
```

In the listing above, we can look at the code that performs a first processing of data, retrieved by the dictionary resulting from the de-serialization of the json content. Missing features are converted in properties with *None* as a value, the objective is to deal with standard format dictionaries, solving the problem of heterogeneous data sources.

3.2.3 Data loading: updating the database

New scraped data, both for features update and addition, once parsed, are stored on MongoDB server by means of PyMongo API for Mongo DBMS. BSON is the standard format used to store documents and make remote procedure calls in MongoDB, luckily python dictionaries are directly mapped to BSON type as object and vice versa, so serialization (writings to Mongo) and de-serialization (readings from Mongo) processes are easily managed.

In order to perform the update on the least recently scraped movies, a *last_scraped* attribute is added (and modified) every time an update operation is executed from the script. The code listing below shows how to process consecutive movies, getting them by *self.getMoviesByLastScraped* method and proceeding on integrating scraped data. The scraper first marks the movie as scraped by updating the *last_scraped* attribute and then checks whether another scraper is scraping the same movie by comparing the *last_scraped* attribute that the *find_one_and_update* method returned (original DB document) with the one that the scraper retrieved earlier from the DB.

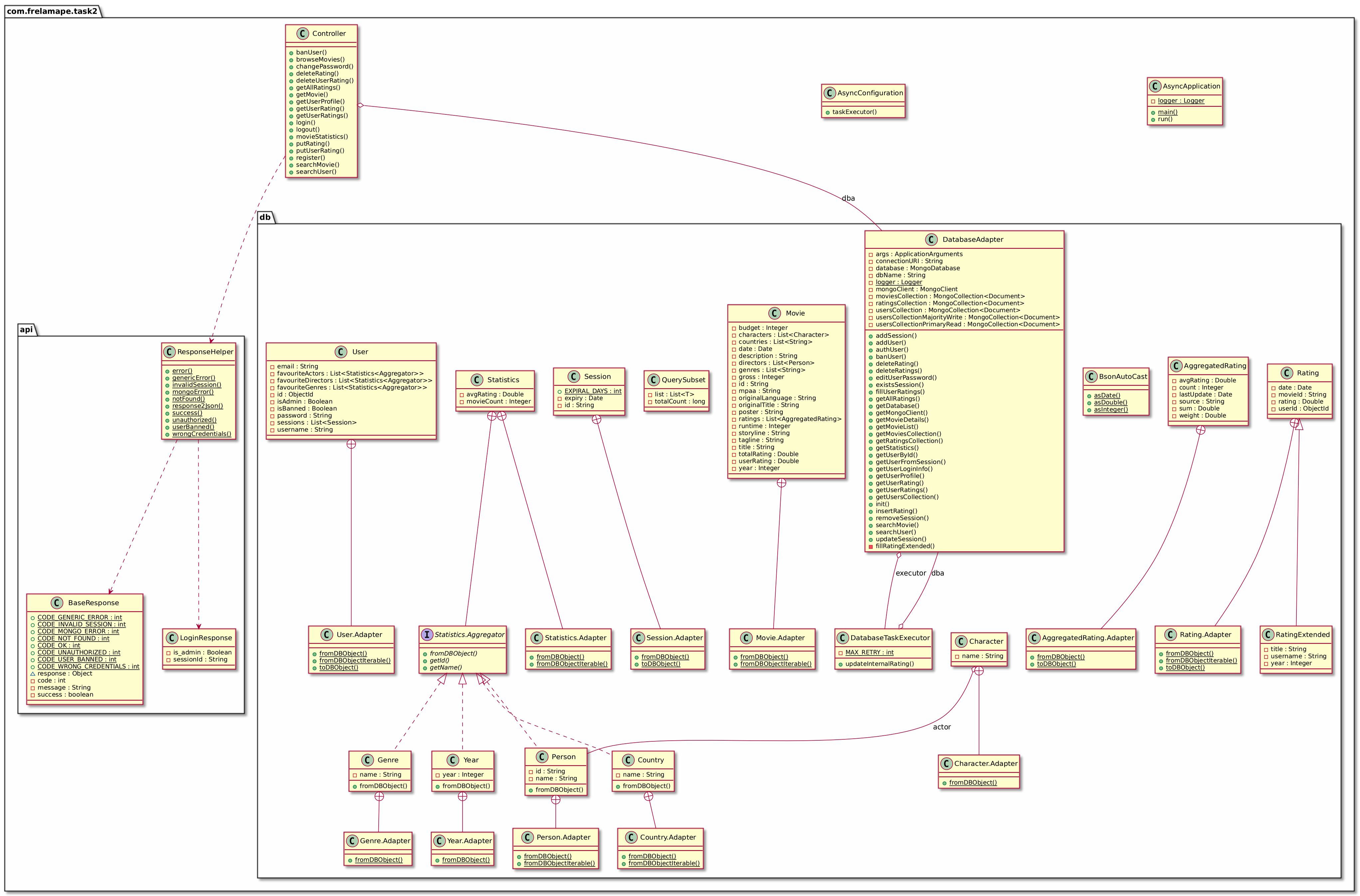
After having build the update dictionary, the scraper builds the list of operations to perform to the movie. In particular, old ratings are pulled and replaced with new ratings, then the *total_rating* is computed and updated along with other information (e.g. *synopsis*, *poster*, ...). In order to interact with Mongo database we exploited several methods from PyMongo module, such as: `find()`, `find_one_and_update()`, `updateOne()`, `bulk_write()` All these methods take dictionaries as argument, with the exception of *bulk_write*, that is used for efficiency's sake.

