Università di Pisa

Large-Scale and Multi-structured Databases - Project: Rotten Movies

Fabio Piras, Giacomo Volpi, Guillaume Quint

Contents

Chapter 1

Introduction

Rotten Movies is a web service where user can keep track of the general liking for movies, they can also share their thoughts with the world after signing up. In addition user can follow renowned top critic to be constantly updated with their latest review.

Guest user, as well as all other, can browse or search movies based on filters like: title, year of release and actor who worked in it. They can also view a Hall of Fame for the most positive review genres, production houses and years in which the best movie were released.

For this project, the application has been developed in Java with the Spring framework and as Thymeleaf as engine to implement a web GUI. The application use a document database to store the main information about the movies, user, review and actor; a graph database is instead used to keep track of the relationship between normal user and top critic in terms of who follows who and between the movie and the user who reviewed it.

Chapter 2

Feasibility Study

The very first step was to perform is to look up for what type of data we need to create the application and how to handle it by performing a feasibility study.

1 Dataset analysis and creation

Initially we searched online for available dataset, we ended up with five different files coming from Kaggle and imdb with a combined storage of 4.18 GB:

- title_principal.tsv
- name_basic.tsv
- title_basic.tsv
- title_crew.tsv
- rotten_movies.csv
- rotten_reviews.csv

The first three were found on the imdb site containing general data about the title of the entries, type (not all were movies), cast, crew and other useful information. The last two came from Kaggle and they contain data scraped from the rotten tomatoes site regarding the movies rating and their reviews. All of these dataset were organized in a relational manner.

Initial steps and creation of the movie collection After a general look it was clear that we needed to process the data to obtain a less bloated dataset by deciding what to keep in base of our needs and specification; we decided to use python as programming language to trim the original file, this was also achieved through the use of Google Colab.

We started by taking all the tsv file and transforming them into a single file, we performed a join operation on all of them based on their id and then discarded the useless information. For the code see (Appendix A .1). After that we performed the same operation on the csv file coming from Rotten Tomatoes (Appendix A .2). We then proceed to join the two files based on the title of the movies (Appendix A .3).

Unfortunately we noticed that after the join there were more rows for the same movie, in fact, due to the relational nature of the first dataset, we had a different entry for personnel on each row, so we rollback to the start, but this time we collapsed all the information in a single row (Appendix A .4).

Due to the variability of the data in the string fields, like the content of the reviews, we decided to perform an escape on various elements to avoid crashes and failure during the import into the DBSM (Appendix B .1). We also had to do a similar process of normalization for the date fields (Appendix B .2)

Finally we achieved our goal of having a json file for the movie collection, the file occupied a space of 270MB.

Creation of the user collection The following step was to generate a collection for the user, this was achieved by starting from the movie one and for each different review author we generated a document later to be placed in a single json file of which the total storage size was 86,6 MB, this step was performed directly on the mongo shell (Appendix B .3).

2 Analysis result

With the dataset for the document database ready we finally had a better understanding on how to shape the models and the relative functionality in the application code. We will discuss later of the design of the collections and the methods used to interact with them. The same goes for the graph database of whom the structure was heavily influenced by the one in the MongoDB

Chapter 3

Design

1 Main actors

As already mentioned in the introduction we have mainly three major actors: guest user and register user, the latter can be divided between normal user and top critic. In addition there is an admin actor who's main role is to oversee the entire service.

2 Functional requirements

This section describes the functional requirements that need to be provided by the application in regards of the actor:

- Guest (Unregistered) User can:
  - login/register into the service
  - search movies by search bar and other filters
  - view movies, their details and relative reviews
  - view the personal page of the author of a selected *top critic* review
  - view the different halls of fame
- Normal use can:
  - logout from the service
  - search movies by search bar and other filters
  - view movies, their details and relative reviews
  - write a review for a selected film
  - view the personal page of the author of a selected *top critic* review
  - view the different halls of fame
  - follow/unfollow a *top critic* user
  - view the feed of latest reviews from the followed *top critic*
  - view a suggestion feed for top critic to follow
  - view the history of its own reviews
  - modify its own reviews
  - delete its own reviews

  - change its account information
- Top Critics can:
  - logout from the service
  - search movies by search bar and other filters
  - view movies, their details and relative reviews
  - write a top critic review for a selected film
  - view the different halls of fame
  - view the history of its own top critics reviews
  - modify its own top critics reviews
  - delete its own top critics reviews
  - change its account information
  - see the number of its followers
- Admin user can:
  - logout from the service
  - search movies by search bar and other filters
  - view movies, their details and relative reviews
  - view the different halls of fame
  - modify films details
  - add/remove films
  - browse *user* and *top critic*
  - ban *user* and *top critic*
  - register new *top critic*
  - perform analytic on the user population

3 Non functional requirements

In the following section are listed the non functional requirements for the application.

- the system must encrypt users password
- the service must be built with OOP language
- user must have 16 or more years to register into the service
- service must be implemented through a responsive website
- avoid single point of failure in data storage
- high availability, accepting data displayed temporarily in an older version

4 Implementation regarding the CAP theorem

We will now discuss on how we decided to tackle the CAP theorem issue. In our minds the application is a read-heavy one where we expect that the number of read transaction are by far more numerous than the write operation, so we decided that our application would prioritize high availability of and low latency capable of withstanding network partitions. In reference of figure 3.1 it is clear that we moved towards a AP approach in spite of data consistency.

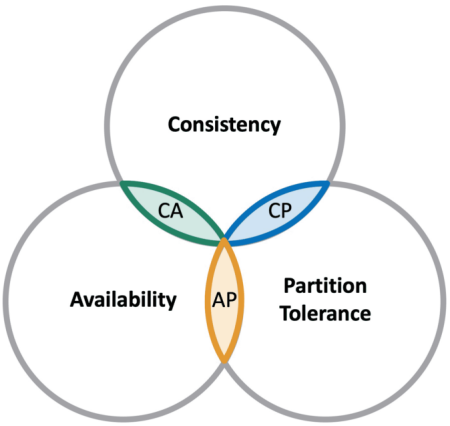
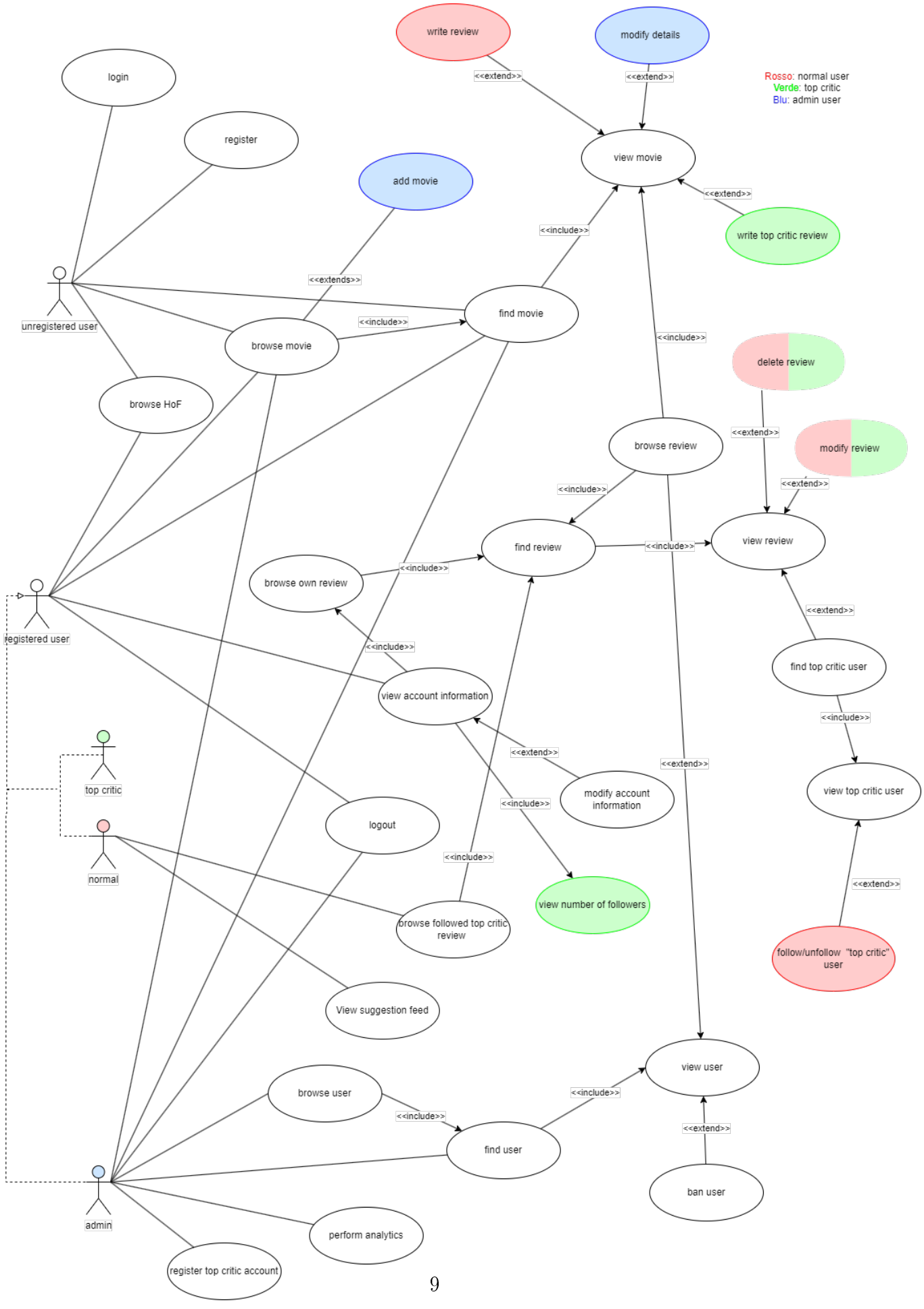


Figure 3.1: CAP theorem diagram

In order to guarantee the requirements of high availability, we decided to accept the cases in witch the data shown to the user could be not updated to the latest version in the database.

5 Use cases


write review
modify details
login
register
add movie
view movie
Rosso: normal user
Verde: top critic
Blu: admin user
<<extend>>
<<extend>>
<<extend>>
write top critic review
<<include>>
<<extends>>
unregistered user
find movie
<<include>>
browse movie
<<include>>
delete review
browse HoF
browse review
<<extend>>
modify review
<<include>>
<<extend>>
find review
<<include>>
view review
browse own review
<<include>>
registered user
<<include>>
<<extend>>
find top critic user
view account information
<<include>>
top critic
<<extend>>
<<extend>>
modify account information
view top critic user
logout
<<include>>
normal
<<include>>
view number of followers
<<extend>>
browse followed top critic review
follow/unfollow "top critic" user
View suggestion feed
view user
browse user
<<include>>
<<include>>
<<extend>>
find user
ban user
admin
perform analytics
register top critic account

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

6 Class analysis

In this section we shall discuss of the design of the various class and how are they related

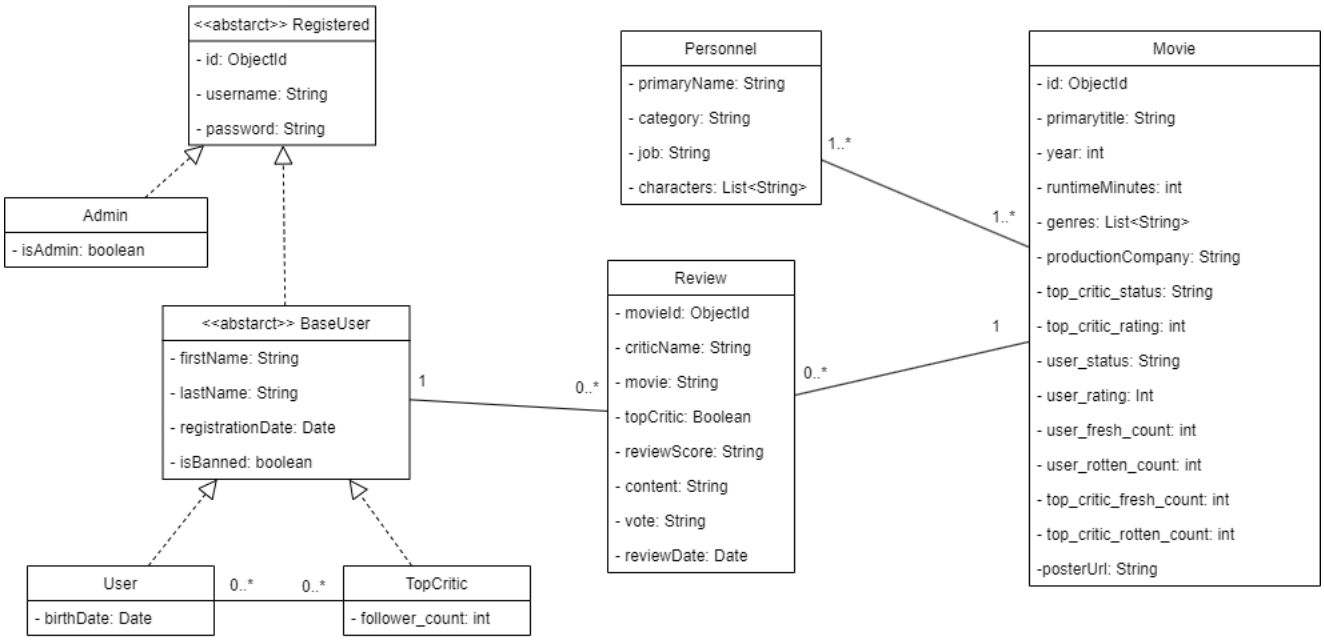


Figure 3.3: Class Diagram

The diagram express the following relationship between the entities.

