Large-Scale and Multi-Structured Databases

Project Documentation
GameCritic

Filippo Lucchesi
Ettore Ricci
Martina Speciale

University of Pisa
Department of Information Engineering
Academic Year 2023/2024

Contents

Chapter 1

Introduction

GameCritic is a social-network application that allows its users to share their opinion about videogames. Among other features, regular users can review games and comment on other users' reviews, while companies that developed or produced a game can run various analytics on it.
The source code for the application is available at `https://github.com/Etto48/LargeScaleProject.git`

Chapter 2

Dataset and Web Scraping

To populate our data, we did both web scraping and random data generation.

2.1 Data Scraping

The two sources for the scraping are the following:

- **MetaCritic**
- **MobyGames**

From the first we retrieved most of the reviews and usernames of our database. In particular, we retrieved the score, date, author name, text, and the name of the game reviewed. From the second one, we retrieved all of the information concerning videogames. We kept the most important attributes, such as Name, Genre, Release Date etc. and we discarded some MobyGames-specific attributes that were not relevant in our use case.

2.2 Data Generation

Some games that we found on MobyGames were not present on MetaCritic, so we randomly generated some reviews and users to associate to those games. We also randomly generated all comments found on all reviews, since none could be found on MetaCritic's reviews. To do so, we used Python algorithms.

2.3 Resulting Dataset

The final volume wanders around 400 MB, which are the result of:

- 70k videogames

- 250k reviews
- 625k comments

Chapter 3

Design

3.1 Actors

The main actors of the application are:

- *Non-Logged User (Guest User)* : anonymous users that access the application. They can either sign up or log in.
- *User* : end-user of the application.
- *Company Manager* : it's a special kind of *user*, who is granted more benefits. Company Managers identify Game Developers/Publishers that, as such, are able to view and run analytics over their own products.
- *Administrator* : *users* that can run and view analytics that concern the whole database. They are the ones who manage the application: they are able to delete any type of content, update information about Games or Users and to ban them at the occurrence.

3.2 Requirements

3.2.1 Functional Requirements

What follows is a list of the functional requirements.

Guest User

- sign up
- log in

All Users

- view other users' profiles
- view companies' profiles
- view information about videogames

All Users except Guest Users

- view recommended games
- view recommended users
- follow/unfollow users
- "like" a review
- view and edit their profile info
- review a videogame
- comment on another user's review

Company Manager Exclusive

Note: all of the following apply only to games published or developed by the company the *company manager* represents

- add a videogame to the database
- modify info about a game
- delete a videogame from the database
- view the score distribution for the company's videogames
- view the top games by average score
- view the best game of the company (by average score)

Administrator Exclusive

- delete (”ban”) a user from the database
- delete a review from the database
- delete a comment from the database
- delete a videogame from the database
- view the top users by number of likes received on their reviews
- view the top users by number of reviews published
- view the most active user (by number of reviews posted in the last 6 month)
- view the user with most likes received
- view the global distribution of the review score

3.2.2 Non-Functional Requirements

The following is a list of all non-functional requirements.

- *Usability*: the application must have a user-friendly interface and have low response times
- *Availability*: the service provided by the application must be always available to all users
- *Reliability*: the application must be stable during its use and it must return reproducible results
- *Flexibility*: company managers should be able to add any attribute to a game they want to publish, and the application should account for this
- *Portability*: the application must be executable in different operating systems without changes in its behaviour
- *Privacy*: every user’s information should be handled securely
- *Maintainability*: the code should be modular and easy to read.