```python
def UpdateMongoMovies(self,coll_name,nrows=20):
    mysource_p = ms.web_sources.get("mymovies",None)
    if mysource_p == None :
        raise Exception("Sorry, requested source not available")

    # coll_iterator = self.getCollection(coll_name,nrows,skipIdx)####
    coll_iterator = self.getMoviesByLastScraped(nrows)

    for movie in coll_iterator:
        # mark movie as scraped to prevent other scrapers to scrape the same
        # movie
        movie_in_db = self.db['movies'].find_one_and_update({
            '_id': movie['_id']
        }, {
            '$set': {
                'last_scraped': datetime.now()
            }
        })

        # check if copy in db changed since last fetch
        if movie_in_db['last_scraped'] > movie['last_scraped']:
            # another scraper took it
            print("\n--already scraped--\n")
```

```python
                continue

        # scrape information from MyMovies -----------------------------------
        # [...] -> upd_dic

        # scrape information from IMDb ---------------------------------------
        # [...] -> upd_dic

        # update movie ratings -----------------------------------------------
        operations = []
        for rating in upd_dic['ratings']:
            operations += [
                UpdateOne(
                    {'_id': movie['_id']},
                    {'$pull': {
                        "ratings": {
                            "source": rating['source']
                        }
                    }}
                ),
                UpdateOne(
                    {'_id': movie['_id']},
                    {'$push': {
                        "ratings": rating
                    }}
                ),
            ]

            # merge new ratings with old ones
            # [...]
            # calculate new total_rating
            # [...]
            # update movie document with total rating andother info
            del upd_dic['ratings'] # remove ratings since already updated
            upd_dic['last_scraped'] = datetime.now()
            operations.append(UpdateOne({
                    '_id': movie['_id']
                }, {
                    '$set': upd_dic
            }))

            if operations:
                self.db['movies'].bulk_write(operations)

            print("\n---movie updated---")
```

4 Back-end Implementation

The API specification can be found in the `docs/api.md` file. This section will go through the implementation of the main APIs.


com.frelamape.task2
Controller
banUser()
browseMovies()
changePassword()
deleteRating()
deleteUserRating()
getAllRatings()
getMovie()
getUserProfile()
getUserRating()
getUserRatings()
login()
logout()
movieStatistics()
putRating()
putUserRating()
register()
searchMovie()
searchUser()
AsyncConfiguration
taskExecutor()
AsyncApplication
logger : Logger
main()
run()
dba
api
ResponseHelper
error()
genericError()
invalidSession()
mongoError()
notFound()
response2Json()
success()
unauthorized()
userBanned()
wrongCredentials()
BaseResponse
CODE_GENERIC_ERROR : int
CODE_INVALID_SESSION : int
CODE_MONGO_ERROR : int
CODE_NOT_FOUND : int
CODE_OK : int
CODE_UNAUTHORIZED : int
CODE_USER_BANNED : int
CODE_WRONG_CREDENTIALS : int
response : Object
code : int
message : String
success : boolean
LoginResponse
is_admin : Boolean
sessionId : String
db
DatabaseAdapter
args : ApplicationArguments
connectionURI : String
database : MongoDatabase
dbName : String
logger : Logger
mongoClient : MongoClient
moviesCollection : MongoCollection<Document>
ratingsCollection : MongoCollection<Document>
usersCollection : MongoCollection<Document>
usersCollectionMajorityWrite : MongoCollection<Document>
usersCollectionPrimaryRead : MongoCollection<Document>
addSession()
addUser()
authUser()
banUser()
deleteRating()
deleteRatings()
editUserPassword()
existsSession()
fillUserRatings()
getAllRatings()
getDatabase()
getMongoClient()
getMovieDetails()
getMovieList()
getMoviesCollection()
getRatingsCollection()
getStatistics()
getUserById()
getUserFromSession()
getUserLoginInfo()
getUserProfile()
getUserRating()
getUserRatings()
getUsersCollection()
init()
insertRating()
removeSession()
searchMovie()
searchUser()
updateSession()
fillRatingExtended()
User
email : String
favouriteActors : List<Statistics<Aggregator>>
favouriteDirectors : List<Statistics<Aggregator>>
favouriteGenres : List<Statistics<Aggregator>>
id : ObjectId
isAdmin : Boolean
isBanned : Boolean
password : String
sessions : List<Session>
username : String
Statistics
avgRating : Double
movieCount : Integer
Session
EXPIRAL_DAYS : int
expiry : Date
id : String
QuerySubset
list : List<T>
totalCount : long
Movie
budget : Integer
characters : List<Character>
countries : List<String>
date : Date
description : String
directors : List<Person>
genres : List<String>
gross : Integer
id : String
mpaa : String
originalLanguage : String
originalTitle : String
poster : String
ratings : List<AggregatedRating>
runtime : Integer
storyline : String
tagline : String
title : String
totalRating : Double
userRating : Double
year : Integer
BsonAutoCast
asDate()
asDouble()
asInteger()
AggregatedRating
avgRating : Double
count : Integer
lastUpdate : Date
source : String
sum : Double
weight : Double
Rating
date : Date
movieId : String
rating : Double
userId : ObjectId
User.Adapter
fromDBObject()
fromDBObjectIterable()
toDBObject()
Statistics.Aggregator
fromDBObject()
getId()
getName()
Statistics.Adapter
fromDBObject()
fromDBObjectIterable()
Session.Adapter
fromDBObject()
toDBObject()
Movie.Adapter
fromDBObject()
fromDBObjectIterable()
executor
dba
DatabaseTaskExecutor
MAX_RETRY : int
updateInternalRating()
Character
name : String
AggregatedRating.Adapter
fromDBObject()
toDBObject()
Rating.Adapter
fromDBObject()
fromDBObjectIterable()
toDBObject()
RatingExtended
title : String
username : String
year : Integer
actor
Genre
name : String
fromDBObject()
Year
year : Integer
fromDBObject()
Person
id : String
name : String
fromDBObject()
Country
name : String
fromDBObject()
Character.Adapter
fromDBObject()
Genre.Adapter
fromDBObject()
Year.Adapter
fromDBObject()
Person.Adapter
fromDBObject()
fromDBObjectIterable()
Country.Adapter
fromDBObject()
fromDBObjectIterable()

4.1 Java classes overview

In the previous page you can see the class diagram of the Java implementation. The code is structured in 3 modules:

- the root module contains the Spring REST controller and other Spring-related classes;
- the `api` module contains the helper classes for formatting the output to the client as a JSON format
- the `db` module contains:
  - the definition of the Java database objects (*User*, *Movie*, *Rating*, ...). In addition, each object has an utility *Adapter* nested class that provides static methods to easily convert a BSON document to the class instance.
  - the *DatabaseAdapter* class that contains the definition of all possible operation to the database.
  - the *DatabaseTaskExecutor* class that provides the definition of the asynchronous tasks that operate on the database (they are explained more in detail in the following sections).
  - other utility classes (*BsonAutoCast*, *QuerySubset*, ...).

4.2 Authentication

This subsection gives an overview over the most important APIs for authentication and session management. Passwords are stored in the database in the form of a Pbkdf2 hash (128-bit, 10k iterations) which is computed using the *Spring Security* package.

Register In order to register a user, it must first be checked that no user with same username and password exists (snippet 1) and then, if no duplicate entry is found, the user can be added to the database (snippet 2). If creation was successful, a new session for the new user is then added (snippet 4). Any subsequent API call must be accompanied by the session identifier in order to identify the user (snippet 5).

Change Password This query (snippet 3) is simply done by matching the user by username and setting the new password hash.

Login In order to authenticate the user, the user document is retrieved from the database and then the password hash is checked with the password.

Logout The session is removed from the DB (snippet 6).