- a BaseUser can write from zero to many reviews
- a Review can be written by a single BaseUser for a single Movie
- a Movie can have zero to many reviews and can have from 1 to many Personnel working in it
- a Personnel can work in 1 to many Movie

Chapter 4

Document database

In this chapter we will discuss of the organization of the document database and how we handled the replicas. We decided to use MongoDB as DBMS for the document database for the purpose of storing the main information for movies, users, reviews and personnel. In MongoDB we created the following three collection:

- movie
- user

Inside the movie collection there are embedded the documents for the reviews and personnel

1 Collection composition

1 .1 Movie collection

The following is the composition of the Movie collection.

```
{
    "_id" : ObjectId(<<id_field>>),
    "primaryTitle": "The first ever movie",
    "year": 1989,
    "runtimeMinutes": 70,
    "genres": ["Crime", "Drama", "Romantic"],
    "productionCompany": "Dingo Picture Production" ,
    "personnel": [
        {
            "name": "John Doe",
            "category": "producer",
            "job": "writer"
        },
        {
            "name": "Christopher Lee",
            "category": "actor",
            "character": ["The one"]
        },
        ...
    ],
    "top_critic_fresh_count": "4",
    "top_critic_rotten_count": 0,
    "user_fresh_count": 14,
```

```
    "user_rotten_count": 1,
    "top_critic_rating": 100,
    "top_critic_status": "Fresh",
    "user_rating": 93,
    "user_status": "Fresh",
    "review":
    [
        {
            "critic_name":    "AntonyE",
            "review_date": "2018-12-07",
            "review_type": "Rotten",
            "top_critic":  false,
            "review_content":"I really didn't liked it!",
            "review_score":   "1/10"
        },
        {
            "critic_name":    "AntonyE",
            "review_date": "2015-12-07",
            "review_type": "Fresh",
            "top_critic":  true,
            "review_content":"I really liked it!",
            "review_score":   "A-"
        },
        ...
    ]
}
```

Listing 4.1: Test

As previously stated the field personnel and review are arrays of embedded documents.

1 .2 User collection

The following is the composition of the User collection.

```
{
    "_id"               : ObjectId(<<id_field>>),
    "username"          :"AntonyE",
    "password"          :"hashed_password",
    "firstName"         :"Anton",
    "lastName  "        :"Ego",
    "registrationreview_date"  :"2019-06-29",
    "review_date of birth"     :"2002-07-16",
    "last3Reviews":
        [
            {
                "_id"               : ObjectId(<<id_field>>),
                "primaryTitle"    : "Star Wars: A new Hope",
                "review_date"          : "2018-12-07",
                "review_type"     : "fresh",
                "top_critic"   : false,
                "review_content"       : "I really liked it!",
                "vote"          : "8/10"
```

```
19              },
20              {
21                  "_id"                : ObjectId(<<id_field>>),
22                  "primaryTitle"     : "Ratatouille",
23                  "review_date"           : "2019-02-17",
24                  "review_type"      : "rotten",
25                  "top_critic"    : false,
26                  "review_content"        : "The director was also
    ↪ controlled by a rat",
27                  "vote"           : "D+"
28              },
29              {
30                  "_id"                : ObjectId(<<id_field>>),
31                  "primaryTitle"     : "300",
32                  "review_date"           : "2020-02-15",
33                  "review_type"      : "fresh",
34                  "top_critic"    : false,
35                  "review_content"        : "Too much slow motion",
36                  "vote"           : "3.5/5"
37              }
38          ],
39      "reviews":[
40          {
41              "movie_id" : ObjectId(<<id_field>>),
42              "primaryTitle" : "Evidence",
43              "review_index": 11
44          },
45          ...
46      ]
47
48 }
```

Listing 4.2: Test

Is important to notice that in this collection we have two fields for the reviews, *last3Reviews* and *reviews* which present different ways of storing data for the same underling entity. As suggested by the name itself *last3Reviews* contains the last three review in the form of an embedded document meanwhile *reviews* contains all the reviews made by a user in the form of document linking towards the movie for which they were written; in particular, the link is formed by the id of the movie, the title and the position in which the review can be found in the array *review* of the movie document. This structure was chosen so that we could avoid a full replica of the reviews in both collections and, at the same time, avoiding to perform join operation for those reviews that are more frequently checked, which are the most recent ones. The idea of creating a separated collection for reviews was immediately discarded because it would have resorted in a design for the document database that resemble a third normal form of relational database.

2 Indexes

In order to provide the best execution speed in search queries we use indexes. In particular we focus on the application part that is available also without registering to the site, like Hall of Fame and search movies functionalities. Without indexes we need a collection scan in user and

movie collections to find the right document.

2 .1 Movie collection

- primaryTitle
- year
- genres
- top_critic _rating
- user_rating

```
1 // Dimension is expressed in Kb
2 {
3   _id_: 228,
4   primaryTitle_1: 292,
5   year_1: 92,
6   genres_1: 180,
7   'personnel.primaryName_1': 1680,
8   top_critic_rating_1: 80,
9   user_rating_1: 80
10 }
```

Listing 4.3: Test

2 .2 User collection

- username
- date_of _birth

```
1 // Dimension is expressed in Kb
2
3 { _id_: 180, username_1: 168, date_of_birth_1: 100 }
```

Listing 4.4: Test

We decided to use these indexes because they provide a jump in term of speed in search queries without occupying much space. We accept the fact that writes are slower because our application is read-intesive. We consider the idea of using an index on personnel.primaryName but the cost in term of space was higher than potential benefit. For a full analityc report for the indexes you can see the appendix. (Appendix C)

3 Partition and replicas

To ensure high-availability we deploy a cluster of replica (3). We install and configure MongoDB in all machines, with these priorities

1. 172.16.5.26 (primary)
2. 172.16.5.27 (secondary)
3. 172.16.5.28 (secondary)

We decided to use **nearest read preference** and **W2 write preference**. In fact we can tolerate that users see temporarily an old version of data with a 33% chanche.

4 Aggregations

In this section we shall discuss on the different aggregation that the application will implement, the values between «» represent a values passed by an above level.

4 .1 Return the best years by top critic and user ratings

Mongo shell

```
db.movie.aggregate([
    {$group:
        {
        _id: "$year",
        topCritic:{$avg:"$top_critic_rating"},
        rate:{$avg:"$user_rating"},
        count:{$sum:1}
        }
    },
    {$match:{count:{$gte:<<i>>}}},
    {$sort:{topCritic:-1, rate:-1}},
    {$limit: <<j>>}
])
```

Listing 4.5: Test

Java implementation

```
AggregateIterable<Document> aggregateResult = collection.aggregate(
                    Arrays.asList(
                            Aggregates.group("$year",
                                    avg("top_critic_rating", "$top_critic_rating"
    ),
                                    avg("user_rating", "$user_rating"),
                                    sum("count",1)),
                            Aggregates.match(gte("count",numberOfMovies)),
                            Aggregates.sort(opt.getBsonAggregationSort()),
                            Aggregates.limit(Constants.
    HALL_OF_FAME_ELEMENT_NUMBERS)
                    )
        );
```

Listing 4.6: Test

4 .2 Return the best genres by top critic and user ratings

Mongo shell

```
db.movie.aggregate([
    {
        $unwind:"$genres"
    },
    {$group:
        {
            _id: "$genres",
            topCritic:{$avg:"$top_critic_rating"},
            rate:{$avg:"$user_rating"},
```

```
            count:{$sum:1}
        }
    },
    {$match:{count:{$gte:<<i>>}}},
    {$sort:{topCritic:-1, rate:-1}},
    {$limit:<<j>>}
])
```

Listing 4.7: Test

Java implementation

```
AggregateIterable<Document> aggregateResult = collection.aggregate(
                Arrays.asList(
                        Aggregates.unwind("$genres"),
                        Aggregates.group("$genres",
                                avg("top_critic_rating", "$top_critic_rating"
    ),
                                avg("user_rating", "$user_rating"),
                                sum("count",1)),
                        Aggregates.match(gte("count",numberOfMovies)),
                        Aggregates.sort(opt.getBsonAggregationSort()),
                        Aggregates.limit(Constants.
    HALL_OF_FAME_ELEMENT_NUMBERS)
                )
        );
```

Listing 4.8: Test

4 .3 Return the best production houses by top critic and user ratings

Mongo shell

```
db.movie.aggregate([
    {$group:
        {
            _id: "$production_company",
            topCritic:{$avg:"$top_critic_rating"},
            rate:{$avg:"$user_rating"},
            count:{$sum:1}
        }
    },
    {$match:{count:{$gte:<<i>>}}},
    {$sort:{topCritic:-1, rate:-1}},
    {$limit:<<j>>}
])
```

Listing 4.9: Test

Java implementation

```
AggregateIterable<Document> aggregateResult = collection.aggregate(
                Arrays.asList(
                        Aggregates.group("$production_company",
                                avg("top_critic_rating", "$top_critic_rating"
    ),
```

```
                                    avg("user_rating", "$user_rating"),
                                    sum("count",1)),
                            Aggregates.match(gte("count",numberOfMovies)),
                            Aggregates.sort(opt.getBsonAggregationSort()),
                            Aggregates.limit(Constants.
    HALL_OF_FAME_ELEMENT_NUMBERS)
                    )
        );
```

Listing 4.10: Test

4 .4 Given a movie count the review it has received by each month

Mongo shell

```
db.movie.aggregate([
    {
        $match:{id:<<"id">>}
    },
    {$unwind:"$review"},
    {
        $group:
        {
            _id:{year:{$year:"$review.review_date"}, month:{$month:"$review.
    review_date"}},
            count:{$sum:1}
        }
    },
    {$sort:{_id:1}}

])
```

Listing 4.11: Test

Java implementation

```
Document yearDoc = new Document("year",new Document("$year","$review.
    review_date"));
        Document monthDoc = new Document("month",new Document("$month","
    $review.review_date"));
        ArrayList<Document> test=new ArrayList<>();
        test.add(yearDoc);
        test.add(monthDoc);
        AggregateIterable<Document> aggregateResult = collection.aggregate(
                Arrays.asList(
                        Aggregates.match(eq("_id",id)),
                        Aggregates.unwind("$review"),
                        Aggregates.group(test,
                                sum("count",1)),
                        Aggregates.sort(Sorts.ascending("_id"))
                )
        );
```

Listing 4.12: Test

4 .5 Given a user count the review he/she has made divided by genres

Mongo shell

```
db.movie.aggregate([
    {$match:
        {"review.critic_name":{$eq:<<"name">>}}
    },
    {$unwind:"$genres"},
    {$group:{_id:"$genres", count:{$sum:1}}},
    {$sort:{count:-1}},
    {$limit: <<n>>}
])
```

Listing 4.13: Test

Java implementation

```
AggregateIterable<Document> aggregateResult = collectionMovie.aggregate(
                Arrays.asList(
                        Aggregates.match(eq("review.critic_name",username)),
                        Aggregates.unwind("$genres"),
                        Aggregates.group("$genres",
                                sum("count",1)),
                        Aggregates.sort(Sorts.descending("count")),
                        Aggregates.limit(Constants.
    HALL_OF_FAME_ELEMENT_NUMBERS)
                )
        );
```

Listing 4.14: Test

4 .6 Return the number of user divided by an age bucket

Mongo shell

```
db.user.aggregate([
    {
        $match:{"date_of_birth":{$exists:true}}
    },

    {
        $bucket:
        {
            groupBy: {$year:"$date_of_birth"},
            boundaries: [<<values>>],
            output:
            {
                "population": {$sum:1}
            }
        }
    }
])
```

Listing 4.15: Test

Java implementation

```
1  BucketOptions opt = new BucketOptions();
2          ArrayList<Integer> buck=new ArrayList<>();
3          opt.output(new BsonField("population",new Document("$sum",1)));
4          int bucketYear=1970;
5          buck.add(bucketYear);
6          while(bucketYear<=2010){
7              bucketYear=(bucketYear+offset);
8              buck.add(bucketYear);
9          }
10         AggregateIterable<Document> aggregateResult = collectionUser.
       aggregate(
11                 Arrays.asList(
12                         Aggregates.match(exists("date_of_birth")),
13                         Aggregates.bucket(new Document("$year","
       $date_of_birth"),buck,opt)
14                 )
15         );
```

Listing 4.16: Test

Chapter 5

Graph database

The DBSM chosen for the graph database is Neo4j, it was use to manege the social part of the site and also to keep track of the reviews made for the feed and suggestion functionality.

The entities used in the database are:

- User
- TopCritic
- Movie

The nodes themselves do not store a lot of data, we decided to keep the bare minimum by having the following attributes:

- for the User and TopCritic:
  - id
  - name
- for Movie:
  - id
  - title

The relationship present are:

- User -[:FOLLOWS]-> TopCritic
- User -[:REVIEWED]-> Movie
- TopCritic -[:REVIEWED]-> Movie

In particular the *REVIEWED* relationship contains the following information:

- content
- date
- freshness

Below is present a snapshot of the graph database taken from Neo4j.