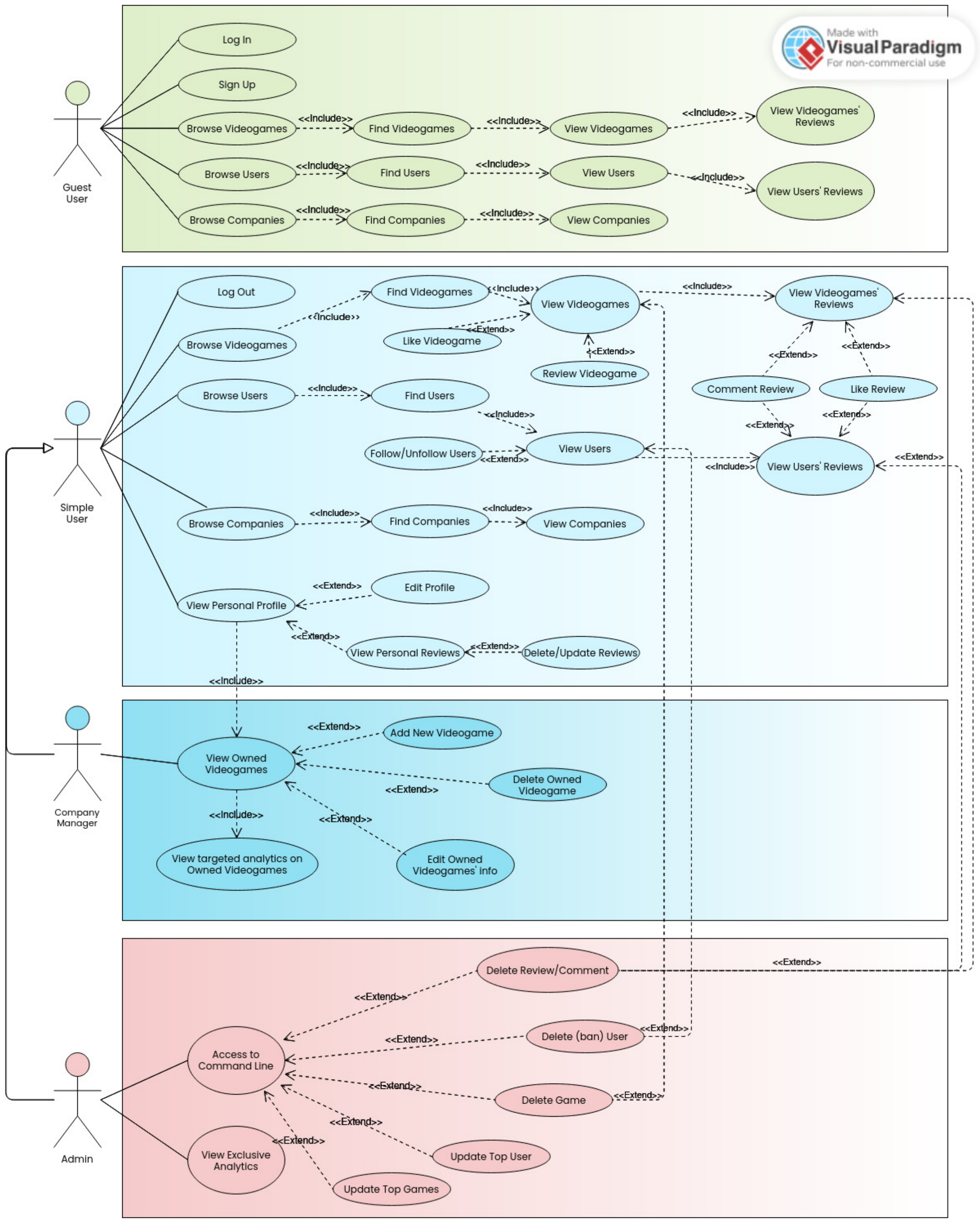

Figure 3.1: Actors and main supported functionalities

3.3 Use Case Diagram

We can look at the use case diagram of the application in Figure 3.1 *Guest users* are able to look at all the content available on the platform, but they can not either review videgames, comment reviews or like them, all actions that only registered users (*Simple Users* that are logged) can perform. Users registered as *Company Managers*, that are associated to some specific *Company* that either developed or published *Videogames* available on the platform, are given the possibility to perform some specific actions in order to assess the consumers' sentiment regarding their products. Moreover they are able to edit the pages related to their own *Videogames*. *Admin Users* are the ones that handle the content over the whole platform. Both *Company Managers* and *Administrators* can still perform all actions linked to a *Simple User*.

3.4 UML Class Diagram

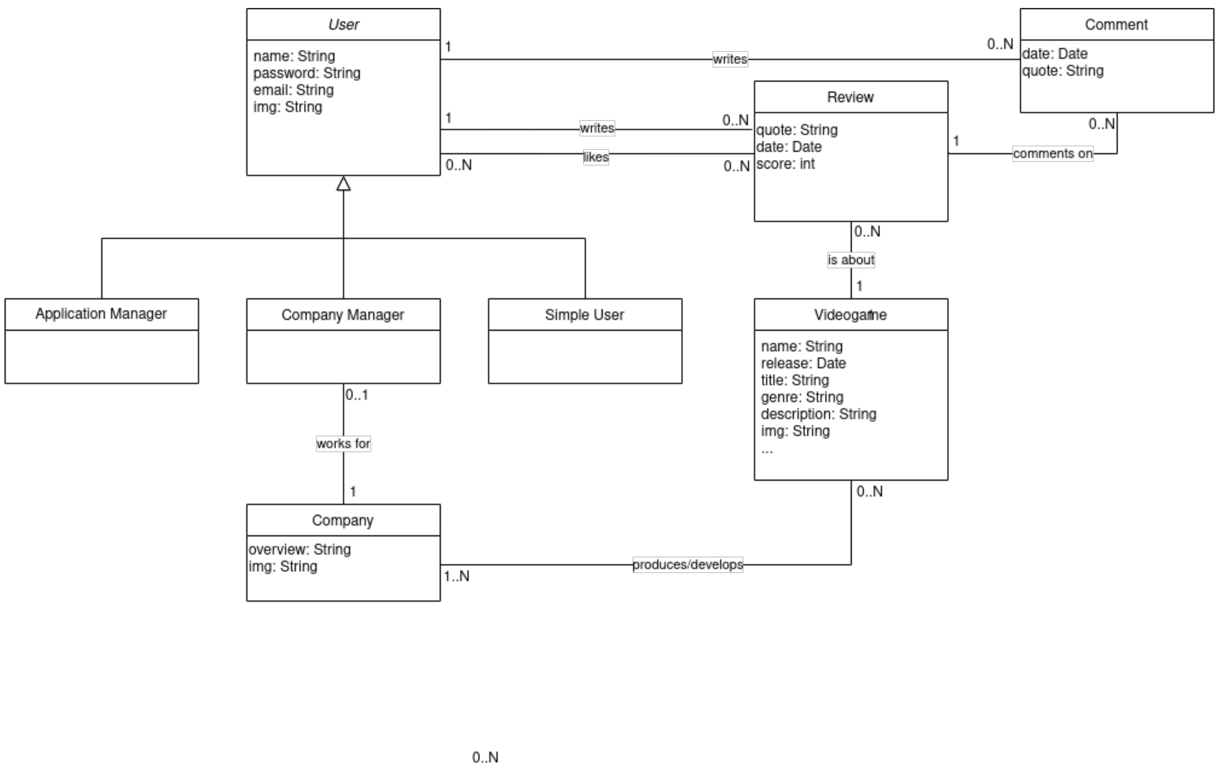

Figure 3.2: UML Class Diagram

The UML class diagram is reported in Figure 3.2

3.4.1 Relationships between classes

- A *User* can be either be a *Simple User*, a *Company Manager* or an *Administrator*.
- a *User*, whatever its specific role, can be associated with zero or more *Reviews* they have written