```
// 1 - check user uniqueness
usersCollection.find(
        or(
            eq("username", u.getUsername()),
            eq("email", u.getEmail())
        )
).first();

// 2 - insert user
```

```
usersCollection.insertOne(User.Adapter.toDBObject(u));

// 3 - edit password
usersCollection.updateOne(
    eq("username", u.getUsername()),
    set("password", u.getPassword())
);

// 4 - add session
usersCollection.updateOne(
    eq("_id", u.getId()),
    push("sessions", Session.Adapter.toDBObject(s))
);

// 5 - finds the user whose session is s
usersCollection.find(
        and(
            eq("sessions._id", s.getId())
        ))
    .first();

// 6 - remove session
usersCollection.updateOne(
    eq("_id", u.getId()),
    pull("sessions", eq("_id", s.getId()))
);
```

4.3 Movies

4.3.1 Browse

Returns a list of filtered movies with the given sorting. The result is paged.

1. build a list of filters based on user specifications. People names are matched if all words in the query string are sub-strings of the full name (this is done using a regex).
2. Make the filter to pass to the *find* function by "and-ing" all conditions.
3. Make the query:

   find all movies matching the conditions

   sort by the user-defined sort order

   project only movie important details

   skip the first *n* pages

   limit the results to the page size

If user is logged in, a separate query will be done to fetch the user's ratings of the returned movies (4.4.2).

```
// 1 - define filters
List<Bson> conditions = new ArrayList<>();
if (minRating != -1)
    conditions.add(gte("total_rating", minRating));

```

```java
if (maxRating != -1)
    conditions.add(lte("total_rating", maxRating));

if (director != null && !director.isEmpty()){
    List<Bson> directorConditions = new ArrayList<>();
    for (String s:director.split(" ")){
        directorConditions.add(regex("directors.name", s, "i"));
    }
    conditions.add(and(directorConditions.toArray(new Bson[]{})));
}

if (actor  != null && !actor.isEmpty()){
    List<Bson> actorConditions = new ArrayList<>();
    for (String s:actor.split(" ")){
        actorConditions.add(regex("characters.actor_name", s, "i"));
    }
    conditions.add(and(actorConditions.toArray(new Bson[]{})));
}

if (country != null && !country.isEmpty())
    conditions.add(eq("countries", country));

if (fromYear != -1){
    conditions.add(gte("year", fromYear));
}

if (toYear != -1)
    conditions.add(lte("year", toYear));

if (genre != null && !genre.isEmpty())
    conditions.add(eq("genres", genre));

// 2 - and conditions
Bson filters;

if (!conditions.isEmpty())
    filters = and(conditions.toArray(new Bson[]{}));
else
    filters = new BsonDocument();

// 3 - make the query
FindIterable<Document> movieIterable = moviesCollection
    .find(filters)
    .sort(sorting)
    .projection(include("title", "year", "poster", "genres", "total_rating", "
    description"))
    .skip(n*(page-1))
    .limit(n);
```

4.3.2 Search

During a search, movies that (fuzzy) match the entire query are returned sorted by "likelihood" (meta text score, in MongoDB). Results are paged as before.

If user is logged in, a separate query will be done to fetch the user's ratings of the returned movies (refer to 4.4.2).

```java
FindIterable<Document> movieIterable = moviesCollection
    .find(text("\""+ query + "\""))
    .projection(Projections.metaTextScore("score"))
    .sort(Sorts.metaTextScore("score"))
    .skip(n*(page-1))
    .limit(n);
```

4.3.3 Get Details

Given a movie id, return all movie details. Code snippet not reported for brevity's sake.

4.3.4 Browse Statistics

This function is used to return aggregated statistics on movies based on one of the following information attributes:

- genre
- year
- country
- director
- character

Before performing the aggregation, movies can be filtered as shown in the *GetMovieList* API. The statistics returned show the following informations about the aggregation attribute:

- aggregated attribute
- average rating calculated on the aggregation
- number of movies present in the aggregation (calculated after filtering)

Results can also be sorted in ascending or descending order based on each of the previous attributes.

The aggregation pipeline is performed with the following code:

```java
AggregateIterable<Document> iterable = moviesCollection.aggregate(
        Arrays.asList(
                Aggregates.match(filters),
                Aggregates.unwind("$" + realUnwindBy),
                Aggregates.group(
                        "$" + realGroupBy,
                        Accumulators.first("name", "$" + realGroupName),
                        Accumulators.avg("avg_rating", "$total_rating"),
                        Accumulators.sum("movie_count", 1)
                ),
                Aggregates.sort(sorting),
                Aggregates.skip(n*(page-1)),
                Aggregates.limit(n)
        )
);
```

The **match** function uses the array of conditions "*filters*" to perform all the filters in a single step. **Unwind** de-constructs the array field and returns a document for each array element (this is necessary for country, director and character fields). Then, the **group** function groups documents by the value expressed in the "*realGroupBy*" and then applies accumulator expression to each group:

- **Accumulators.first** sets "*realGroupName*" as name of the group
- **Accumulators.avg** calculates the average rating of all the "*total_rating*" and saves them in the "*avg_rating*" attribute
- **Accumulators.sum** calculates the number of movies in the aggregation, by adding 1 to the variable "*movie_count*" for each movie

After that, the results are sorted with the **sort** function, where the argument "*sorting*" is calculated as follow:

```java
Bson sorting;
if (sortOrder == 1) {
    sorting = ascending(realSortBy);
} else if (sortOrder == -1) {
    sorting = descending(realSortBy);
} else {
    throw new RuntimeException("sortOrder must be 1 or -1.");
}
```

Here, "*realSortBy*" is the value to sort.

Finally, the **skip** and the **limit** functions are used to manage the display of the results.

4.4 Ratings

4.4.1 CRUD operations

A rating can be added, read, updated or deleted. Related code snippets are not shown for brevity's sake.

After a rating is updated, the corresponding movie rating information are asynchronously updated (refer to 4.4.5).

4.4.2 Get all ratings for a user and list of movies

This particular operation is done to add the user's ratings to a query result set. First the list of movie ids of the movies in the query result is built and then all ratings of the given user to any of those movies is returned.

By building the array of identifiers, just one query to the database is necessary.

NB: the number of movies will always be small due to paging (20).

```java
// 1 - build list of ids of the movies I'm interested in
List<String> ids = new ArrayList<>();
for (Movie m:movies){
    ids.add(m.getId());
}

// 2 - fetch corresponding ratings from database
```

```java
FindIterable<Document> ratingIterable = ratingsCollection.find(
    and(
        eq("_id.user_id", u.getId()),
        in("_id.movie_id", ids)
    )
);
```

4.4.3 Browse All

Administrators can browse all ratings in the collection in descending date order. Results are paged as seen in previous sections. Related code snippet is not shown for brevity's sake.

4.4.4 Browse User Ratings

Users can browse all their ratings in descending date order. Results are paged as seen in previous sections. Related code snippet is not shown for brevity's sake.

4.4.5 Update Internal Rating

After a rating is changed, the average internal rating and the total rating of the movie should be updated. This is done asynchronously after the update in order not to make the client wait for this additional operation.

The update takes in input the old and the new rating (one of them can be null in case of insertion or deletion) and consists in three parts:

1. fetch the movie from the DB
2. build the update request
    (a) Update *internal rating*. The *internal rating* nested document may be missing, in which case we need to push it. Otherwise, addition, deletion or modification must be correctly handled by incrementing `sum` and `count` appropriately.
    (b) Update the *total rating* by calculating the new average. If no rating is present any longer, the *total rating* is unset.

NB: the internal rating update is protected from concurrent modifications since it uses the atomic update operation `$inc` to update the sum and the count. Furthermore, note also that all other aggregate ratings do not have the sum attribute but directly the average attribute since they are scraped. On the other hand, the update of the total rating is not consistent since another asynchronous entity (e.g. scraper) may overwrite it with stale information. However, this is not a big issue since at maximum just a few ratings will be lost among a big pool (thousands and more) so the average does not change much.

```java
// 1 - Fetch movie from DB
Movie m = dba.getMovieDetails(movieId);

// 2- Build the update request
Bson match;
List<Bson> updates = new ArrayList<>();

// 2a - update internal rating
```

```java

if (internalRating != null) { // there are previous ratings
    // match the related nested document
    match = and(
        eq("_id", movieId),
        elemMatch("ratings", new Document() // always present
            .append("source", "internal"))
    );

    if (oldRating == null) { // adding
        updates.add(inc("ratings.$.sum", newRating.getRating()));
        updates.add(inc("ratings.$.count", 1));
    } else if (newRating == null) { // deleting
        updates.add(inc("ratings.$.sum", -oldRating.getRating()));
        updates.add(inc("ratings.$.count", -1));
    } else { // changing
        updates.add(inc("ratings.$.sum", newRating.getRating() - oldRating.
    getRating()));
        updates.add(set("ratings.$.last_update", new Date()));
    }
    // [...] update internalRating
} else { // internal rating is missing and must be created
    match = eq("_id", movieId);
    // [...] create new internalRating
    updates.add(
        push("ratings", AggregatedRating.Adapter.toDBObject(internalRating))
    );
}

// 2b - update total rating
// [...] calculate total rating
if (ar_count > 0) {
    updates.add(set("total_rating", m.getTotalRating()));
} else { // if no rating source found
    updates.add(unset("total_rating"));
}

// 3 - execute query on DB
dba.getMoviesCollection().updateOne(match, combine(updates));
```

4.5 Users

4.5.1 Get Profile

The user profile is composed by his personal information and his rating statistics. The first part is just a *find* operation. The second part uses an aggregation pipeline similar to the one used to find movie statistics with the difference that in the end results are sorted by descending rating and only first 3 results are shown to the user.

Note that this aggregation pipeline is, in a real scenario, very fast since a user will not have more than a few hundreds of ratings and ratings are indexed also by `user_id` (so the match stage is extremely fast).

```java
// 1 - fetch User from DB
User u = User.Adapter.fromDBObject(
        usersCollection.find(
                eq("username", username)
        ).first()
);

// 2 - calculate statistics

for (String[] field:new String[][]{actors, directors, genres}) {
    // for each aggregation (actors, directors, genres) th pipeline is the
    // same so I just set the specific fields and reuse the same code in a
    // loop
    AggregateIterable<Document> iterable = ratingsCollection
        .aggregate(Arrays.asList(
            Aggregates.match(eq("_id.user_id", u.getId())),
            // [...] lookup, project, unwind, unwind, group
            // sort by rating
            Aggregates.sort(descending("avg_rating", "movie_count")),
            // get three most liked
            Aggregates.limit(3)
    ));

    // 3 - add result to user profile
    if (field == actors){
        u.setFavouriteActors(
            Statistics.Adapter.fromDBObjectIterable(iterable, Person.class));
    } else if (field == directors){
        u.setFavouriteDirectors(
            Statistics.Adapter.fromDBObjectIterable(iterable, Person.class));
    } else if (field == genres){
        u.setFavouriteGenres(
            Statistics.Adapter.fromDBObjectIterable(iterable, Genre.class));
    }
}
```

4.5.2 Ban

When a user is banned, a flag is set on its document and all his ratings are removed from the database (triggering the internal rating update). Code snippet is not reported for brevity's sake.

4.5.3 Search

User search works alike the movie search, taking advantage of a *text index*, sorting results by best match and paging the result. Code snippet is not reported for brevity's sake.

5 Authentication and replication

5.1 Configuration of the replica set

Replication has been achieved through the set up of a MongoDB replica set in a remote cluster of 3 virtual machines provided by the University of Pisa.

A configuration document has been written, specifying the id of the replica set and the addresses of the machines. Additionally, a location tag was included. This could be useful if, in the future, the application is deployed to multiple data-centers for a more fine-grained control over the database requests (e.g. writes could be applied to at least one machine in every data-center before returning).