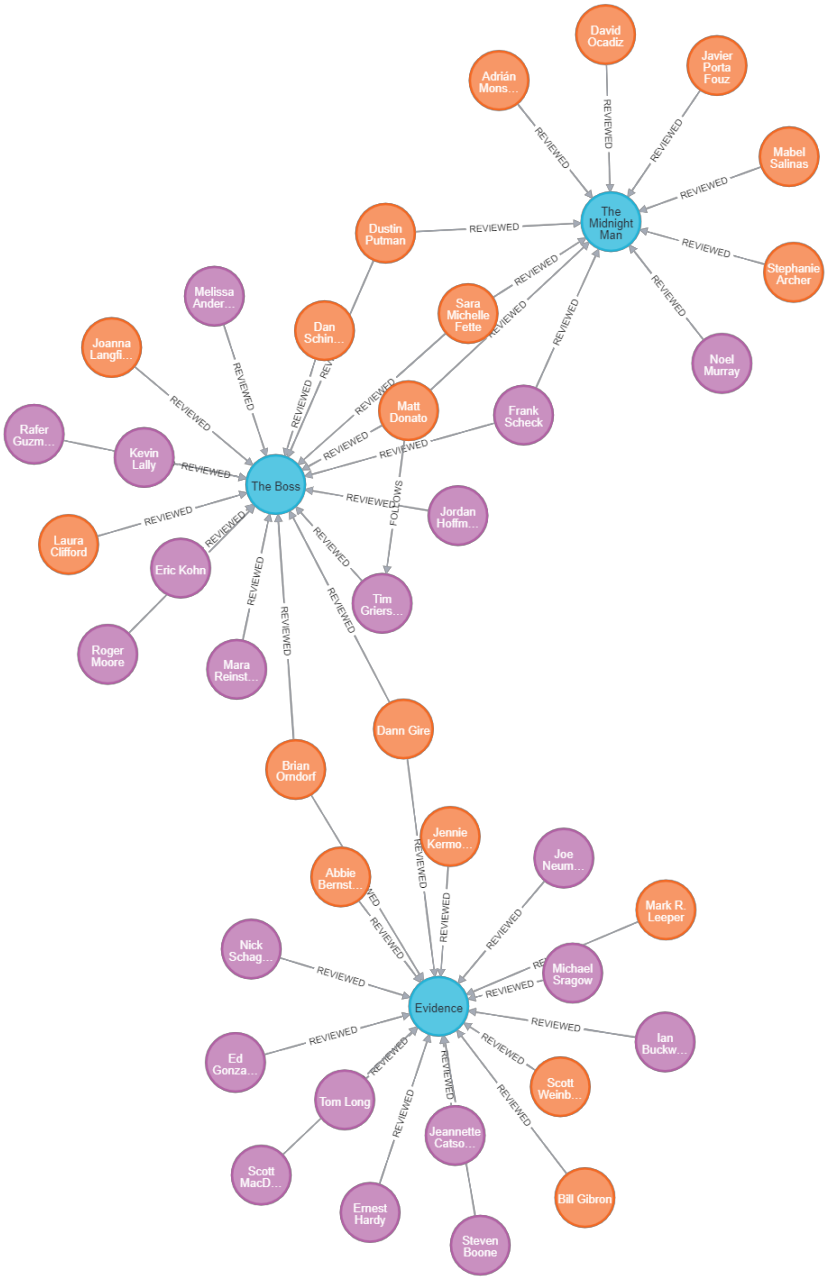


Figure 5.1: graph DB

As previously stated in the Feasibility study section 2 , the graph database design was heavily influenced by the document one, it was in fact generated by python script (Appendix A .6) that starts from the movie collection and, after generating the nodes for the *Movie* entity, does the same for the user by dividing them between normal user and top critic. It then generates the *REVIEWED* relationship based upon the data stored in the document database and finally it generates the *FOLLOWS* relationship randomly

1 Index

The *id* attributes is common to all nodes and it stores a string which is also used as id by the document database in the _ *id* field, this choice was made so that we could have a way to identify the same object across the different databases with only one string from the application. Due to the OOP language used to build the latter the data was manipulated with classes containing the id of the object so, to improved the performance of the graph database we decided to use the *id* attribute as a index.

Below the information in a json format from the Neo4j DBSM about the new index (some attributes were omitted for space)

```
[
  {
    "id": 3,
    "name": "movie_id_index",
    "type": "RANGE",
    "entityType": "NODE",
    "labelsOrTypes": [
      "Movie"
    ],
    "properties": [
      "id"
    ]
  },
  {
    "id": 5,
    "name": "topcritic_id_index",
    "type": "RANGE",
    "entityType": "NODE",
    "labelsOrTypes": [
      "TopCritic"
    ],
    "properties": [
      "id"
    ]
  },
  {
    "id": 4,
    "name": "user_id_index",
    "type": "RANGE",
    "entityType": "NODE",
    "labelsOrTypes": [
      "User"
    ],
    "properties": [
      "id"
    ]
  }
]
```

Listing 5.1: Test

We then proceed to confront the profile of the queries with and without the above indexes, here we can see the result of this analysis:

Figure 5.2: search by id without and with movie_id _index

Figure 5.3: search by id without and with user_id _index

2 Queries

In the following section we will show the queries that drove the design of the graph database.

Graph-Centric Query	Domain-Specific Query
Which are the User nodes with the most outgoing edges of type REVIEWED	Which are the non top critic user with most reviews made
Which are the TopCritic nodes with the most incoming edges of type FOLLOWS	Which are the most followed top critics
Which are the Movies nodes that have been most recently connected with TopCritic nodes by a REVIEWED relationship meanwhile the latter are connected to a User node with a FOLLOWS relationship	Which are the latest reviewed movies by a followed top critics given a starting user
Which are the User and TopCritic nodes that have outgoing REVIEWED edges to the same Movie node without having a FOLLOWS edge between them, which of these TopCritic nodes have more similar attribute on the REVIEWED relationship with the one on the same type going towards the same Movie node given a User node	Which are the non-followed top critics with the most affinity towards a given user
Which Movie nodes have different ratio of freshness in the incoming REVIEWED edges spitted in base of their date attribute	Which movies have been targeted by review bombing in the last x month

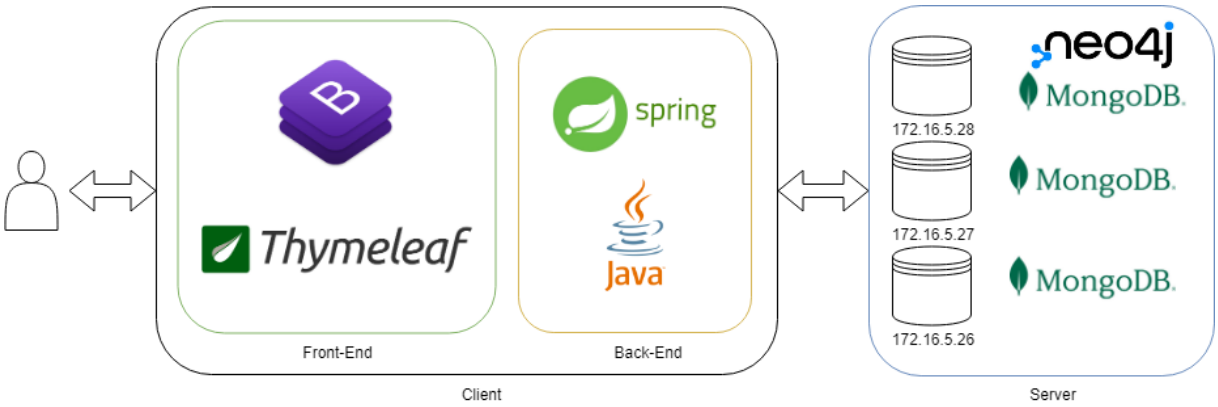
Chapter 6

Software Architecture

Rotten Movies is a web application developed in *Java* that implements the Model-View-Controller (MVC) paradigm through the use of the *Spring Framework*.

Specifically, the *view* layer is handled by the templating engine *Thymeleaf*, which enables the creation of custom web pages by interpolating data provided by the middle-ware with predefined html templates. Interface styling has been eased by the use of some predefined CSS classes defined in *Bootstrap 5.0*[1]

The back-end side of the application is supported by the document database *MongoDB* and the graph database *Neo4J*, which are accessed via the official mongo driver[2] and the neo4j driver[3] for Java, respectively



[1] https://getbootstrap.com/
[2] https://www.mongodb.com/docs/drivers/java/sync/current/
[3] https://neo4j.com/developer/java/

Chapter 7

Application Structure

1 Modules and code organization

The picture below represent the packages in which the application is organized.

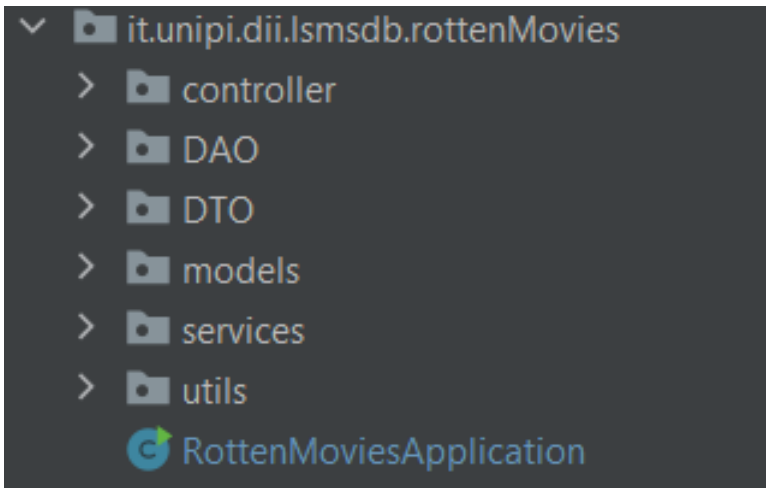


Figure 7.1: module organization

First of all we follow the reverse-domain convention for the root package, before passing to analyzing the source code, we put in the Appendix the pom.xml file for the dependency (Appendix E .1).

- *controller* is responsible for handling the request to the various endpoint with the use of Spring

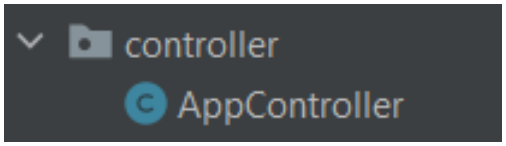


Figure 7.2: controller

- *DAO* (Data Access Object) is responsible for accessing the databases and retrieving the necessary object

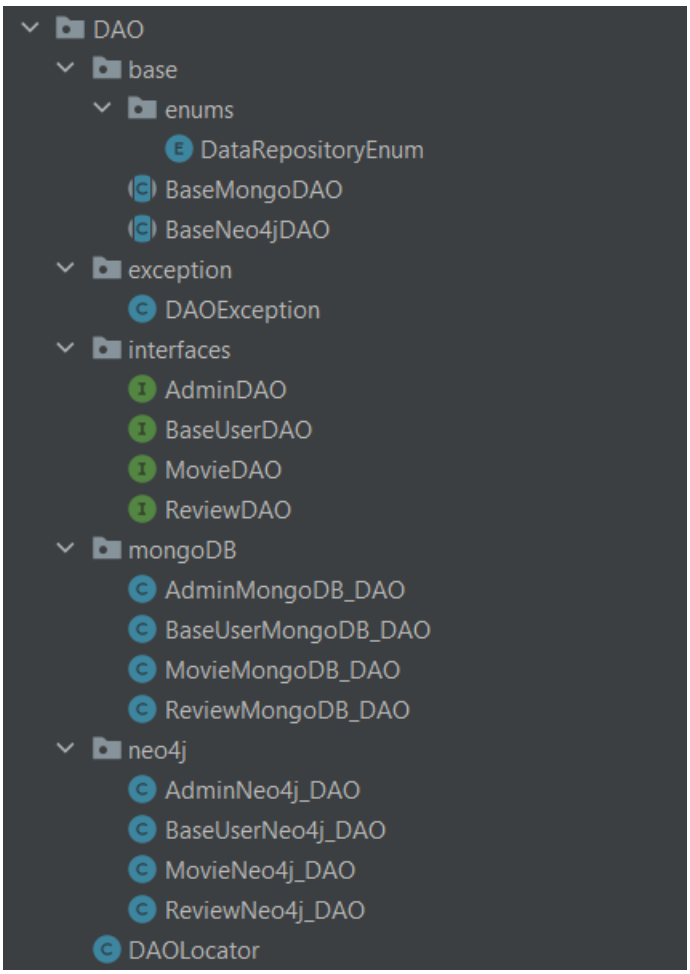


Figure 7.3: DAO

  - *base* contains the classes responsible for handling the base connection the DBs, *enums* is used as packet to differentiate the connection type in base of an enum for higher calls
  - *exception* is responsible for generating and handling a custom exception invoked when trying to access the wrong database
  - *interfaces* contains the various interfaces that map all the method for accessing the databases differentiated in base of the general field for the operation, they extends the *AutoClosable* interface
  - *mongoDB* handles the operation on the MongoDB for the different entities
  - *neo4j* is the same as *mongoDB* but for the Neo4j database