- Each *Review* has a specific author (it is written by one *User* alone)
- a *User* can like zero or more *Reviews*
- Each *Review* may have been liked by zero or more *Users*
- a *User* may have written zero or more *Comments* attached to some *Review* (either written by themselves or by other *Users*)
- Each *Comment* has a specific author (*User*)
- Each *User* has a *User Image* linked to their personal profile (a default Image gets associated to each user, that then has the possibility to change it by editing his personal profile)
- Each *User Image* is linked to a specific *User*
- Each *Comment* is related to one specific *Review*
- Each *Review* may have zero or more *Comments*
- Each *Review* is associated to a specific *Videogame*
- Each *Videogame* may have zero or more *Reviews* that talk about it
- Every *Videogame* is linked to one or more *Companies*, that either developed or published it
- Each *Company* registered on the platform is associated to zero or more *Videogames*
- Every *Company Manager* is linked to the *Company* they work for
- Each *Company* may or may not have one *Company Manager* registered as a user (with privileges discussed later on) over the platform

3.4.2 Design of the classes

Here we offer a brief description of some of the entities' attributes

User - Simple User, Company Manager, Admin

- `name` : a string that contains the username associated to a specific user. It is unique for each user stored inside the database
- `password` : string that contains the hash of the password the user has to enter in order to log in

- `email` : a string that contains the email offered during the *sign up* procedure to create an account

Review

- `quote` : a string that contains the text of the review
- `date` : timestamp that keeps track of the day the review has been posted
- `score` : an integer that represents the review score - ranging from 0 to 10 - that is linked to the review and is used to make analytics

Comment

- `date` : timestamp that keeps track of the day the comment has been posted
- `quote` : a string that contains the content of the comment

Videogame

- `name` : a string that contains name of the videogame
- `release` : date on which the videogame was released
- `genre` : a string that contains the genre to which the game belongs to (i.g. Action, Platform Games, Adventure, Puzzle etc.)
- `description` : a string that contains a brief description of the videogame
- `img` : a string that contains the URL of the image of the videogame's cover

We listed the more common attributes, but we remember the number and the type of attributes may change from one videogame document to another

Company

- `overview` : a string that describes the profile of the company

User Image

- `img` : a string that contains the URL of the image used as an icon to associate to a specific user

Chapter 4

Data Model

4.1 Document Database - MongoDB

Starting to reason from the queries we wanted to run against our database, we started the modeling process deciding to use a Document Database, that allows for scalability and flexibility. It did seem like the most reasonable choice by looking at the amount of data we needed to deal with.

4.1.1 Document DB Collections

Videogames

Whenever a videogame document is inserted, company managers may choose any number of attributes to add to it, of any significance. However, we defined the following attributes to be mandatory when a new videogame is created:

- `Name`
- `Description`
- `Image`
- `Release Date`
- `Platform`
- `Publishers`
- `Developers`
- `Genre`

Of those, *Name*, *Release Date*, *Publishers* and *Developers* are the attributes that are the focus of the analytic queries that we run. A videogame also has the following attributes:

- `reviews`: here we find all the reviews related to a specific videogame, in a simplified form; the attributes stored are ”reviewId, score, author, date”.
- `user_review`: it holds the average score for the game.
- `reviewCount`: it holds the total count of reviews for the game.
- `Top3ReviewsByLikes`: it embeds the top 3 reviews (with all their attributes) for the game by amount of likes received.

The first three are updated every time a review is added or deleted from the database. Updating the last one is an expensive operation, so it is done only periodically, or whenever an administrator requests it.

Users

As previously said a *user* can either be a simple *reviewer*, a *Company Manager* or an *Administrator* (look below at attributes `company_name` ans `is_admin` to see how to discriminate between the various types of users). Here we have some of the attributes we may find in a typical user's document :

- `username` : it is a unique string that identifies a specific user into the database. It is set during the *sign up* procedure after checking the chosen username is not associated to an already existing account.
- `email` : string that contains the email given during the *sign up* procedure.
- `password_hash` : string that contains the hash of the password chosen during the *sign up* procedure .
- `company_name` : string that if present into a user's document identifies that user as a *Company Manager*.
- `is_admin` : boolean value, if set to *true* in a user's document than that user is identified as an *Administrator*.

Companies

- `Name` : string that uniquely identifies a Company. It is mandatory to have a name associated to each Company in the system.

- `imglink` : string that may or may not be present, containing the URL of an image linked to the specific Company it is attached to.
- `Overview` : string that may or may not be present. It contains informations about the Company.
- `Top3Games` : here we have the top three videogames owned by the Company embedded. The average score of each videogame may of course change over time, so we periodically update the three embedded documents (if needed). Being a quite expensive operation, the update gets scheduled periodically - or whenever the administrator requests it.