```
rsconf = {
    _id: "replicaset0",
    members: [
    {_id: 0, host: "172.16.1.251:27017", tags: { location: 'unipi'} },
    {_id: 0, host: "172.16.1.162:27017", tags: { location: 'unipi'} },
    {_id: 0, host: "172.16.1.195:27017", tags: { location: 'unipi'} }
    ]
}
```

By default MongoDB binds only to localhost, to allow connection through the network the `mongod` daemon was started with the `--bind_ip` flag.

```
mongod  --replSet replicaset0
        --dbpath ~/data/rs_db
        --oplogSize 200
        --bind_ip 172.16.1.162
```

5.2 Authentication

In order to keep our system secure, authentication must be set-up. A user responsible of handling our database, *moviedb*, was created from the mongo shell.

```
db.createUser(
    {
        user: 'lsmsdb',
        pwd: '******',
        roles: [ { role: "readWrite", db: "moviedb" } ]
    }
)
```

This is not enough since particular operations, like adding or removing new nodes to the replica set, require special privileges. The solution we choose was to create a superuser:

```
db.createUser(
    {
        user: "lsmsdb_admin",
        pwd: "******",
        roles: [ {role: "root", db: 'admin' }],
        passwordDigestor : "server"
    }
)
```

The `passwordDigestor` is an optional parameter which allows us to specify who has to digest the password (client or server). Since we use the SCRAM-SHA-256 mechanism, which is not compatible with the client, the server choice was forced.

After the users have been created, we run the MongoDB daemon with the authentication enabled, adding the `--auth` flag:

```
mongod  --replSet replicaset0
        --dbpath ~/data/rs_db
        --oplogSize 200
        --bind_ip 172.16.1.162
        --auth
```

All the connections from our Spring Boot server will now have to be authenticated with the *lsmsdb* user we just created, while the administration of the cluster or of its node is to be done with *lsmsdb_admin*.

5.3 Consistency, Availability and Partition tolerance

The CAP theorem states that only two between consistency, availability and partition tolerance could be achieved in a database: our main service, providing movies infos and ratings, could work well even with outdated data. Movie's description or title rarely change while average ratings should have a low variability: it is therefore the most convenient choice to increase our availability and partition tolerance. This is not true for what concerns logins and accounts in general: e.g a password change should be set in stone or the user creation should grant login consistency. A bad user experience is the consequence of not handling correctly this aspects.

Different handling of the write and read requests depending on the data we want could be a reasonable solution. The drivers connecting to the database can specify a *WriteConcern* and a *ReadPreference*: the first determines the number of members of the replica set that should have written the new data before a positive response is sent to the driver, the second option is about the possibility of reading from nodes other than the master. The default configuration of MongoDB is a WriteConcern equal to 1 (only the primary server is updated) and a ReadPreference of 'primary' (we can read only from the primary). This means that, in case of a network partition excluding a number of nodes less than the majority and including the master, thus with the election of a new primary, consistency is not guaranteed since a write could have been made just before the partitioning (and the read would work just fine from the newly elected master).

What has been chosen, as already anticipated, is a differentiated approach depending on the request. To access the `users` collection, the driver establishes three different connections with the database.

```
mongoClient = MongoClients.create(connectionURI);
database = mongoClient.getDatabase(dbName);

moviesCollection = database.getCollection("movies").withReadPreference(
    ReadPreference.nearest());

usersCollection = database.getCollection("users").withReadPreference(ReadPreference
    .nearest());

usersCollectionMajorityWrite = database.getCollection("users").withWriteConcern(
    WriteConcern.MAJORITY);

usersCollectionPrimaryRead = database.getCollection("users").withReadPreference(
    ReadPreference.primary());

```

```
ratingsCollection = database.getCollection("ratings").withReadPreference(
    ReadPreference.nearest());
```

`usersCollection` will be used for all the requests where we want high availability, like showing up the profile page or finding the user from the admin control page.

`usersCollectionMajorityWrite` handles all the requests for which we want the updates to be permanent (e.g. password changes, new session id etc.): a write concern of ’majority’ (which is computed against the total nodes in the replica set) protects us against network partition since no matter what happens to the nodes, as long as a majority is reached, the election system will make sure the successive primary node will have our write. Since the secondary nodes could or could not have it, when we want consistency we have to read only from the primary. Therefore, the `usersCollectionPrimaryRead` will set the correct ReadPreference and will be used accordingly.

For what concerns the other requests, consistency is not strictly necessary and we would like our system to be highly available. For this reason, we keep the default write concern (equal to 1) and set the read preference to ’nearest’, thus allowing reads from secondary nodes (which could have outdated informations).

5.4 Test and evaluation

5.4.1 Election

A test of the election mechanism has been done and, as we can see from the log reported below, it worked flawlessly: since no primary was seen for 10 seconds, a new election started (or, to be precise, a dry run followed by a *real* run).