- *DTO* (Data Transfer Object) presents all the classes that are used as container of the data passed between the service layer and the presentation layer

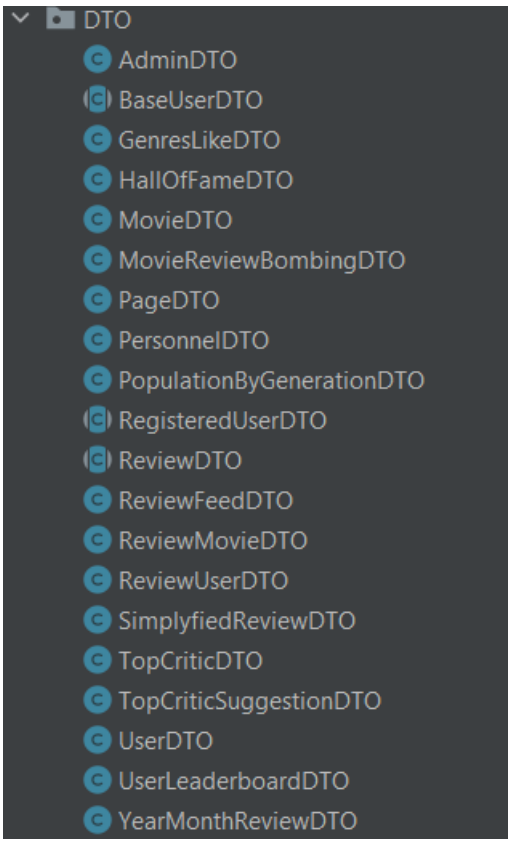


Figure 7.4: DTO

2 Managing consistency between MongoDB and Neo4j

Because we use two databases we need to manage consistency among them. An example on how it is managed is the addMovie method in MovieService. Here first we try to add a movie in MongoDB, if the Mongo operation is successfull we try to add the movie in Neo4j. If Neo4j fails we decided to roll-back the insert on MongoDB, deleting the movie added previously. This strategy is also adopted in add/update/delete operations.

2 .1 Insert movie

```
1  public ObjectId addMovie(String title) {
2          if (title == null || title.isEmpty()) {
3              return null;
4          }
5          Movie newMovie = new Movie();
6          newMovie.setPrimaryTitle(title);
7          ObjectId id = null;
8          try (MovieDAO moviedao = DAOLocator.getMovieDAO(DataRepositoryEnum.
       MONGO)) {
9              id = moviedao.insert(newMovie);
10         } catch (Exception e) {
11             System.err.println(e);
12         }
13         if (id == null) {
14             return null;
15         }
16         newMovie.setId(id);
```

```
17          try (MovieDAO moviedao = DAOLocator.getMovieDAO(DataRepositoryEnum.
    NEO4j)) {
18              id = moviedao.insert(newMovie);
19          } catch (Exception e) {
20              System.err.println(e);
21          }
22          if (id == null){ // roll back di mongo
23              try (MovieDAO moviedao = DAOLocator.getMovieDAO(
    DataRepositoryEnum.MONGO)) {
24                  moviedao.delete(newMovie);
25              } catch (Exception e) {
26                  System.err.println(e);
27              }
28              return null;
29          }
30          return id;
31      }
```

Listing 7.1: Test

Chapter 8

Instruction Manual

A Python Code

A .1 Creation of one single dataset from the tsv imdb file

```
1
2 import pandas as pd
3 from google.colab import drive
4 drive.mount('/content/drive')
5 numberofrows=None #100000
6 title_basics = pd.read_csv("/content/drive/MyDrive/Dataset/Original/
      title_basics.tsv",sep='\t',nrows=numberofrows,header=0)
7 title_principals = pd.read_csv("/content/drive/MyDrive/Dataset/Original/
      title_principals.tsv",nrows=numberofrows,sep='\t',header=0)
8
9 keep_col = ["tconst","titleType","primaryTitle","originalTitle","startYear","
      runtimeMinutes","genres"]
10 title_basics = title_basics[keep_col]
11 title_basics = title_basics[title_basics["titleType"].str.contains("movie")
      == True]
12
13 print(title_basics.head(3))
14
15 merged1=pd.merge(title_basics,title_principals,how='inner',on='tconst')
16 del title_basics,title_principals
17 print(merged1)
18
19 name_basics=pd.read_csv("/content/drive/MyDrive/Dataset/Original/name_basics.
      tsv",sep='\t',nrows=numberofrows,header=0)
20
21 merged2=pd.merge(merged1,name_basics,how='inner',on='nconst')
22 del merged1
23 merged2=merged2.drop(columns=["ordering","nconst","birthYear","deathYear","
      knownForTitles","primaryProfession"])
24 del name_basics
25 print(merged2)
26
27 category=merged2.groupby('tconst')['category'].apply(list).reset_index(name='
      category')
28 job=merged2.groupby('tconst')['job'].apply(list).reset_index(name='job')
29 characters=merged2.groupby('tconst')['characters'].apply(list).reset_index(
      name='characters')
30 primaryName=merged2.groupby('tconst')['primaryName'].apply(list).reset_index(
      name='primaryName')
31 result=merged2.drop_duplicates(subset=['tconst'])
32 result=result.drop(['category'], axis=1).drop(['job'], axis=1).drop(['
      characters'], axis=1).drop(['primaryName'], axis=1)
33 result=result.merge(category,on='tconst').merge(job,on='tconst').merge(
      characters,on='tconst').merge(primaryName,on='tconst')
34 print(result)
35
36 result.to_csv("/content/drive/MyDrive/Dataset/resultSetFinale.csv",index=
      False)
```

Listing 8.1: Test

A .2 Creation of one single dataset from the csv kaggle file

```
1
```

```
import pandas as pd
from google.colab import drive
drive.mount('/content/drive')
numberofrows=None #100000
movies = pd.read_csv("/content/drive/MyDrive/Dataset/Original/rotten_movies.csv",nrows=numberofrows,header=0)
reviews = pd.read_csv("/content/drive/MyDrive/Dataset/Original/rotten_reviews.csv",nrows=numberofrows,header=0)
to_keep = ["rotten_tomatoes_link", "movie_title", "production_company","critics_consensus",
           "tomatometer_status", "tomatometer_rating", "tomatometer_count",
           "audience_status", "audience_rating", "audience_count",
           "tomatometer_top_critics_count", "tomatometer_fresh_critics_count",
           "tomatometer_rotten_critics_count"]

movies = movies[to_keep]
to_drop = ["publisher_name"]
reviews=reviews.drop(columns=to_drop)

merged=pd.merge(movies,reviews,how='inner',on="rotten_tomatoes_link")
print(merged)

categories = {}
arr = ["critic_name", "top_critic", "review_type", "review_score", "review_date", "review_content"]
for x in arr:
  categories[x]=merged.groupby('rotten_tomatoes_link')[x].apply(list).reset_index(name=x)

result=merged.drop_duplicates(subset=['rotten_tomatoes_link'])
for x in arr:
  result=result.drop([x], axis=1)
for x in arr:
  result=result.merge(categories[x],on='rotten_tomatoes_link')

print(result)

result.to_csv("/content/drive/MyDrive/Dataset/resultSetRotten.csv",index=False)
```

Listing 8.2: Test

A .3 Merging of the file generated in the previous script

```

import pandas as pd
from google.colab import drive
drive.mount('/content/drive')
numberofrows=None
imdb = pd.read_csv("/content/drive/MyDrive/Dataset/resultSetFinale.csv",nrows=numberofrows,header=0)
rotten = pd.read_csv("/content/drive/MyDrive/Dataset/resultSetRotten.csv",nrows=numberofrows,header=0)

merged = {}
choose = ['primaryTitle', 'originalTitle']
rowHeadDataset = 20
for x in choose:
```

```
  merged[x]=pd.merge(imdb,rotten,how='inner', left_on=x, right_on='
    movie_title')
  print(x)
  merged[x]=merged[x].drop_duplicates(subset=[x])
  merged[x]=merged[x].drop(columns=['titleType','tomatometer_count','
    tomatometer_top_critics_count'])
  merged[x]=merged[x].rename(columns={'startYear':'year'})
  merged[x]=merged[x].drop(columns=['rotten_tomatoes_link', 'movie_title']+[j
     for j in choose if j!=x])
  print(len(pd.unique(merged[x][x])))
  print(list(merged[x]))
  print("==========================")
  merged[x].to_csv(f"/content/drive/MyDrive/Dataset/ImdbJoinRotten{x}.csv",
    index=False)
  merged[x]=merged[x].head(rowHeadDataset)
  merged[x].to_csv(f"/content/drive/MyDrive/Dataset/headDataset{x}.csv",index
    =False)
```

Listing 8.3: Test

A .4 Collapsing different rows in a single one generating an array for personnel field

```

import pandas as pd
from ast import literal_eval
from google.colab import drive
drive.mount('/content/drive')

numberofrows=None
df = pd.read_csv("/content/drive/MyDrive/Dataset/ImdbJoinRottenprimaryTitle.
    csv",nrows=numberofrows,header=0)

#print([x.split(',') for x in df['genres']])
print(df)

col = ["primaryName","category","job","characters"]
col1 = ["critic_name","top_critic","review_type","review_score","review_date"
    , "review_content"]

df['personnel'] = ""
df['review'] = ""

for row in range(df[col[0]].size):
  it = df['genres'][row]
  df['genres'][row] = ['"' + x + '"' for x in it.split(',')] if it != '\\N'
    else []
  tmp = []
  for c in col:
    tmp.append({c:eval(df[c][row])})
  res = []
  for c in range(len(tmp[0][col[0]])):
    res.append({})
  for i, j in zip(col, tmp):
    for idx, x in enumerate(j[i]):
      #print(i, idx, x)
      if x != '\\N':
        if i == 'characters':
          x = eval(x)
```

```
34          res[idx]["'" + i + "'"] = "'" + str(x).replace("'", "##single-quote##
    ").replace('"', "##double-quote##") + "'"
35      df['personnel'][row] = list(res)
36      #print(res)
37      ###
38      tmp = []
39      for c in col1:
40        to_eval = df[c][row].replace('nan', 'None')
41        arr = eval(to_eval)
42        if c == "review_date":
43          for i, elem in enumerate(arr):
44            arr[i] = elem + "T00:00:00.000+00:00"
45        tmp.append({c:arr})
46        #print(tmp)
47      res = []
48      for c in range(len(tmp[0][col1[0]])):
49        res.append({})
50      for i, j in zip(col1, tmp):
51        for idx, x in enumerate(j[i]):
52          #print(i, idx, x)
53          if x != '\\N':
54            res[idx]["'" + i + "'"] = "'" + str(x).replace("True", "true").
    replace("False", "false").replace("'", "##single-quote##").replace('"', "
    ##double-quote##") + "'"
55      df['review'][row] = list(res)
56      #df['review'][row] = eval(str(res))
57      #print(res)
58      #print()
59
60  df=df.drop(columns=col)
61  df=df.drop(columns=col1)
62  df=df.drop(columns=['tconst'])
63
64  print(df["review"][0])
65
66  it = df['personnel'][0]#[4]['review_content']
67  print(type(it))
68  print(it)
69
70  df.to_csv("/content/drive/MyDrive/Dataset/
        movieCollectionEmbeddedReviewPersonnel.csv",index=False)
71  df = df.head(20)
72  df.to_csv("/content/drive/MyDrive/Dataset/
        headmovieCollectionEmbeddedReviewPersonnel.csv",index=False)
```

Listing 8.4: Test

A .5 Generates a hashed password for all the users

```
1  import hashlib
2  #from pprint import pprint as print
3  from pymongo import MongoClient
4
5  def get_database():
6      CONNECTION_STRING = "mongodb://localhost:27017"
7      client = MongoClient(CONNECTION_STRING)
8      return client['rottenMovies']
9
10 if __name__ == "__main__":
```

```
    dbname = get_database()
    collection = dbname['user']
    total = collection.count_documents({})
    for i, user in enumerate(collection.find()):
        all_reviews = user['last_3_reviews']
        sorted_list = sorted(all_reviews, key=lambda t: t['review_date'])
    [-3:]

        hashed = hashlib.md5(user["username"].encode()).hexdigest()

        newvalues = { "$set": { 'password': hashed, 'last_3_reviews':
    sorted_list } }
        filter = { 'username': user['username']}
        collection.update_one(filter, newvalues)
        print(f"{i/total:%}\r", end='')
    print()
```

Listing 8.5: Test

A .6 Generates the graph database

```
from pymongo import MongoClient
from neo4j import GraphDatabase
from random import randint, shuffle

def get_database():
    CONNECTION_STRING = "mongodb://localhost:27017"
    client = MongoClient(CONNECTION_STRING)
    return client['rottenMovies']

class Neo4jGraph:

    def __init__(self, uri, user, password):
        self.driver = GraphDatabase.driver(uri, auth=(user, password),
    database="rottenmoviesgraphdb")

    def close(self):
        self.driver.close()

    def addUser(self, uid, name, isTop):
        with self.driver.session() as session:
            if isTop:
                result = session.execute_write(self._addTopCritic, uid, name)
            else:
                result = session.execute_write(self._addUser, uid, name)

    def addMovie(self, mid, title):
        with self.driver.session() as session:
            result = session.execute_write(self._addMovie, mid, title)

    def addReview(self, name, mid, freshness, content, date):
        with self.driver.session() as session:
            result = session.execute_write(self._addReview, name, mid,
    freshness, content, date)

    def addFollow(self, uid, cid):
        with self.driver.session() as session:
            result = session.execute_write(self._addFollow, uid, cid)