Reviews

We did consider the possibility to embed entirely the reviews (i.e. including their actual content, the text - here into the `quote` attribute) into the videogames collection (or even the users one), but we soon realized the amount of expected data, all together, could grow too much, making the aggregations we perform over reviews heavier and much slower. Moreover we'd have a maximum number of storable reviews that would get around $25k$ reviews per videogame (or user). We settled for a reduced version (not including content) of each review embedded into the specific document representing the videogame it was written for. This way we have a list for every videogame of its reviews, which is accessible directly from the videogames' collection, plus the review collection that contains all the reviews' details.

- `score` : each review has to have a score attached to it.
- `quote` : text of the review.
- `author` : each review belongs to a specific user, here we find the related `username`.
- `date` : date of creation of the review in format `yyyy-MM-dd`.
- `likes` : integer that holds the number of likes of each specific review, updated at every added or deleted like.

Comments

- `reviewId`: id of the review the comment is for.
- `author` : each comment belongs to a specific user, here we find the related `username`.
- `quote` : text of the comment.

- `date` : date of creation of the comment in format `yyyy-MM-dd`.

As in the previous case we considered the possibility to embed comments into the reviews collection, each comment as an element of an array containing all the comments related to a specific review, but just as before we realized the document size could grow quickly with the average amount of comments we expect to have for each review. Again, aggregations would get heavier and that is what makes embedding not a viable possibility, even more so if we consider we are dealing with a relatively *small* social network and some of the heavier aggregations (especially the ones we need to perform relatively rarely), already as it is, take in the order of $10s$ to complete.

User Images

- `username`
- `image`

We decided to add this redundancy to improve the performance of all the operations that require to show users' images, having compared the performance results obtained with and without it.

4.1.2 Document DB Examples

Videogame

```
{
  "_id": {
    "$oid": "000000000000000000000001"
  },
  "Name": "(Almost) Total Mayhem",
  "Released": {
    "Release Date": "2011-01-14",
    "Platform": "Xbox 360"
  },
  "Publishers": "Peanut Gallery",
  "Developers": "Peanut Gallery",
  "Genre": "Action",
  "Perspective": "Side view",
  "Gameplay": "Platform",
  "Setting": "Fantasy",
  "Media Type": "Download",
  "Multiplayer Options": "Same/Split-Screen",
  "Number of Offline Players": "1-2 Players",
  "Description": "(Almost) Total Mayhem is a 2D team-based action platformer ...",
  "img": "https://cdn.mobygames.com/4b4ee410-ab7c-11ed-93d8-02420a000198.webp",
  "user_review": 6,
  "reviews": [
    {
      "reviewId": {
        "$oid": "000000000000000000000001"
      },
      "score": 8,
      "date": "2023-10-30",
      "author": "Yusoreqa"
    },
    {
      "reviewId": {
        "$oid": "000000000000000000000002"
      },
      "score": 4,
      "date": "2011-08-07",
      "author": "Hojosu"
    }
  ],
  "reviewCount": 2,
  "Top3ReviewsByLikes": [
    {
      "_id": {
        "$oid": "000000000000000000000002"
      },
      "game": "(Almost) Total Mayhem",
      "quote": "Clunky and unresponsive controls that hindereb my enjoyment. Fairly craftec game world.",
      "author": "Hojosu",
      "date": "2011-08-07",
      "score": 4,
      "likes": 22
    },
    {
      "_id": {
        "$oid": "000000000000000000000001"
      },
      "game": "(Almost) Total Mayhem",
      "quote": "stunning graphhcs!",
      "author": "Yusoreqa",
      "date": "2023-10-30",
      "score": 8,
      "likes": 18
    }
  ]
}
```

Review

```
{
  "_id": {
    "$oid": "000000000000000000000001"
  },
  "game": "(Almost) Total Mayhem",
  "score": 8,
  "quote": "stunning graphhcs!",
  "author": "Yusoreqa",
  "date": "2023-10-30",
  "likes": 18
}
```

Comment

```
{
  "_id": {
    "$oid": "000000000000000000000001"
  },
  "reviewId": {
    "$oid": "000000000000000000000001"
  },
  "author": "Hojosu",
  "quote": "I see that Yusoreqa agrees with me.",
  "date": "2023-12-06"
}
```

User

```
{
  "_id": {
    "$oid": "000000000000000000000008"
  },
  "username": "Yidazareha",
  "email": "Yidazareha@outlook.com",
  "password_hash": "FHAklujAlFE51nr4RcG0b24FvfzwxN0PXSHMC04zKGo=",
  "Top3ReviewsByLikes": [
    {
      "_id": {
        "$oid": "00000000000000000002a00e"
      },
      "game": "Rocksmith: All-new 2014 Edition - Sublime: Badfish",
      "quote": "Seamless online connectivity for a smooth multiplayer
         experience.",
      "author": "Yidazareha",
      "date": "2017-04-09",
      "score": 8,
      "likes": 8
    }
  ]
}
```

Company

```
{
  "_id": {
    "$oid": "000000000000000000000006"
  },
  "imglink": "https://cdn.mobygames.com/015cde6c-bc74-11ed-bde2-02420
     a000179.webp",
  "Name": "10tacle studios AG",
  "Overview": "10tacle studios AG was a German game publisher founded by
       CEO Michele Pes in August 2003 ...",
  "Top3Games": [
    {
      "Name": "Boulder Dash Rocks!",
      "Description": "...",
      "img": "https://cdn.mobygames.com/72f87b4e-abee-11ed-80b1-02420
         a000133.webp",
      "user_review": 10
    },
    {
      "Name": "GTR: FIA GT Racing Game",
      "Description": "...",
      "img": "https://cdn.mobygames.com/46e57b30-abab-11ed-9201-02420
         a00019c.webp",
      "user_review": 10
    },
    {
      "Name": "Neocron 2: Beyond Dome of York",
      "Description": "...",
      "img": "https://cdn.mobygames.com/be80ef04-abaf-11ed-aecf-02420
         a000198.webp",
      "user_review": 9
    }
  ]
}
```