```
2020-05-16T19:55:36.242+0100 I  ELECTION [replexec-0] Starting an election, since
    we’ve seen no PRIMARY in the past 10000ms
2020-05-16T19:55:36.242+0100 I  ELECTION [replexec-0] conducting a dry run election
     to see if we could be elected. current term: 41
2020-05-16T19:55:36.242+0100 I  REPL     [replexec-0] Scheduling remote command
    request for vote request: RemoteCommand 305 -- target:172.16.1.162:27017 db:
    admin cmd:{ replSetRequestVotes: 1, setName: "replicase
t0", dryRun: true, term: 41, candidateIndex: 0, configVersion: 6, lastCommittedOp:
    { ts: Timestamp(1589655321, 1), t: 41 } }
2020-05-16T19:55:36.242+0100 I  REPL     [replexec-0] Scheduling remote command
    request for vote request: RemoteCommand 306 -- target:172.16.1.195:27017 db:
    admin cmd:{ replSetRequestVotes: 1, setName: "replicase
t0", dryRun: true, term: 41, candidateIndex: 0, configVersion: 6, lastCommittedOp:
    { ts: Timestamp(1589655321, 1), t: 41 } }
2020-05-16T19:55:36.242+0100 I  ELECTION [replexec-5] VoteRequester(term 41 dry run
    ) failed to receive response from 172.16.1.162:27017: HostUnreachable: Error
    connecting to 172.16.1.162:27017 :: caused by :: C$nnection refused
2020-05-16T19:55:36.244+0100 I  ELECTION [replexec-2] VoteRequester(term 41 dry run
    ) received a yes vote from 172.16.1.195:27017; response message: { term: 41,
    voteGranted: true, reason: "", ok: 1.0, $clusterTi$e: { clusterTime: Timestamp
    (1589655321, 1), signature: { hash: BinData(0,
    0000000000000000000000000000000000000000), keyId: 0 } }, operationTime:
    Timestamp(1589655321, 1) }
2020-05-16T19:55:36.244+0100 I  ELECTION [replexec-2] dry election run succeeded,
    running for election in term 42
2020-05-16T19:55:36.250+0100 I  REPL     [replexec-2] Scheduling remote command
    request for vote request: RemoteCommand 307 -- target:172.16.1.162:27017 db:
```