```

```
37      @staticmethod
38      def _addUser(tx, uid, name):
39          query = "CREATE (n:User{id:\"" + str(uid) + "\", name:\"" + name.
        replace('"', '\\"') + "\"})"
40          #print(query)
41          result = tx.run(query)
42
43      @staticmethod
44      def _addTopCritic(tx, cid, name):
45          query = "CREATE(m:TopCritic{id:\"" + str(cid) + "\", name:\"" + name.
        replace('"', '\\"') + "\"})"
46          #print(query)
47          result = tx.run(query)
48
49      @staticmethod
50      def _addMovie(tx, mid, title):
51          query = "CREATE(o:Movie{id:\"" + str(mid) + "\", title:\"" + title.
        replace('"', '\\"') + "\"})"
52          #print(query)
53          result = tx.run(query)
54
55      @staticmethod
56      def _addReview(tx, name, mid, freshness, content, date): # date in format
        YYYY-mm-dd, freshness in [TRUE, FALSE]
57          query = "MATCH(n{name:\"" + str(name).replace('"', '\\"') + "\"}), (m
        :Movie{id:\"" + str(mid) + "\"}) CREATE (n)-[r:REVIEWED{freshness:" +
        freshness + ", date:date('" + date + "'), content:\"" + content.replace('
        "', '\\"') + "\"}]->(m)"
58          #print(query)
59          result = tx.run(query)
60
61      @staticmethod
62      def _addFollow(tx, uid, cid):
63          query = "MATCH(n:User{id:\"" + str(uid) + "\"}), (m:TopCritic{id:\""
        + str(cid) + "\"}) CREATE (n)-[r:FOLLOWS]->(m)"
64          #print(query)
65          result = tx.run(query)
66
67  if __name__ == "__main__":
68      # dbs initialization
69      dbname = get_database()
70      graphDB = Neo4jGraph("bolt://localhost:7687", "neo4j", "password")
71
72      # user creation
73      collection = dbname['user']
74      total = collection.count_documents({})
75      print(f"user {total = }")
76      for i, user in enumerate(list(collection.find({}, {"_id":1, "username":1,
        "date_of_birth":1}))):
77          graphDB.addUser(user['_id'], user['username'], 'date_of_birth' not in
        user)
78          if not i%100:
79              print(f"{(i+1)/total:%}\r", end='')
80
81      # movie creation and review linking
82      collection = dbname['movie']
83      total = collection.count_documents({})
84      print(f"\nmovie {total = }")
85      for i, movie in enumerate(list(collection.find({}, {"_id":1, "
        primaryTitle":1, "review":1}))):
86          graphDB.addMovie(movie['_id'], movie['primaryTitle'])
```

```
87          movie['review'] = list({v['critic_name']:v for v in movie['review']}.
        values()) # make unique reviews per critic
88          for rev in movie['review']:
89              graphDB.addReview(rev['critic_name'], movie['_id'], {"Fresh":"
        TRUE", "Rotten":"FALSE"}[rev['review_type']], str(rev['review_content'])
        [:15], str(rev['review_date'])[:10])
90          print(f"{(i+1)/total:%}\r", end='')
91
92      # follow linking
93      collection = dbname['user']
94      uids = [x['_id'] for x in list(collection.find({"date_of_birth":{"$exists
        ":True}}, {"_id":1}))]
95      cids = [x['_id'] for x in list(collection.find({"date_of_birth":{"$exists
        ":False}}, {"_id":1}))]
96      total = len(uids)
97      print(f"\nfollow {total = }")
98      for i, user in enumerate(uids):
99          shuffle(cids)
100         for j in range(randint(0, 20)):
101             graphDB.addFollow(user, cids[j])
102         print(f"{i/total:%}\r", end='')
103
104     graphDB.close()
```

Listing 8.6: Test

B Mongosh scripts

B .1 Perform the escape on the string fields

```
1
2 db.movie.find().forEach(
3     x => {
4         print(x.primaryTitle);
5         x.review = JSON.parse(
6             x.review.replaceAll('"\'', '"')
7                 .replaceAll('\'"', '"')
8                 .replaceAll('"false"', 'false')
9                 .replaceAll('"true"', 'true')
10                .replaceAll('"None"', 'null')
11                .replaceAll(/\\x\d{2}/g, "")
12                .replaceAll("##single-quote##", "\'")
13                .replaceAll("##double-quote##", '\\"')
14                .replaceAll("\\x", "x")
15        );
16        x.personnel = JSON.parse(
17            x.personnel.replaceAll('"\'', '"')
18                .replaceAll('\'"', '"')
19                .replaceAll('"None"', 'null')
20                .replaceAll("##single-quote##", '\'')
21                .replaceAll("##double-quote##", '\\"')
22                .replaceAll('"[\'', '["')
23                .replaceAll('"[\\"', '["')
24                .replaceAll('\']"', '"]')
25                .replaceAll('\\"]"', '"]')
26                .replaceAll(/(\[[^[:]*)\\", \\"([^]:]*\])/g, '$1", "$2')
27                .replaceAll(/(\[[^[:]*)\', \\"([^]:]*\])/g, '$1", "$2')
28                .replaceAll(/(\[[^[:]*)\\", \'([^]:]*\])/g, '$1", "$2')
29        );
30        x.genres = JSON.parse(
31            x.genres = x.genres.replaceAll('"\'', '"')
32                    .replaceAll('\'"', '"')
33                    .replaceAll('"None"', 'null')
34                    .replaceAll("##single-quote##", "\'")
35                    .replaceAll("##double-quote##", '\\"')
36        );
37        db.movie.updateOne(
38            {"_id": x._id},
39            {$set:
40                {
41                    "review": x.review,
42                    "personnel": x.personnel,
43                    "genres": x.genres,
44                    "runtimeMinutes":parseInt(x.runtimeMinutes),
45                    "year":parseInt(x.year),
46                    "tomatometer_rating":parseFloat(x.tomatometer_rating),
47                    "audience_rating":parseFloat(x.audience_rating),
48                    "audience_count":parseFloat(x.audience_count),
49                    "tomatometer_fresh_critics_count":parseInt(x.
    tomatometer_fresh_critics_count),
50                    "tomatometer_rotten_critics_count":parseInt(x.
    tomatometer_rotten_critics_count)
51                }
52            }
53        );
```

```
    }
);
```

Listing 8.7: Test

B .2 Normalize the date field in the DB

```
total = db.movie.find().count();
i = 0;
db.movie.find().forEach(
    x => {
        print(x.primaryTitle);
        x.review.forEach(rev =>{
            if(typeof (rev.review_date) === "string" ){
                db.movie.updateOne(
                    {primaryTitle: x.primaryTitle },
                    { $set: { "review.$[elem].review_date" : new Date(rev.review_date) } },
                    { arrayFilters: [ { "elem.critic_name": rev.critic_name } ] }
                )
            }
        })
        print(100*i++/total);
});
```

Listing 8.8: Test

B .3 Create a new collection for the user based on the data present in the movie collection

```
total = db.runCommand({ distinct: "movie", key: "review.critic_name", query: {"review.critic_name":{$ne:null}}}).values.length
i = 0;
db.runCommand(
{ distinct: "movie", key: "review.critic_name", query: {"review.critic_name":{$ne:null}}}).values.forEach(
    (x) => {
        review_arr = []
        movie_arr = []
        is_top = false
        db.movie.aggregate(
            [
                { $project:
                    {
                        index: { $indexOfArray: ["$review.critic_name", x]},
                        primaryTitle: 1
                    }},
                {$match:{index:{$gt:-1}}}
            ]
        ).forEach(
            y => {
                tmp = db.movie.aggregate([
                    {
                        $project:
                        {
                            top_critic: {
```

```
                                    $arrayElemAt: ["$review.top_critic", y.index]
                                },
                                primaryTitle: y.primaryTitle,
                                review_type: {
                                    $arrayElemAt: ["$review.review_type", y.index
    ]
                                },
                                review_score: {
                                    $arrayElemAt: ["$review.review_score", y.
    index]
                                },
                                review_date: {
                                    $arrayElemAt: ["$review.review_date", y.index
    ]
                                },
                                review_content: {
                                    $arrayElemAt: ["$review.review_content", y.
    index]
                                }
                            }
                        },
                        {
                            $match:{_id:{$eq:y._id}}
                        }
                    ]).toArray()[0];
                    is_top |= tmp.top_critic;
                    review_arr.push(tmp)
                    //movie_arr.push(tmp._id)
                    movie_arr.push({"movie_id": tmp._id, "primaryTitle": y.
    primaryTitle, "review_index": y.index})
                })

        name_parts = x.split(/\s/)
        first_name = name_parts.splice(0, 1)[0]
        last_name = name_parts.join(' ')

        print(100*i++/total, x, is_top)
        //print(first_name, ':', last_name)
        //print(review_arr)
        //print(movie_arr)
        db.user.insertOne(
            {
                "username": x,
                "password": "",
                "first_name": first_name,
                "last_name": last_name,
                "registration_date": new Date("2000-01-01"),
                "last_3_reviews": review_arr,
                "reviews" : movie_arr
            }
        );
        if (!is_top){
            db.user.updateOne(
                {"username": x},
                {$set:
                    {"date_of_birth": new Date("1970-07-20")}
                }
            )
        }
        print("================================")
    }
```

```
)
```

Listing 8.9: Test

C MongoDB indexes:Movie collection

C .1 primaryTitle

Before the index

```
{
  explainVersion: '1',
  queryPlanner: {
    namespace: 'rottenMovies.movie',
    indexFilterSet: false,
    parsedQuery: { primaryTitle: { '$eq': 'Evidence' } },
    queryHash: '9839850C',
    planCacheKey: '9839850C',
    maxIndexedOrSolutionsReached: false,
    maxIndexedAndSolutionsReached: false,
    maxScansToExplodeReached: false,
    winningPlan: {
      stage: 'COLLSCAN',
      filter: { primaryTitle: { '$eq': 'Evidence' } },
      direction: 'forward'
    },
    rejectedPlans: []
  },
  executionStats: {
    executionSuccess: true,
    nReturned: 1,
    executionTimeMillis: 275,
    totalKeysExamined: 0,
    totalDocsExamined: 14104,
    executionStages: {
      stage: 'COLLSCAN',
      filter: { primaryTitle: { '$eq': 'Evidence' } },
      nReturned: 1,
      executionTimeMillisEstimate: 245,
      works: 14106,
      advanced: 1,
      needTime: 14104,
      needYield: 0,
      saveState: 18,
      restoreState: 18,
      isEOF: 1,
      direction: 'forward',
      docsExamined: 14104
    }
  },
  command: {
    find: 'movie',
    filter: { primaryTitle: 'Evidence' },
    '$db': 'rottenMovies'
  },
  serverInfo: {
    host: 'Profile2022LARGE10',
    port: 27017,
    version: '6.0.3',
```

```
    gitVersion: 'f803681c3ae19817d31958965850193de067c516'
  },
  serverParameters: {
    internalQueryFacetBufferSizeBytes: 104857600,
    internalQueryFacetMaxOutputDocSizeBytes: 104857600,
    internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,
    internalDocumentSourceGroupMaxMemoryBytes: 104857600,
    internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,
    internalQueryProhibitBlockingMergeOnMongoS: 0,
    internalQueryMaxAddToSetBytes: 104857600,
    internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600
  },
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1673280853, i: 1 }),
    signature: {
      hash: Binary(Buffer.from("0000000000000000000000000000000000000000", "
    hex"), 0),
      keyId: Long("0")
    }
  },
  operationTime: Timestamp({ t: 1673280853, i: 1 })
}
```