User Image

```
{
  "_id": {
    "$oid": "000000000000000000000001"
  },
  "username": "0",
  "image": "iVBORw0KGgoAAAANSUhEUgAAAQAAAAEACAIAAADTE..."
}
```

4.2 Graph Database - Neo4j

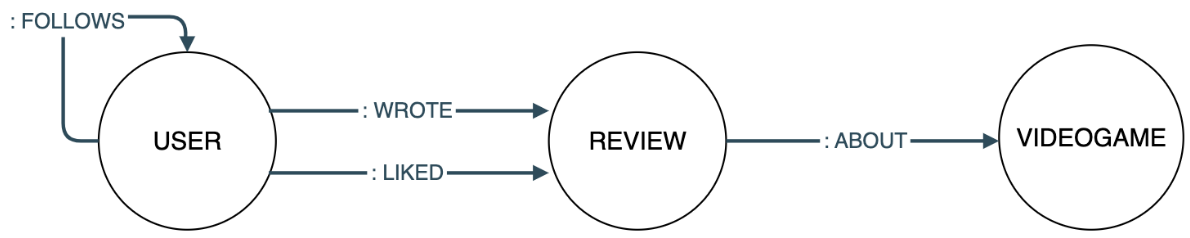

Figure 4.1: Entities handled by *GraphDB* and their relationships

We handled the following entities via GraphDB (figure 4.1) :

- User
- Videogame
- Review

At the very base of our project's idea we find a social networking system. The graph database is exploited to keep track of the users within our network and mainly to manage and make the most out of their connections and relationships. The graph database's architecture we benefit from allow us to make efficient querying of relationships between

users, to perform analytics and subsequent actions based on connection findings. We decided to have a simplified representation of all the entities involved (users, reviews and videogames), that are stored in detail on a document database. Having the three entities over both the two different types of database we need to deal with consistency between MongoDB and Neo4j (we'll address this problem in the next section). The idea is to use this specific type of architecture to identify which users share the same taste in games, and then use this information to suggest new connections.
The system will recommend two entities to a logged in user: users to follow and games to play. The first depends on the games that have being reviewed by the user, and the second depends on the users already followed. More details on the implementation will be provided later.

4.2.1 Nodes within the graph

We have three types of nodes:

- *User* : simplified version of the MongoDB collection *users* that for each user only keeps track of the `username`, that we remember has to be unique.
- *Review* : simplified version of the MongoDB collection *reviews*. For each review we have the `reviewID` (that uniquely identifies each review into the database) and the related `score`.
- *Videogame* : simplified version of the MongoDB collection *videogames*. We keep only the videogames' unique `name` attribute, that is enough for the queries we need to perform.

we added these redundancies (all the information in Neo4j is a subset of the overall data stored in MongoDB) to have at our disposal solely the information needed to perform the queries we decided to implement using the strenghts of the graph database's architecture.

4.2.2 Relationships between nodes of the graph

Here we look at the type of relationships within the graph:

- *Follows* : if the user `A` follows another user `B` there is a `:FOLLOWS` relationship from `A` to `B`

$$\mathtt{A} \xrightarrow{\mathtt{:FOLLOWS}} \mathtt{B}$$

- *Wrote* : if the user `U` writes a review `R`, we create a `:WROTE` relationship from `U` to `R`

$$\mathtt{U} \xrightarrow{\mathtt{:WROTE}} \mathtt{R}$$

- *Liked* : if the user U likes a review R we keep track of this by generating a :LIKED relationship from U to R

$$\mathtt{U} \xrightarrow{\mathtt{:LIKED}} \mathtt{R}$$

- *About* : if a review R was written for the specific videogame V we need to create an :ABOUT relationship from R to V

$$\mathtt{R} \xrightarrow{\mathtt{:ABOUT}} \mathtt{V}$$

4.3 Distributed Database Design

4.3.1 Replicas

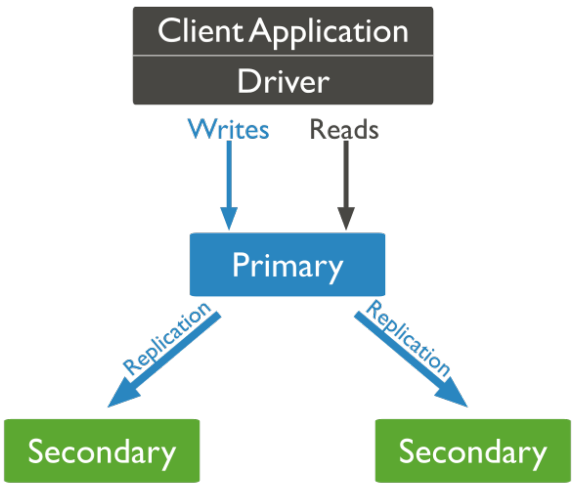

Figure 4.2: Replica Set

We were given the chance to exploit a cluster of three nodes for the project. We deployed one *MongoDB replica* for each node (replicas were not implemented for *Neo4j* since to do so we would have needed the *Enterprise edition*). The *primary* is the only member in the replica set that receives write operations (figure 4.2). MongoDB applies write operations on the primary and then records the operations on the primary's *oplog*. *Secondary* members replicate this log and apply the operations to their data sets. All members of the replica set can accept read operations. By default, however, an application directs its read operations to the primary node.