```
    admin cmd:{ replSetRequestVotes: 1, setName: "replicas$t0", dryRun: false, term:
     42, candidateIndex: 0, configVersion: 6, lastCommittedOp: { ts: Timestamp
    (1589655321, 1), t: 41 } }
11 2020-05-16T19:55:36.251+0100 I  REPL     [replexec-2] Scheduling remote command
    request for vote request: RemoteCommand 308 -- target:172.16.1.195:27017 db:
    admin cmd:{ replSetRequestVotes: 1, setName: "replicas$t0", dryRun: false, term:
     42, candidateIndex: 0, configVersion: 6, lastCommittedOp: { ts: Timestamp
    (1589655321, 1), t: 41 } }
12 2020-05-16T19:55:36.251+0100 I  ELECTION [replexec-5] VoteRequester(term 42) failed
     to receive response from 172.16.1.162:27017: HostUnreachable: Error connecting
    to 172.16.1.162:27017 :: caused by :: Connectio$ refused
13 2020-05-16T19:55:36.265+0100 I  ELECTION [replexec-3] VoteRequester(term 42)
    received a yes vote from 172.16.1.195:27017; response message: { term: 42,
    voteGranted: true, reason: "", ok: 1.0, $clusterTime: { cl$sterTime: Timestamp
    (1589655321, 1), signature: { hash: BinData(0,
    0000000000000000000000000000000000000000), keyId: 0 } }, operationTime:
    Timestamp(1589655321, 1) }
14 2020-05-16T19:55:36.265+0100 I  ELECTION [replexec-3] election succeeded, assuming
    primary role in term 42
15 2020-05-16T19:55:36.265+0100 I  REPL     [replexec-3] transition to PRIMARY from
    SECONDARY
16 2020-05-16T19:55:36.265+0100 I  REPL     [replexec-3] Resetting sync source to
    empty, which was :27017
17 2020-05-16T19:55:36.266+0100 I  REPL     [replexec-3] Entering primary catch-up
    mode.
18 2020-05-16T19:55:36.268+0100 I  REPL_HB  [replexec-5] Heartbeat to
    172.16.1.162:27017 failed after 2 retries, response status: HostUnreachable:
    Error connecting to 172.16.1.162:27017 :: caused by :: Connection $efused
19 2020-05-16T19:55:36.268+0100 I  REPL     [replexec-5] Caught up to the latest
    optime known via heartbeats after becoming primary. Target optime: { ts:
    Timestamp(1589655321, 1), t: 41 }. My Last Applied: { ts: T$mestamp(1589655321,
    1), t: 41 }
20 2020-05-16T19:55:36.268+0100 I  REPL     [replexec-5] Exited primary catch-up mode.
21 2020-05-16T19:55:36.268+0100 I  REPL     [replexec-5] Stopping replication producer
22 2020-05-16T19:55:36.268+0100 I  REPL     [ReplBatcher] Oplog buffer has been
    drained in term 42
23 2020-05-16T19:55:36.268+0100 I  REPL     [ReplBatcher] Oplog buffer has been
    drained in term 42
24 2020-05-16T19:55:36.268+0100 I  REPL     [RstlKillOpThread] Starting to kill user
    operations
25 2020-05-16T19:55:36.269+0100 I  REPL     [RstlKillOpThread] Stopped killing user
    operations
26 2020-05-16T19:55:36.269+0100 I  REPL     [RstlKillOpThread] State transition ops
    metrics: { lastStateTransition: "stepUp", userOpsKilled: 0, userOpsRunning: 0 }
27 2020-05-16T19:55:36.269+0100 I  REPL     [rsSync-0] transition to primary complete;
     database writes are now permitted
```

The candidate node was `172.16.1.251` and received a positive vote from `172.16.1.195`, while `172.16.1.162` was unreachable (since it was down). A majority (2 out of 3) was reached and the secondary node transitioned to primary. It then caught up with the latest optime (row 19), cleaned the oplog and killed all the current user operations (row 25).

From the time of the dry run for election (19:55:36 and 242ms), which is 10 seconds after the last heartbeat in our case, to the new primary being up and running (19:55:36 and 269ms) only 27ms were elapsed: this is possible since the three nodes are connected to

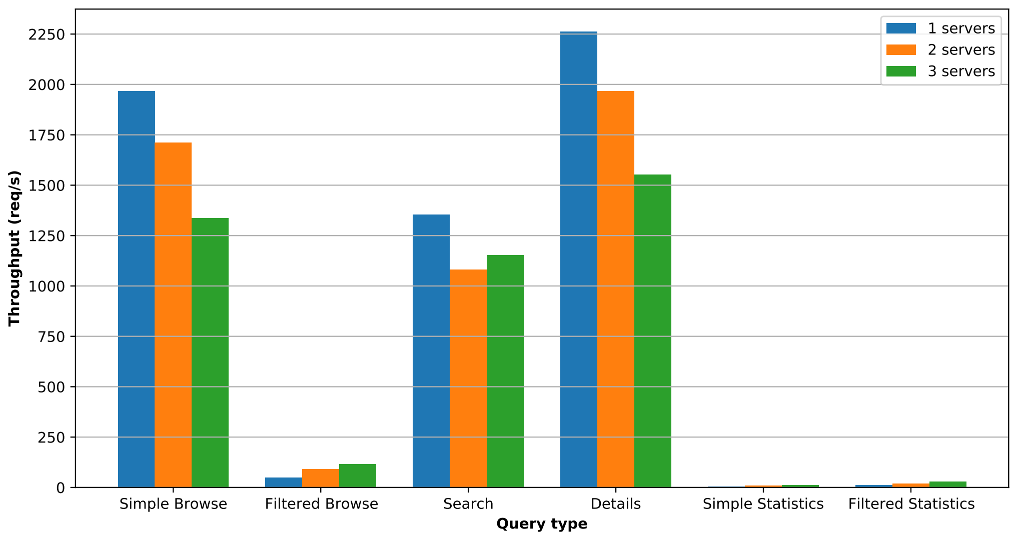


Figure 3: Results of the performance test for all 6 request types and for

the same LAN, but certifies how the election mechanism is fast and efficient.

5.4.2 Performance

Another test we made is the evaluation of the performance, in terms of throughput (requests that can be served per second on peak load), of the database. We chose to test only the read performance of the most common API calls since these will be by far the most common. In particular, we individuated six kinds of API calls:

- "Simple browse": movie browse without filters (10 movies per request).
- "Filtered browse": movie browse with filters (10 movies per request)
- "Search": movie search (10 movies per request)
- "Details": movie details
- "Simple Statistics": movie statistics without filters (10 groups per request)
- "Filtered Statistics": movie statistics with filters (10 groups per request)

For each type of API call, we generated a random subset of requests from which the testing application will randomly choose a request. The tests have been repeated three times, with a different number of servers in the replica set (1, 2 or 3). We used *siege* as a testing application, with 100 concurrent users and no delay between requests (benchmark mode), which executed for a minute for each run. Both back-end server serving the APIs and the *siege* application ran on a server that was part of the replica set (this is the only member of the replica set that was always on during the experiments). All replica members are hosted on one of the virtual machines provided by the University of Pisa, each having 2 virtual CPU cores and a variable amount of RAM (from 2GB to 8GB, in the one hosting the Java back-end). The database contains around 50k movies.

Results From figures 3 and 4 we can note three things:

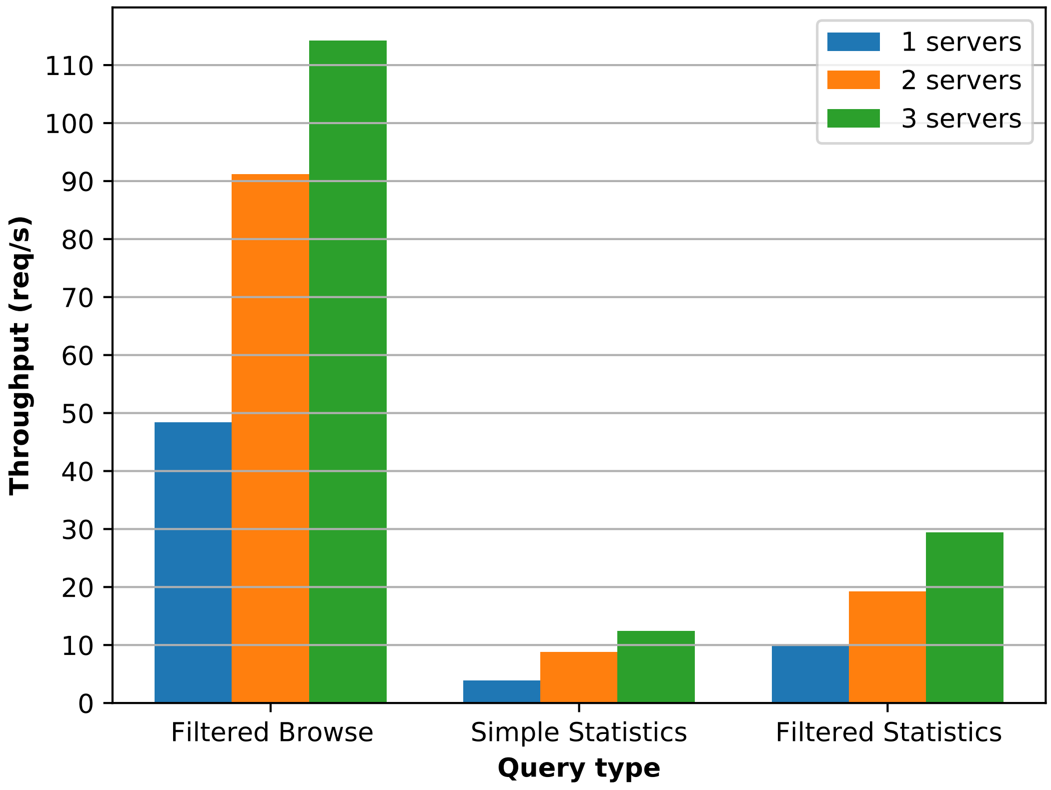


Figure 4: Same plot as figure 3 but zoomed in to show slowest API calls.