Listing 8.10: Test

After the index

```
{
  explainVersion: '1',
  queryPlanner: {
    namespace: 'rottenMovies.movie',
    indexFilterSet: false,
    parsedQuery: { primaryTitle: { '$eq': 'Evidence' } },
    queryHash: '9839850C',
    planCacheKey: 'B734708E',
    maxIndexedOrSolutionsReached: false,
    maxIndexedAndSolutionsReached: false,
    maxScansToExplodeReached: false,
    winningPlan: {
      stage: 'FETCH',
      inputStage: {
        stage: 'IXSCAN',
        keyPattern: { primaryTitle: 1 },
        indexName: 'primaryTitle_1',
        isMultiKey: false,
        multiKeyPaths: { primaryTitle: [] },
        isUnique: false,
        isSparse: false,
        isPartial: false,
        indexVersion: 2,
        direction: 'forward',
        indexBounds: { primaryTitle: [ '["Evidence", "Evidence"]' ] }
      }
    },
    rejectedPlans: []
  },
  executionStats: {
    executionSuccess: true,
    nReturned: 1,
    executionTimeMillis: 1,
    totalKeysExamined: 1,
```

```
    totalDocsExamined: 1,
    executionStages: {
      stage: 'FETCH',
      nReturned: 1,
      executionTimeMillisEstimate: 0,
      works: 2,
      advanced: 1,
      needTime: 0,
      needYield: 0,
      saveState: 0,
      restoreState: 0,
      isEOF: 1,
      docsExamined: 1,
      alreadyHasObj: 0,
      inputStage: {
        stage: 'IXSCAN',
        nReturned: 1,
        executionTimeMillisEstimate: 0,
        works: 2,
        advanced: 1,
        needTime: 0,
        needYield: 0,
        saveState: 0,
        restoreState: 0,
        isEOF: 1,
        keyPattern: { primaryTitle: 1 },
        indexName: 'primaryTitle_1',
        isMultiKey: false,
        multiKeyPaths: { primaryTitle: [] },
        isUnique: false,
        isSparse: false,
        isPartial: false,
        indexVersion: 2,
        direction: 'forward',
        indexBounds: { primaryTitle: [ '["Evidence", "Evidence"]' ] },
        keysExamined: 1,
        seeks: 1,
        dupsTested: 0,
        dupsDropped: 0
      }
    }
  },
  command: {
    find: 'movie',
    filter: { primaryTitle: 'Evidence' },
    '$db': 'rottenMovies'
  },
  serverInfo: {
    host: 'Profile2022LARGE10',
    port: 27017,
    version: '6.0.3',
    gitVersion: 'f803681c3ae19817d31958965850193de067c516'
  },
  serverParameters: {
    internalQueryFacetBufferSizeBytes: 104857600,
    internalQueryFacetMaxOutputDocSizeBytes: 104857600,
    internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,
    internalDocumentSourceGroupMaxMemoryBytes: 104857600,
    internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,
    internalQueryProhibitBlockingMergeOnMongoS: 0,
    internalQueryMaxAddToSetBytes: 104857600,
```

```
    internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600
  },
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1673285103, i: 1 }),
    signature: {
      hash: Binary(Buffer.from("0000000000000000000000000000000000000000", "
    hex"), 0),
      keyId: Long("0")
    }
  },
  operationTime: Timestamp({ t: 1673285103, i: 1 })
}
```

Listing 8.11: Test

C .2 year

Before the index

```
{
  explainVersion: '1',
  queryPlanner: {
    namespace: 'rottenMovies.movie',
    indexFilterSet: false,
    parsedQuery: { year: { '$eq': 2012 } },
    queryHash: '412E8B51',
    planCacheKey: '412E8B51',
    maxIndexedOrSolutionsReached: false,
    maxIndexedAndSolutionsReached: false,
    maxScansToExplodeReached: false,
    winningPlan: {
      stage: 'COLLSCAN',
      filter: { year: { '$eq': 2012 } },
      direction: 'forward'
    },
    rejectedPlans: []
  },
  executionStats: {
    executionSuccess: true,
    nReturned: 480,
    executionTimeMillis: 13,
    totalKeysExamined: 0,
    totalDocsExamined: 14104,
    executionStages: {
      stage: 'COLLSCAN',
      filter: { year: { '$eq': 2012 } },
      nReturned: 480,
      executionTimeMillisEstimate: 1,
      works: 14106,
      advanced: 480,
      needTime: 13625,
      needYield: 0,
      saveState: 14,
      restoreState: 14,
      isEOF: 1,
      direction: 'forward',
      docsExamined: 14104
    }
  },
```

```
  command: { find: 'movie', filter: { year: 2012 }, '$db': 'rottenMovies' },
  serverInfo: {
    host: 'Profile2022LARGE10',
    port: 27017,
    version: '6.0.3',
    gitVersion: 'f803681c3ae19817d31958965850193de067c516'
  },
  serverParameters: {
    internalQueryFacetBufferSizeBytes: 104857600,
    internalQueryFacetMaxOutputDocSizeBytes: 104857600,
    internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,
    internalDocumentSourceGroupMaxMemoryBytes: 104857600,
    internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,
    internalQueryProhibitBlockingMergeOnMongoS: 0,
    internalQueryMaxAddToSetBytes: 104857600,
    internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600
  },
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1673280923, i: 1 }),
    signature: {
      hash: Binary(Buffer.from("0000000000000000000000000000000000000000", "
    hex"), 0),
      keyId: Long("0")
    }
  },
  operationTime: Timestamp({ t: 1673280923, i: 1 })
}
```

Listing 8.12: Test

After the index

```
{
  explainVersion: '1',
  queryPlanner: {
    namespace: 'rottenMovies.movie',
    indexFilterSet: false,
    parsedQuery: { year: { '$eq': 2012 } },
    queryHash: '412E8B51',
    planCacheKey: '62915BA3',
    maxIndexedOrSolutionsReached: false,
    maxIndexedAndSolutionsReached: false,
    maxScansToExplodeReached: false,
    winningPlan: {
      stage: 'FETCH',
      inputStage: {
        stage: 'IXSCAN',
        keyPattern: { year: 1 },
        indexName: 'year_1',
        isMultiKey: false,
        multiKeyPaths: { year: [] },
        isUnique: false,
        isSparse: false,
        isPartial: false,
        indexVersion: 2,
        direction: 'forward',
        indexBounds: { year: [ '[2012, 2012]' ] }
      }
    },
    rejectedPlans: []
  },
```

```
executionStats: {
  executionSuccess: true,
  nReturned: 480,
  executionTimeMillis: 2,
  totalKeysExamined: 480,
  totalDocsExamined: 480,
  executionStages: {
    stage: 'FETCH',
    nReturned: 480,
    executionTimeMillisEstimate: 0,
    works: 481,
    advanced: 480,
    needTime: 0,
    needYield: 0,
    saveState: 0,
    restoreState: 0,
    isEOF: 1,
    docsExamined: 480,
    alreadyHasObj: 0,
    inputStage: {
      stage: 'IXSCAN',
      nReturned: 480,
      executionTimeMillisEstimate: 0,
      works: 481,
      advanced: 480,
      needTime: 0,
      needYield: 0,
      saveState: 0,
      restoreState: 0,
      isEOF: 1,
      keyPattern: { year: 1 },
      indexName: 'year_1',
      isMultiKey: false,
      multiKeyPaths: { year: [] },
      isUnique: false,
      isSparse: false,
      isPartial: false,
      indexVersion: 2,
      direction: 'forward',
      indexBounds: { year: [ '[2012, 2012]' ] },
      keysExamined: 480,
      seeks: 1,
      dupsTested: 0,
      dupsDropped: 0
    }
  }
},
command: { find: 'movie', filter: { year: 2012 }, '$db': 'rottenMovies' },
serverInfo: {
  host: 'Profile2022LARGE10',
  port: 27017,
  version: '6.0.3',
  gitVersion: 'f803681c3ae19817d31958965850193de067c516'
},
serverParameters: {
  internalQueryFacetBufferSizeBytes: 104857600,
  internalQueryFacetMaxOutputDocSizeBytes: 104857600,
  internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,
  internalDocumentSourceGroupMaxMemoryBytes: 104857600,
  internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,
  internalQueryProhibitBlockingMergeOnMongoS: 0,
```

```
    internalQueryMaxAddToSetBytes: 104857600,
    internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600
  },
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1673285143, i: 1 }),
    signature: {
      hash: Binary(Buffer.from("0000000000000000000000000000000000000000", "
    hex"), 0),
      keyId: Long("0")
    }
  },
  operationTime: Timestamp({ t: 1673285143, i: 1 })
}
```

Listing 8.13: Test

C .3 top critic rating

Before the index

```
{
  explainVersion: '1',
  queryPlanner: {
    namespace: 'rottenMovies.movie',
    indexFilterSet: false,
    parsedQuery: {},
    queryHash: '33018E32',
    planCacheKey: '33018E32',
    maxIndexedOrSolutionsReached: false,
    maxIndexedAndSolutionsReached: false,
    maxScansToExplodeReached: false,
    winningPlan: {
      stage: 'SORT',
      sortPattern: { top_critic_rating: 1 },
      memLimit: 104857600,
      type: 'simple',
      inputStage: { stage: 'COLLSCAN', direction: 'forward' }
    },
    rejectedPlans: []
  },
  executionStats: {
    executionSuccess: true,
    nReturned: 14104,
    executionTimeMillis: 1818,
    totalKeysExamined: 0,
    totalDocsExamined: 14104,
    executionStages: {
      stage: 'SORT',
      nReturned: 14104,
      executionTimeMillisEstimate: 1740,
      works: 28211,
      advanced: 14104,
      needTime: 14106,
      needYield: 0,
      saveState: 47,
      restoreState: 47,
      isEOF: 1,
      sortPattern: { top_critic_rating: 1 },
      memLimit: 104857600,
```

```
      type: 'simple',
      totalDataSizeSorted: 262640385,
      usedDisk: true,
      spills: 3,
      inputStage: {
        stage: 'COLLSCAN',
        nReturned: 14104,
        executionTimeMillisEstimate: 0,
        works: 14106,
        advanced: 14104,
        needTime: 1,
        needYield: 0,
        saveState: 47,
        restoreState: 47,
        isEOF: 1,
        direction: 'forward',
        docsExamined: 14104
      }
    }
  },
  command: {
    find: 'movie',
    filter: {},
    sort: { top_critic_rating: 1 },
    '$db': 'rottenMovies'
  },
  serverInfo: {
    host: 'Profile2022LARGE10',
    port: 27017,
    version: '6.0.3',
    gitVersion: 'f803681c3ae19817d31958965850193de067c516'
  },
  serverParameters: {
    internalQueryFacetBufferSizeBytes: 104857600,
    internalQueryFacetMaxOutputDocSizeBytes: 104857600,
    internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,
    internalDocumentSourceGroupMaxMemoryBytes: 104857600,
    internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,
    internalQueryProhibitBlockingMergeOnMongoS: 0,
    internalQueryMaxAddToSetBytes: 104857600,
    internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600
  },
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1673287293, i: 1 }),
    signature: {
      hash: Binary(Buffer.from("0000000000000000000000000000000000000000", "
    hex"), 0),
      keyId: Long("0")
    }
  },
  operationTime: Timestamp({ t: 1673287293, i: 1 })
}
```

Listing 8.14: Test

After the index

```
{
  explainVersion: '1',
  queryPlanner: {
    namespace: 'rottenMovies.movie',
```

```
    indexFilterSet: false,
    parsedQuery: {},
    queryHash: '33018E32',
    planCacheKey: '33018E32',
    maxIndexedOrSolutionsReached: false,
    maxIndexedAndSolutionsReached: false,
    maxScansToExplodeReached: false,
    winningPlan: {
      stage: 'FETCH',
      inputStage: {
        stage: 'IXSCAN',
        keyPattern: { top_critic_rating: 1 },
        indexName: 'top_critic_rating_1',
        isMultiKey: false,
        multiKeyPaths: { top_critic_rating: [] },
        isUnique: false,
        isSparse: false,
        isPartial: false,
        indexVersion: 2,
        direction: 'forward',
        indexBounds: { top_critic_rating: [ '[MinKey, MaxKey]' ] }
      }
    },
    rejectedPlans: []
  },
  executionStats: {
    executionSuccess: true,
    nReturned: 14104,
    executionTimeMillis: 24,
    totalKeysExamined: 14104,
    totalDocsExamined: 14104,
    executionStages: {
      stage: 'FETCH',
      nReturned: 14104,
      executionTimeMillisEstimate: 5,
      works: 14105,
      advanced: 14104,
      needTime: 0,
      needYield: 0,
      saveState: 14,
      restoreState: 14,
      isEOF: 1,
      docsExamined: 14104,
      alreadyHasObj: 0,
      inputStage: {
        stage: 'IXSCAN',
        nReturned: 14104,
        executionTimeMillisEstimate: 1,
        works: 14105,
        advanced: 14104,
        needTime: 0,
        needYield: 0,
        saveState: 14,
        restoreState: 14,
        isEOF: 1,
        keyPattern: { top_critic_rating: 1 },
        indexName: 'top_critic_rating_1',
        isMultiKey: false,
        multiKeyPaths: { top_critic_rating: [] },
        isUnique: false,
        isSparse: false,
```

```
66          isPartial: false,
67          indexVersion: 2,
68          direction: 'forward',
69          indexBounds: { top_critic_rating: [ '[MinKey, MaxKey]' ] },
70          keysExamined: 14104,
71          seeks: 1,
72          dupsTested: 0,
73          dupsDropped: 0
74        }
75      }
76    },
77    command: {
78      find: 'movie',
79      filter: {},
80      sort: { top_critic_rating: 1 },
81      '$db': 'rottenMovies'
82    },
83    serverInfo: {
84      host: 'Profile2022LARGE10',
85      port: 27017,
86      version: '6.0.3',
87      gitVersion: 'f803681c3ae19817d31958965850193de067c516'
88    },
89    serverParameters: {
90      internalQueryFacetBufferSizeBytes: 104857600,
91      internalQueryFacetMaxOutputDocSizeBytes: 104857600,
92      internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,
93      internalDocumentSourceGroupMaxMemoryBytes: 104857600,
94      internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,
95      internalQueryProhibitBlockingMergeOnMongoS: 0,
96      internalQueryMaxAddToSetBytes: 104857600,
97      internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600
98    },
99    ok: 1,
100   '$clusterTime': {
101     clusterTime: Timestamp({ t: 1673285423, i: 1 }),
102     signature: {
103       hash: Binary(Buffer.from("0000000000000000000000000000000000000000", "
      hex"), 0),
104       keyId: Long("0")
105     }
106   },
107   operationTime: Timestamp({ t: 1673285423, i: 1 })
108 }
```