Consistency Level

Consistency Level refers to the consistency between copies of data on different replicas. We consider data to be *strictly consistent* if all replicas have the same data. Enforcing strong consistency, anyway, would be too costly and - even before that - unnecessary. Eventual consistency is sufficient : we'll have a *write concern* (i.e. the level of acknowledgment requested from MongoDB for write operations on the *Replica set*) with the `w` option set to 1, requesting acknowledgment that each write operation has been propagated to the primary in the replica set. Data can be rolled back if the primary steps down before the write operations have replicated to any of the secondaries (`w: 1` is specified into the `spring.data.mongodb.uri` placed in `application.properties`, under the `resources` directory of the project, where we also specify the *read concern* `readConcernLevel=local`, i.e. `r: 1`). Early results of eventual consistency data queries might not have the most recent updates, it will take time for updates to reach replicas across the database cluster. The consistency level has to be set according to several requirements. Returning the latest data for every query would result in high latency and delay, two aspects we need to be really careful about. We always need to keep in mind that we are working on a social networking system, in which engagement with the user has to be given the top one priority. By exploiting *eventual consistency* results will be less consistent early on, but they will be provided much faster, with low latency. Facing inconsistency does not really represent a problem for the purposes of the application.

4.3.2 Sharding

Heavy loads on a server, than in the specific case of our application may be due to a large number of users in our network, can be dealt with through a process of horizontal partitioning. Horizontal partitioning of a document database is often referred to as *sharding* and it consists in dividing a database by documents into different sections (known as *shards*) stored on separate servers. This process could be exploited to enable our document database to scale in order to meet a possibly growing demand for our application. Each server within the document database cluster will have only *one* shard per server, so if our database is configured to replicate data, than a single shard will be stored on multiple servers. We'll consider sharding only for the Document side of our database, since the GraphDB architecture presents challenges for distributing the graph across multiple servers: while a node can, of course, point to an adjacent node located on another machine, the overhead of routing the traversal across multiple machines eliminates the advantages of the graph database model, because inter-server communication is far more time consuming than local access.
To implement sharding on the document side, we need to select a *shard key* and a *partitioning method.* The shard key specifies the values to use when grouping documents into different shards, so it is represented by one or more fields that need to exist in *all* documents in a collection, and that should be chosen accordingly to the specific requests we expected to get from our clients, in order to optimize the response of the application.

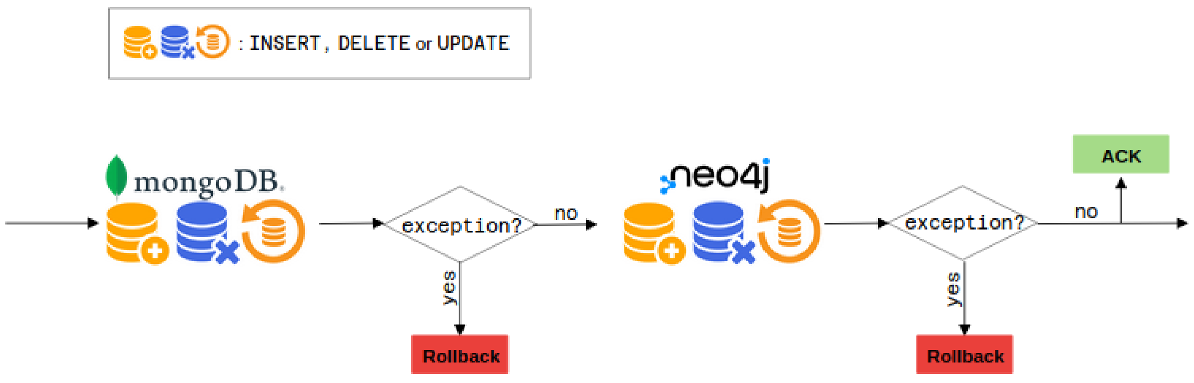

Figure 4.3: Handling consistency between *MongoDB* and *Neo4j*

We can implement a hash-based sharding for Reviews and Comments, using their *IDs* as sharding keys. In this perspective we would use hashing as a partitioning algorithm, which would be helpful to evenly balance the load on different servers by applying an hash function to the shard keys: with hash-based partitioning new documents are distributed evenly across all members of the cluster. Considering the possibility of having to deal with a larger and larger amount of *Videogames*, on the other hand, we could shard the videogames' collection using as *sharding key* the `release_date` - an attribute that every *videogame* has to have (which actually makes it exploitable to implement the *key*). This particular type of sharding could significantly optimize some of the queries we often need to perform over the database. We can look specifically at the one implemented in order to return and show to the final users the *newest* videogames, that was an operation specifically thought to advertise and keep the users updated on the latest releases, offering *Company Managers* a showcase on which to advertise their new products. Whenever range partitioning is enabled and the shard key is continuously incremented, as it is in our case considering the *release date*, we need to be careful, each time a new videogame is added to the database, to the fact that the load will tend to aggregate against only one of the shards, possibly unbalancing the cluster. However, we *do not* consider the addition of new videogames to be so frequent that it could become critical, leading to problems.