1. “Simple Browse”, “Search” and “Details” are much faster than the other queries. This was expected since these queries require little to no effort to the database.

2. For the fast queries listed above, the performance actually lowers with an increased number of servers in the replica set. This is a consequence of the set-up we ran experiments on. The Java back-end server ran on the same node as the MongoDB node that was kept alive. Since these queries are IO-bound (in contrast, the other slower ones are computationally bound), when the Mongo driver, configured with the *nearest* read preference, notices that the node is slow, it tries to balance read requests among the other nodes but, since they are remote, this actually introduces overheads.

3. Statistics are faster on a filtered query then on a simple one. This is somehow counter-intuitive but keep in mind that the filter phase is very fast since it operates on indices and it reduces the number of elements to aggregate by possibly a large amount. Since the reduce phase needs to run on fewer records, its execution time is reduced.

Possible solutions Since statistics are a core part of the Movienator website, a way to make them run faster must be found. The solution could be caching the most common statistics pages using an in-memory-database (e.g. Redis or Memcached). For instance, we could save the first few pages (e.g. 3 pages of 10 elements) for each possible aggregation (4) and visualization order (3). This would require space for just 360 records but would provide a huge impact on the server throughput. With regards to the problem of the fast queries being slow on a bigger cluster, a possible solution could be deploying a Java back-end server in each MongoDB node and set-up a load-balancer (e.g. HAProxy) to balance HTTP requests among the different back-ends. By doing so, latencies are minimized by locating the application near to the data source.

5.4.3 Availability

Some tests have been performed to test the ability of our MongoDB cluster to resist the failure of some nodes. Therefore some randomly chosen nodes have been shut down but the application kept responding to requests as expected.

A more interesting test is the observation of the performance degradation (in terms of average response time) over time when nodes are removed from the replica set (fig. 5). In the following experiment, we ran 100 concurrent users making random back-to-back requests on the whole public section of Movienator. Every minute, a secondary node is either removed from the replica set or re-added, following the following sequence of alive node count: 3, 2, 1, 2, 3.

Results From the figure we can make two observations:

1. there is no substantial difference in the average response time with 3 and 2 nodes in the replica set. In contrast, the 95-percentile increases noticeably. This is explained by the fact that, as seen in the previous section, it’s long running queries that are hurt harder by the missing server.

2. when a node is restored, going from 1 to 2 nodes in the cluster, the users experience a transient spike of the response time before stabilising. A similar effect is not experienced when a node is removed from the set or when going from 2 to 3 nodes. This

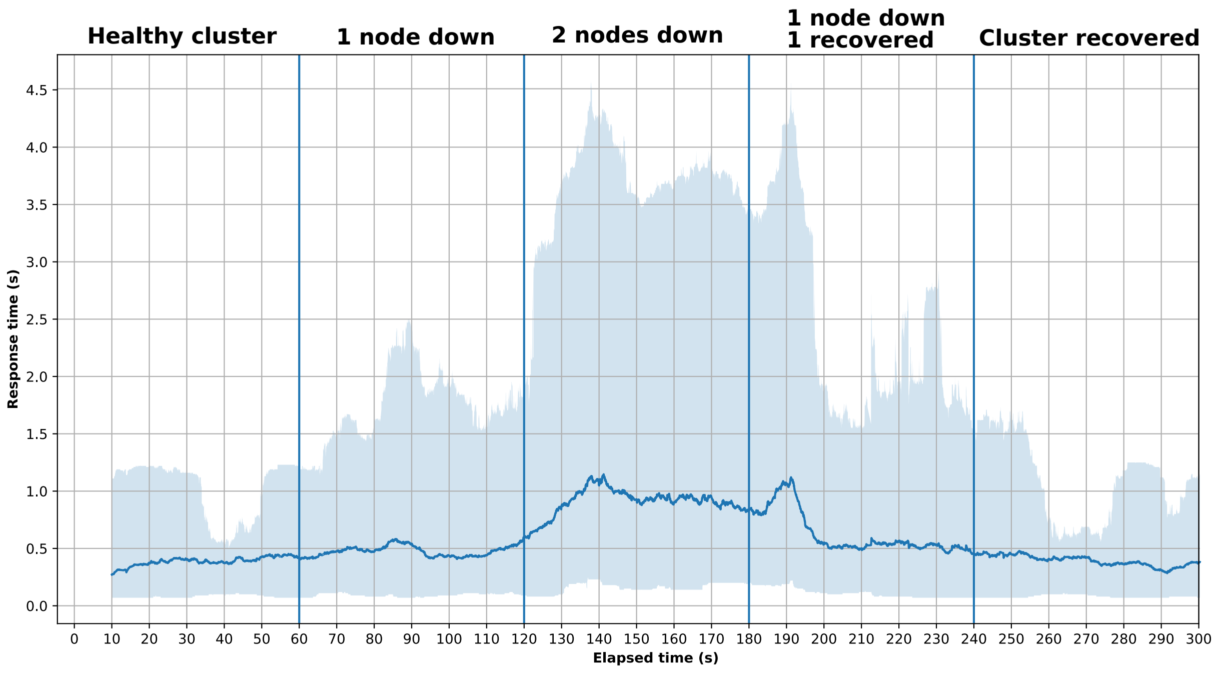


Figure 5: Plot of the moving average of the response time over time (line) and the 95-percentile to 5-percentile gap (area) inside the window. The window size of the moving average is 10s, therefore changes to the replica set are seen with a 10s delay. At t=60,120,180,240 the number of nodes in the replica set changes (3, 2, 1, 2, 3).

can be explained by the fact that when a node is bring up, it has to communicate with the primary to recover its state. This increased burden on the primary temporarily slows down requests. This does not happen when removing the nodes since, in this experiment, only secondary nodes have been removed.

5.5 Sharding

Let's now discuss the possible introduction of sharding in our MongoDB. The discussion will be differentiated by collection.

Movies While the size of the *movies* collection is quite large, it has a very low variability and it is not expected to grow fast over time. Furthermore, its contents easily fit into the memory of a single server. For these reasons, the sharding of the movie collection is not suggested.

Users Also the *users* collection is not very large but, differently to the *movies* collection, it grows faster and is composed of many small documents. If the number of users is small (up to 100000 order of magnitude), there is no need to shard them but, if this number grows to millions or even more, the collection could be sharded without problems since operations on users are a really small portion of all operations in Movienator and the additional overhead will not be a problem. The partitioning could be performed on the hash of the *_id* if the shards are co-located in the same geographical location or could be based on the TLD of the user's email in case of a geographically distributed database.

Ratings The *ratings* collection is the fastest growing collection in the database. It is also possibly the biggest one and the only write-heavy one. Furthermore, users only access their own ratings, with the exception of admins. This makes it possible to flawlessly introduce sharding based on the *user_id* of the user. By doing so, all ratings of a user will be co-located on the same node. The administrators may experience a higher latency for getting the most recent ratings but it's not important. Of course, there should be a significant amount of ratings to justify the introduction of sharding.