Listing 8.15: Test

C .4 user rating

Before the index

```
1 {
2   explainVersion: '1',
3   queryPlanner: {
4     namespace: 'rottenMovies.movie',
5     indexFilterSet: false,
6     parsedQuery: {},
7     queryHash: '3E9B1E6C',
8     planCacheKey: '3E9B1E6C',
9     maxIndexedOrSolutionsReached: false,
```

```
    maxIndexedAndSolutionsReached: false,
    maxScansToExplodeReached: false,
    winningPlan: {
      stage: 'SORT',
      sortPattern: { user_rating: 1 },
      memLimit: 104857600,
      type: 'simple',
      inputStage: { stage: 'COLLSCAN', direction: 'forward' }
    },
    rejectedPlans: []
  },
  executionStats: {
    executionSuccess: true,
    nReturned: 14104,
    executionTimeMillis: 1779,
    totalKeysExamined: 0,
    totalDocsExamined: 14104,
    executionStages: {
      stage: 'SORT',
      nReturned: 14104,
      executionTimeMillisEstimate: 1698,
      works: 28211,
      advanced: 14104,
      needTime: 14106,
      needYield: 0,
      saveState: 45,
      restoreState: 45,
      isEOF: 1,
      sortPattern: { user_rating: 1 },
      memLimit: 104857600,
      type: 'simple',
      totalDataSizeSorted: 262640385,
      usedDisk: true,
      spills: 3,
      inputStage: {
        stage: 'COLLSCAN',
        nReturned: 14104,
        executionTimeMillisEstimate: 0,
        works: 14106,
        advanced: 14104,
        needTime: 1,
        needYield: 0,
        saveState: 45,
        restoreState: 45,
        isEOF: 1,
        direction: 'forward',
        docsExamined: 14104
      }
    }
  },
  command: {
    find: 'movie',
    filter: {},
    sort: { user_rating: 1 },
    '$db': 'rottenMovies'
  },
  serverInfo: {
    host: 'Profile2022LARGE10',
    port: 27017,
    version: '6.0.3',
    gitVersion: 'f803681c3ae19817d31958965850193de067c516'
```

```
  },
  serverParameters: {
    internalQueryFacetBufferSizeBytes: 104857600,
    internalQueryFacetMaxOutputDocSizeBytes: 104857600,
    internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,
    internalDocumentSourceGroupMaxMemoryBytes: 104857600,
    internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,
    internalQueryProhibitBlockingMergeOnMongoS: 0,
    internalQueryMaxAddToSetBytes: 104857600,
    internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600
  },
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1673287353, i: 1 }),
    signature: {
      hash: Binary(Buffer.from("0000000000000000000000000000000000000000", "
    hex"), 0),
      keyId: Long("0")
    }
  },
  operationTime: Timestamp({ t: 1673287353, i: 1 })
}
```

Listing 8.16: Test

After the index

```
{
  explainVersion: '1',
  queryPlanner: {
    namespace: 'rottenMovies.movie',
    indexFilterSet: false,
    parsedQuery: {},
    queryHash: '3E9B1E6C',
    planCacheKey: '3E9B1E6C',
    maxIndexedOrSolutionsReached: false,
    maxIndexedAndSolutionsReached: false,
    maxScansToExplodeReached: false,
    winningPlan: {
      stage: 'FETCH',
      inputStage: {
        stage: 'IXSCAN',
        keyPattern: { user_rating: 1 },
        indexName: 'user_rating_1',
        isMultiKey: false,
        multiKeyPaths: { user_rating: [] },
        isUnique: false,
        isSparse: false,
        isPartial: false,
        indexVersion: 2,
        direction: 'forward',
        indexBounds: { user_rating: [ '[MinKey, MaxKey]' ] }
      }
    },
    rejectedPlans: []
  },
  executionStats: {
    executionSuccess: true,
    nReturned: 14104,
    executionTimeMillis: 25,
    totalKeysExamined: 14104,
    totalDocsExamined: 14104,
```

```
    executionStages: {
      stage: 'FETCH',
      nReturned: 14104,
      executionTimeMillisEstimate: 5,
      works: 14105,
      advanced: 14104,
      needTime: 0,
      needYield: 0,
      saveState: 14,
      restoreState: 14,
      isEOF: 1,
      docsExamined: 14104,
      alreadyHasObj: 0,
      inputStage: {
        stage: 'IXSCAN',
        nReturned: 14104,
        executionTimeMillisEstimate: 2,
        works: 14105,
        advanced: 14104,
        needTime: 0,
        needYield: 0,
        saveState: 14,
        restoreState: 14,
        isEOF: 1,
        keyPattern: { user_rating: 1 },
        indexName: 'user_rating_1',
        isMultiKey: false,
        multiKeyPaths: { user_rating: [] },
        isUnique: false,
        isSparse: false,
        isPartial: false,
        indexVersion: 2,
        direction: 'forward',
        indexBounds: { user_rating: [ '[MinKey, MaxKey]' ] },
        keysExamined: 14104,
        seeks: 1,
        dupsTested: 0,
        dupsDropped: 0
      }
    }
  },
  command: {
    find: 'movie',
    filter: {},
    sort: { user_rating: 1 },
    '$db': 'rottenMovies'
  },
  serverInfo: {
    host: 'Profile2022LARGE10',
    port: 27017,
    version: '6.0.3',
    gitVersion: 'f803681c3ae19817d31958965850193de067c516'
  },
  serverParameters: {
    internalQueryFacetBufferSizeBytes: 104857600,
    internalQueryFacetMaxOutputDocSizeBytes: 104857600,
    internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,
    internalDocumentSourceGroupMaxMemoryBytes: 104857600,
    internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,
    internalQueryProhibitBlockingMergeOnMongoS: 0,
    internalQueryMaxAddToSetBytes: 104857600,
```

```
    internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600
  },
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1673285383, i: 1 }),
    signature: {
      hash: Binary(Buffer.from("0000000000000000000000000000000000000000", "
    hex"), 0),
      keyId: Long("0")
    }
  },
  operationTime: Timestamp({ t: 1673285383, i: 1 })
}
```

Listing 8.17: Test

C .5 personnel.primaryName

Before the index

```
{
  explainVersion: '1',
  queryPlanner: {
    namespace: 'rottenMovies.movie',
    indexFilterSet: false,
    parsedQuery: { 'personnel.primaryName': { '$eq': '' } },
    queryHash: 'E212F03B',
    planCacheKey: 'E212F03B',
    maxIndexedOrSolutionsReached: false,
    maxIndexedAndSolutionsReached: false,
    maxScansToExplodeReached: false,
    winningPlan: {
      stage: 'COLLSCAN',
      filter: { 'personnel.primaryName': { '$eq': '' } },
      direction: 'forward'
    },
    rejectedPlans: []
  },
  executionStats: {
    executionSuccess: true,
    nReturned: 0,
    executionTimeMillis: 47,
    totalKeysExamined: 0,
    totalDocsExamined: 14104,
    executionStages: {
      stage: 'COLLSCAN',
      filter: { 'personnel.primaryName': { '$eq': '' } },
      nReturned: 0,
      executionTimeMillisEstimate: 9,
      works: 14106,
      advanced: 0,
      needTime: 14105,
      needYield: 0,
      saveState: 14,
      restoreState: 14,
      isEOF: 1,
      direction: 'forward',
      docsExamined: 14104
    }
  },
```

```
  command: {
    find: 'movie',
    filter: { 'personnel.primaryName': '' },
    '$db': 'rottenMovies'
  },
  serverInfo: {
    host: 'Profile2022LARGE10',
    port: 27017,
    version: '6.0.3',
    gitVersion: 'f803681c3ae19817d31958965850193de067c516'
  },
  serverParameters: {
    internalQueryFacetBufferSizeBytes: 104857600,
    internalQueryFacetMaxOutputDocSizeBytes: 104857600,
    internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,
    internalDocumentSourceGroupMaxMemoryBytes: 104857600,
    internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,
    internalQueryProhibitBlockingMergeOnMongoS: 0,
    internalQueryMaxAddToSetBytes: 104857600,
    internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600
  },
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1673287593, i: 1 }),
    signature: {
      hash: Binary(Buffer.from("0000000000000000000000000000000000000000", "
    hex"), 0),
      keyId: Long("0")
    }
  },
  operationTime: Timestamp({ t: 1673287593, i: 1 })
}
```

Listing 8.18: Test

After the index

```
{
  explainVersion: '1',
  queryPlanner: {
    namespace: 'rottenMovies.movie',
    indexFilterSet: false,
    parsedQuery: { 'personnel.primaryName': { '$eq': '' } },
    queryHash: 'E212F03B',
    planCacheKey: '9D4A6814',
    maxIndexedOrSolutionsReached: false,
    maxIndexedAndSolutionsReached: false,
    maxScansToExplodeReached: false,
    winningPlan: {
      stage: 'FETCH',
      inputStage: {
        stage: 'IXSCAN',
        keyPattern: { 'personnel.primaryName': 1 },
        indexName: 'personnel.primaryName_1',
        isMultiKey: true,
        multiKeyPaths: { 'personnel.primaryName': [ 'personnel' ] },
        isUnique: false,
        isSparse: false,
        isPartial: false,
        indexVersion: 2,
        direction: 'forward',
        indexBounds: { 'personnel.primaryName': [ '["", "")' ] }
```

```
      }
    },
    rejectedPlans: []
  },
  executionStats: {
    executionSuccess: true,
    nReturned: 0,
    executionTimeMillis: 0,
    totalKeysExamined: 0,
    totalDocsExamined: 0,
    executionStages: {
      stage: 'FETCH',
      nReturned: 0,
      executionTimeMillisEstimate: 0,
      works: 1,
      advanced: 0,
      needTime: 0,
      needYield: 0,
      saveState: 0,
      restoreState: 0,
      isEOF: 1,
      docsExamined: 0,
      alreadyHasObj: 0,
      inputStage: {
        stage: 'IXSCAN',
        nReturned: 0,
        executionTimeMillisEstimate: 0,
        works: 1,
        advanced: 0,
        needTime: 0,
        needYield: 0,
        saveState: 0,
        restoreState: 0,
        isEOF: 1,
        keyPattern: { 'personnel.primaryName': 1 },
        indexName: 'personnel.primaryName_1',
        isMultiKey: true,
        multiKeyPaths: { 'personnel.primaryName': [ 'personnel' ] },
        isUnique: false,
        isSparse: false,
        isPartial: false,
        indexVersion: 2,
        direction: 'forward',
        indexBounds: { 'personnel.primaryName': [ '["", "")' ] },
        keysExamined: 0,
        seeks: 1,
        dupsTested: 0,
        dupsDropped: 0
      }
    }
  },
  command: {
    find: 'movie',
    filter: { 'personnel.primaryName': '' },
    '$db': 'rottenMovies'
  },
  serverInfo: {
    host: 'Profile2022LARGE10',
    port: 27017,
    version: '6.0.3',
    gitVersion: 'f803681c3ae19817d31958965850193de067c516'
```

```
  },
  serverParameters: {
    internalQueryFacetBufferSizeBytes: 104857600,
    internalQueryFacetMaxOutputDocSizeBytes: 104857600,
    internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,
    internalDocumentSourceGroupMaxMemoryBytes: 104857600,
    internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,
    internalQueryProhibitBlockingMergeOnMongoS: 0,
    internalQueryMaxAddToSetBytes: 104857600,
    internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600
  },
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1673287763, i: 1 }),
    signature: {
      hash: Binary(Buffer.from("0000000000000000000000000000000000000000", "
    hex"), 0),
      keyId: Long("0")
    }
  },
  operationTime: Timestamp({ t: 1673287763, i: 1 })
}
```

Listing 8.19: Test

D MongoDB indexes:User collection

D .1 username

Before the index

```
{
  explainVersion: '1',
  queryPlanner: {
    namespace: 'rottenMovies.user',
    indexFilterSet: false,
    parsedQuery: { username: { '$eq': 'Abbie Bernstein' } },
    queryHash: '7D9BB680',
    planCacheKey: '7D9BB680',
    maxIndexedOrSolutionsReached: false,
    maxIndexedAndSolutionsReached: false,
    maxScansToExplodeReached: false,
    winningPlan: {
      stage: 'COLLSCAN',
      filter: { username: { '$eq': 'Abbie Bernstein' } },
      direction: 'forward'
    },
    rejectedPlans: []
  },
  executionStats: {
    executionSuccess: true,
    nReturned: 1,
    executionTimeMillis: 6,
    totalKeysExamined: 0,
    totalDocsExamined: 8339,
    executionStages: {
      stage: 'COLLSCAN',
      filter: { username: { '$eq': 'Abbie Bernstein' } },
      nReturned: 1,
```

```
      executionTimeMillisEstimate: 0,
      works: 8341,
      advanced: 1,
      needTime: 8339,
      needYield: 0,
      saveState: 8,
      restoreState: 8,
      isEOF: 1,
      direction: 'forward',
      docsExamined: 8339
    }
  },
  command: {
    find: 'user',
    filter: { username: 'Abbie Bernstein' },
    '$db': 'rottenMovies'
  },
  serverInfo: {
    host: 'Profile2022LARGE10',
    port: 27017,
    version: '6.0.3',
    gitVersion: 'f803681c3ae19817d31958965850193de067c516'
  },
  serverParameters: {
    internalQueryFacetBufferSizeBytes: 104857600,
    internalQueryFacetMaxOutputDocSizeBytes: 104857600,
    internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,
    internalDocumentSourceGroupMaxMemoryBytes: 104857600,
    internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,
    internalQueryProhibitBlockingMergeOnMongoS: 0,
    internalQueryMaxAddToSetBytes: 104857600,
    internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600
  },
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1673280753, i: 1 }),
    signature: {
      hash: Binary(Buffer.from("0000000000000000000000000000000000000000", "
    hex"), 0),
      keyId: Long("0")
    }
  },
  operationTime: Timestamp({ t: 1673280753, i: 1 })
}
```