4.3.3 Handling inter-databases consistency

Having to deal with two different architectures (*DocumentDB* and *GraphDB*), we need to face the problem of redundancy and handle the consistency of data that are stored within both databases. Any rollback triggered by a failure of an insert, an update or a delete operation, can lead to inconsistencies. To address this problem, the system will start write operations on *Neo4J* only when the same operation is performed successfully on *MongoDB*. Then, if any exception occurs when the updates over data on *Neo4j* are attempted, then a rollback operation will be executed in order to bring back both the databases in a consistent state (just as shown in figure 4.3). All of this is made possible by the annotation *@Transactional*, made available by Spring Data.

High Availability, Eventual Consistency

Eventual consistency is dealt with by exploiting the *asynchronous execution support* in *Spring* and the *@Async* annotation. The caller will not have to wait for the complete execution of the called method: annotating a method of a bean with *@Async* will make it execute in a separate thread, thus increasing the availability of our application.

Chapter 5

Implementation

5.1 Front-End

Our application's front end is implemented as a typical web app, with a combination of Javascript, HTML and CSS. Data is retrieved from the server through AJAX requests.

5.2 Back-End

The back-end is implemented using Java Spring Boot. Requests sent by the clients are resolved by a series of repositories which utilize Spring Data methods to handle database data. The server can dynamically set HTML content thanks to the Thymeleaf template engine.

5.3 Spring Boot

5.3.1 Spring Data Neo4j

Spring Data Neo4j provides repository support for the Neo4j graph database. It eases development of applications with a consistent programming model that need to access Neo4j data sources.

5.3.2 Spring Data MongoDB

Spring Data for MongoDB is part of the umbrella Spring Data providing integration with the MongoDB document database, offering a familiar and consistent Spring based programming model while retaining store specific features and capabilities.

Features

- *MongoTemplate*: Helper class that increases productivity for common tasks.
- *Data Repositories*: Repository interfaces including support for custom queries and aggregations.

5.4 Java Entities (for MongoDB)

5.4.1 Users

Simple User - Reviewer

```java
@Document("users")
public class User {
    @SuppressWarnings("unused")
    private static final Logger logger = LoggerFactory.getLogger(User.
        class);
    @Id
    public ObjectId id;
    @Field("username")
    public String username;
    @Field("password_hash")
    public String password_hash;
    @Field("email")
    public String email;
    @Field("Top3ReviewsByLikes")
    public List<Review> top_reviews;

    public User(String username, String password_hash, String email,
        List<Review> top_reviews) {
        this.username = username;
        this.password_hash = password_hash;
        this.email = email;
        this.top_reviews = top_reviews;
    }

    public String getCompany_name() {
        return "";
    }
    public User() {
    }

    public User(DBObject dbObject)
    {
        this.id = (ObjectId) dbObject.get("_id");
        this.username = (String) dbObject.get("username");
        this.password_hash = (String) dbObject.get("password_hash");
        this.email = (String) dbObject.get("email");
        @SuppressWarnings("unchecked")
        List<org.bson.Document> top_review = (List<org.bson.Document>)
           dbObject.get("Top3ReviewsByLikes");
        this.top_reviews = top_review.stream().map(Review::new).toList()
           ;
    }

    public static User userFactory(DBObject dbObject)
    {
        if(dbObject.get("company_name") != null)
            return new CompanyManager(dbObject);
        else if(dbObject.get("is_admin") != null && (boolean)dbObject.
           get("is_admin"))
            return new Admin(dbObject);
        else
            return new User(dbObject);
    }

    public String getAccountType() {
        return "User";
    }
    public String getUsername() {
        return this.username;
    }

    private String encodePassword(String password) throws
       NoSuchAlgorithmException {
        MessageDigest md = MessageDigest.getInstance("SHA-256");
        md.update(password.getBytes());
        byte[] digest = md.digest();
        Base64.Encoder encoder = Base64.getEncoder();
        return encoder.encodeToString(digest);
    }

    public void setPasswordHash(String password) throws
       NoSuchAlgorithmException{
        this.password_hash = encodePassword(password);
    }

    public boolean checkPassword(String password) throws
       NoSuchAlgorithmException{
        return this.password_hash.equals(encodePassword(password));
    }
}
```

Company Manager

```
public class CompanyManager extends User {
    @Field("company_name")
    public String company_name;
    @Override
    public String getAccountType() {
        return "Company";
    }

    public CompanyManager() {

    }

    @Override
    public String getUsername() {
        return super.getUsername();
    }

    @Override
    public String getCompany_name() {
        return company_name;
    }

    public CompanyManager(DBObject dbObject) {
        super(dbObject);
        this.company_name = (String) dbObject.get("company_name");
    }
}
```

Admin

```
public class Admin extends User {
    @Override
    public String getAccountType() {
        return "Admin";
    }

    public Admin() {

    }
```

```java

    public Admin(DBObject dbObject) {
        super(dbObject);
    }
}
```