Listing 8.20: Test

After the index

```
{
  explainVersion: '1',
  queryPlanner: {
    namespace: 'rottenMovies.user',
    indexFilterSet: false,
    parsedQuery: { username: { '$eq': 'Abbie Bernstein' } },
    queryHash: '7D9BB680',
    planCacheKey: '24069050',
    maxIndexedOrSolutionsReached: false,
    maxIndexedAndSolutionsReached: false,
    maxScansToExplodeReached: false,
    winningPlan: {
      stage: 'FETCH',
```

```
14        inputStage: {
15          stage: 'IXSCAN',
16          keyPattern: { username: 1 },
17          indexName: 'username_1',
18          isMultiKey: false,
19          multiKeyPaths: { username: [] },
20          isUnique: false,
21          isSparse: false,
22          isPartial: false,
23          indexVersion: 2,
24          direction: 'forward',
25          indexBounds: { username: [ '["Abbie Bernstein", "Abbie Bernstein"]' ]
      }
26        }
27      },
28      rejectedPlans: []
29    },
30    executionStats: {
31      executionSuccess: true,
32      nReturned: 1,
33      executionTimeMillis: 1,
34      totalKeysExamined: 1,
35      totalDocsExamined: 1,
36      executionStages: {
37        stage: 'FETCH',
38        nReturned: 1,
39        executionTimeMillisEstimate: 1,
40        works: 2,
41        advanced: 1,
42        needTime: 0,
43        needYield: 0,
44        saveState: 0,
45        restoreState: 0,
46        isEOF: 1,
47        docsExamined: 1,
48        alreadyHasObj: 0,
49        inputStage: {
50          stage: 'IXSCAN',
51          nReturned: 1,
52          executionTimeMillisEstimate: 1,
53          works: 2,
54          advanced: 1,
55          needTime: 0,
56          needYield: 0,
57          saveState: 0,
58          restoreState: 0,
59          isEOF: 1,
60          keyPattern: { username: 1 },
61          indexName: 'username_1',
62          isMultiKey: false,
63          multiKeyPaths: { username: [] },
64          isUnique: false,
65          isSparse: false,
66          isPartial: false,
67          indexVersion: 2,
68          direction: 'forward',
69          indexBounds: { username: [ '["Abbie Bernstein", "Abbie Bernstein"]' ]
      },
70          keysExamined: 1,
71          seeks: 1,
72          dupsTested: 0,
```

```
        dupsDropped: 0
      }
    }
  },
  command: {
    find: 'user',
    filter: { username: 'Abbie Bernstein' },
    '$db': 'rottenMovies'
  },
  serverInfo: {
    host: 'Profile2022LARGE10',
    port: 27017,
    version: '6.0.3',
    gitVersion: 'f803681c3ae19817d31958965850193de067c516'
  },
  serverParameters: {
    internalQueryFacetBufferSizeBytes: 104857600,
    internalQueryFacetMaxOutputDocSizeBytes: 104857600,
    internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,
    internalDocumentSourceGroupMaxMemoryBytes: 104857600,
    internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,
    internalQueryProhibitBlockingMergeOnMongoS: 0,
    internalQueryMaxAddToSetBytes: 104857600,
    internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600
  },
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1673285013, i: 1 }),
    signature: {
      hash: Binary(Buffer.from("0000000000000000000000000000000000000000", "
    hex"), 0),
      keyId: Long("0")
    }
  },
  operationTime: Timestamp({ t: 1673285013, i: 1 })
}
```

Listing 8.21: Test

D .2 date of birth

Before the index

```
{
  explainVersion: '1',
  queryPlanner: {
    namespace: 'rottenMovies.user',
    indexFilterSet: false,
    parsedQuery: {
      date_of_birth: {
        '$eq': 'Mon Jan 09 2023 17:05:07 GMT+0000 (Western European Standard
    Time)'
      }
    },
    queryHash: 'D7A0117C',
    planCacheKey: 'D7A0117C',
    maxIndexedOrSolutionsReached: false,
    maxIndexedAndSolutionsReached: false,
    maxScansToExplodeReached: false,
    winningPlan: {
```

```
      stage: 'COLLSCAN',
      filter: {
        date_of_birth: {
          '$eq': 'Mon Jan 09 2023 17:05:07 GMT+0000 (Western European Standard Time)'
        }
      },
      direction: 'forward'
    },
    rejectedPlans: []
  },
  executionStats: {
    executionSuccess: true,
    nReturned: 0,
    executionTimeMillis: 32,
    totalKeysExamined: 0,
    totalDocsExamined: 8339,
    executionStages: {
      stage: 'COLLSCAN',
      filter: {
        date_of_birth: {
          '$eq': 'Mon Jan 09 2023 17:05:07 GMT+0000 (Western European Standard Time)'
        }
      },
      nReturned: 0,
      executionTimeMillisEstimate: 23,
      works: 8341,
      advanced: 0,
      needTime: 8340,
      needYield: 0,
      saveState: 8,
      restoreState: 8,
      isEOF: 1,
      direction: 'forward',
      docsExamined: 8339
    }
  },
  command: {
    find: 'user',
    filter: {
      date_of_birth: 'Mon Jan 09 2023 17:05:07 GMT+0000 (Western European Standard Time)'
    },
    '$db': 'rottenMovies'
  },
  serverInfo: {
    host: 'Profile2022LARGE10',
    port: 27017,
    version: '6.0.3',
    gitVersion: 'f803681c3ae19817d31958965850193de067c516'
  },
  serverParameters: {
    internalQueryFacetBufferSizeBytes: 104857600,
    internalQueryFacetMaxOutputDocSizeBytes: 104857600,
    internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,
    internalDocumentSourceGroupMaxMemoryBytes: 104857600,
    internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,
    internalQueryProhibitBlockingMergeOnMongoS: 0,
    internalQueryMaxAddToSetBytes: 104857600,
    internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600
```

```
  },
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1673283903, i: 1 }),
    signature: {
      hash: Binary(Buffer.from("0000000000000000000000000000000000000000", "
    hex"), 0),
      keyId: Long("0")
    }
  },
  operationTime: Timestamp({ t: 1673283903, i: 1 })
}
```

Listing 8.22: Test

After the index

```
{
  explainVersion: '1',
  queryPlanner: {
    namespace: 'rottenMovies.user',
    indexFilterSet: false,
    parsedQuery: {
      date_of_birth: {
        '$eq': 'Mon Jan 09 2023 17:24:42 GMT+0000 (Western European Standard
    Time)'
      }
    },
    queryHash: 'D7A0117C',
    planCacheKey: '90F68BB6',
    maxIndexedOrSolutionsReached: false,
    maxIndexedAndSolutionsReached: false,
    maxScansToExplodeReached: false,
    winningPlan: {
      stage: 'FETCH',
      inputStage: {
        stage: 'IXSCAN',
        keyPattern: { date_of_birth: 1 },
        indexName: 'date_of_birth_1',
        isMultiKey: false,
        multiKeyPaths: { date_of_birth: [] },
        isUnique: false,
        isSparse: false,
        isPartial: false,
        indexVersion: 2,
        direction: 'forward',
        indexBounds: {
          date_of_birth: [
            '["Mon Jan 09 2023 17:24:42 GMT+0000 (Western European Standard
    Time)", "Mon Jan 09 2023 17:24:42 GMT+0000 (Western European Standard
    Time)"]'
          ]
        }
      }
    },
    rejectedPlans: []
  },
  executionStats: {
    executionSuccess: true,
    nReturned: 0,
    executionTimeMillis: 1,
    totalKeysExamined: 0,
```

```
    totalDocsExamined: 0,
    executionStages: {
      stage: 'FETCH',
      nReturned: 0,
      executionTimeMillisEstimate: 0,
      works: 1,
      advanced: 0,
      needTime: 0,
      needYield: 0,
      saveState: 0,
      restoreState: 0,
      isEOF: 1,
      docsExamined: 0,
      alreadyHasObj: 0,
      inputStage: {
        stage: 'IXSCAN',
        nReturned: 0,
        executionTimeMillisEstimate: 0,
        works: 1,
        advanced: 0,
        needTime: 0,
        needYield: 0,
        saveState: 0,
        restoreState: 0,
        isEOF: 1,
        keyPattern: { date_of_birth: 1 },
        indexName: 'date_of_birth_1',
        isMultiKey: false,
        multiKeyPaths: { date_of_birth: [] },
        isUnique: false,
        isSparse: false,
        isPartial: false,
        indexVersion: 2,
        direction: 'forward',
        indexBounds: {
          date_of_birth: [
            '["Mon Jan 09 2023 17:24:42 GMT+0000 (Western European Standard
    Time)", "Mon Jan 09 2023 17:24:42 GMT+0000 (Western European Standard
    Time)"]'
          ]
        },
        keysExamined: 0,
        seeks: 1,
        dupsTested: 0,
        dupsDropped: 0
      }
    }
  },
  command: {
    find: 'user',
    filter: {
      date_of_birth: 'Mon Jan 09 2023 17:24:42 GMT+0000 (Western European
    Standard Time)'
    },
    '$db': 'rottenMovies'
  },
  serverInfo: {
    host: 'Profile2022LARGE10',
    port: 27017,
    version: '6.0.3',
    gitVersion: 'f803681c3ae19817d31958965850193de067c516'
```

```
  },
  serverParameters: {
    internalQueryFacetBufferSizeBytes: 104857600,
    internalQueryFacetMaxOutputDocSizeBytes: 104857600,
    internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,
    internalDocumentSourceGroupMaxMemoryBytes: 104857600,
    internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,
    internalQueryProhibitBlockingMergeOnMongoS: 0,
    internalQueryMaxAddToSetBytes: 104857600,
    internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600
  },
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1673285073, i: 1 }),
    signature: {
      hash: Binary(Buffer.from("0000000000000000000000000000000000000000", "
    hex"), 0),
      keyId: Long("0")
    }
  },
  operationTime: Timestamp({ t: 1673285073, i: 1 })
}
```

Listing 8.23: Test

E Application code

E .1 Pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.
    org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache
    .org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.0.0</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>it.unipi.dii.lsmsdb</groupId>
    <artifactId>rottenMovies</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>rottenMovies</name>
    <description>Project for the rotten movies service</description>
    <properties>
        <java.version>19</java.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-thymeleaf</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
```

```
28          <dependency>
29              <groupId>org.springframework.boot</groupId>
30              <artifactId>spring-boot-starter-data-mongodb</artifactId>
31          </dependency>
32          <dependency>
33              <groupId>org.springframework.boot</groupId>
34              <artifactId>spring-boot-starter-data-mongodb-reactive</artifactId>
35          </dependency>
36          <dependency>
37              <groupId>org.springframework.boot</groupId>
38              <artifactId>spring-boot-starter-data-neo4j</artifactId>
39          </dependency>
40
41          <dependency>
42              <groupId>org.springframework.boot</groupId>
43              <artifactId>spring-boot-starter-test</artifactId>
44              <scope>test</scope>
45          </dependency>
46          <dependency>
47              <groupId>io.projectreactor</groupId>
48              <artifactId>reactor-test</artifactId>
49              <scope>test</scope>
50          </dependency>
51          <dependency>
52              <groupId>com.google.code.gson</groupId>
53              <artifactId>gson</artifactId>
54              <version>2.10</version>
55          </dependency>
56          <dependency>
57              <groupId>com.fasterxml.jackson.core</groupId>
58              <artifactId>jackson-annotations</artifactId>
59              <version>2.14.1</version>
60          </dependency>
61          <dependency>
62              <groupId>com.fasterxml.jackson.core</groupId>
63              <artifactId>jackson-databind</artifactId>
64              <version>2.14.0</version>
65          </dependency>
66          <dependency>
67              <groupId>org.neo4j.driver</groupId>
68              <artifactId>neo4j-java-driver</artifactId>
69              <version>5.3.0</version>
70          </dependency>
71      </dependencies>
72
73      <build>
74          <plugins>
75              <plugin>
76                  <groupId>org.springframework.boot</groupId>
77                  <artifactId>spring-boot-maven-plugin</artifactId>
78              </plugin>
79          </plugins>
80      </build>
81
82  </project>
```

Listing 8.24: Test