5.4.2 Videogames

Documents inside the collection ”videogames” can have different attributes from each other, and some may have attributes that others don’t. To handle this, we decided to implement a Map object that holds all of the attributes of a videogame, whichever and however many they may be.

```java
@Document(collection = "videogames")
public class Game {
    private static final Logger logger = LoggerFactory.getLogger(Game.
       class);
    @Id
    public ObjectId id;
    @Field("Name")
    public String name;
    @Field("Released")
    public String released;
    @Field("Top3ReviewsByLikes")
    public List<Review> top_reviews;


    @Field("customAttributes")
    public Map<String, Object> customAttributes = new HashMap<>();

    // other fields, getters, setters

    public Map<String, Object> getCustomAttributes() {
        return customAttributes;
    }

    public void setCustomAttributes(Map<String, Object> customAttributes
       ) {
        this.customAttributes = customAttributes;
    }

    public void setCustomAttributes(DBObject db) {
        try {
            @SuppressWarnings("unchecked")
            Map<String,Object> map = new ObjectMapper().readValue(db.toString(), HashMap.class);
            customAttributes = map;
        }
        catch (Exception e){
            logger.error("Error while setting custom attributes: " + e.getMessage());
        }
    }

    public Game (DBObject db){
        try {
            @SuppressWarnings("unchecked")
            Map<String, Object> map = new ObjectMapper().readValue(db.toString(), HashMap.class);
            customAttributes = map;
        } catch (Exception e) {
            logger.error("Error while setting custom attributes: " + e.getMessage());
        }
        this.name = customAttributes.get("Name").toString();
        @SuppressWarnings("unchecked")
        List<org.bson.Document> reviews_object = (List<org.bson.Document>)db.get("Top3ReviewsByLikes");
        if (reviews_object != null){
            this.top_reviews = reviews_object.stream().map(Review::new).toList();
        }
        else this.top_reviews = new ArrayList<>();
        @SuppressWarnings("unchecked")
        HashMap<String,Object> released = (HashMap<String,Object>)customAttributes.get("Released");
        this.released = released.get("Release Date").toString();
    }

    public Game(String st) {
        try {
            @SuppressWarnings("unchecked")
            Map<String, Object> map = new ObjectMapper().readValue(st, HashMap.class);
            customAttributes = map;
        } catch (Exception e) {
            logger.error("Error while setting custom attributes: " + e.getMessage());
        }
        this.name = customAttributes.get("Name").toString();

        @SuppressWarnings("unchecked")
        HashMap<String, Object> released = (HashMap<String, Object>)
            customAttributes.get("Released");
        this.released = released.get("Release Date").toString();
    }

    public static org.bson.Document documentFromJson(String json) throws
         IllegalArgumentException {
        org.bson.Document doc = org.bson.Document.parse(json);
        if (doc == null)
        {
            throw new IllegalArgumentException("Invalid JSON");
        }
        if (doc.containsKey("_id")) {
            doc.remove("_id");
        }
        if (!doc.containsKey("Name")) {
            throw new IllegalArgumentException("Name is a required field
                ");
        }
        if (!doc.containsKey("Developers")) {
            doc.put("Developers", List.of());
        }
        if (!doc.containsKey("Publishers")) {
            doc.put("Publishers", List.of());
        }
        doc.put("reviews", List.of());
        doc.put("user_review",null);
        doc.put("reviewCount",0);
        doc.put("Top3ReviewsByLikes", List.of());
        return doc;
    }
```

5.4.3 Review

```java
public class Review {
    @Id
    public ObjectId id;
    public String game;
    public Integer score;
    public String quote;
    public String author;
    public String date;

    public Review() {
    }

    /**
     *  Set date to null if you want to use the current date
     **/
    public Review(String game, Integer score, String quote, String
       author, String date)
    {
        this.game = game;
        this.score = score;
        this.quote = quote;
        this.author = author;
        if (date == null)
        {
            Date d = Calendar.getInstance().getTime();
            DateFormat df = new SimpleDateFormat("yyyy-MM-dd");
            this.date = df.format(d);
        }
        else
        {
            this.date = date;
        }
    }

    public Review(DBObject dbObject)
    {
        this.id = (ObjectId) dbObject.get("_id");
        this.game = (String) dbObject.get("game");
        this.score = (Integer) dbObject.get("score");
        this.quote = (String) dbObject.get("quote");
        this.author = (String) dbObject.get("author");
        this.date = (String) dbObject.get("date");
    }

    public Review(Document document)
    {
        this.id = (ObjectId) document.get("_id");
        this.game = (String) document.get("game");
        this.score = (Integer) document.get("score");
        this.quote = (String) document.get("quote");
        this.author = (String) document.get("author");
        this.date = (String) document.get("date");
    }

    public String getId() {
        return id.toHexString();
    }
}
```

5.4.4 Comment

```
public class Comment {
    @Id
    public ObjectId id;
    public ObjectId reviewId;
    public String author;
    public String quote;
    public String date;

    public Comment() {}

    /**
     * Set date to null if you want to use the current date
     */
    public Comment(String reivew_id, String author, String quote, String
        date) {
        this.reviewId = new ObjectId(reivew_id);
        this.author = author;
        this.quote = quote;
        if (date == null)
        {
            Date d = Calendar.getInstance().getTime();
            DateFormat df = new SimpleDateFormat("yyyy-MM-dd");
            this.date = df.format(d);
        }
        else
        {
            this.date = date;
        }
    }

    public String getId() {
        return id.toHexString();
    }

    public String getReviewId() {
        return reviewId.toHexString();
    }
}
```

5.5 Java Entities (for Neo4J)

5.5.1 Users

```java
@Node("User")
public class UserDTO {
    @Id
    @GeneratedValue
    public UUID id;
    @Property("username")
    public String username;
    public  UserDTO(String username){
        this.username = username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
}
```

5.5.2 Review

```java
@Node("Review")
public class ReviewDTO {

    @Id
    @GeneratedValue
    public UUID id;
    @Property("reviewId")
    public String reviewId;
    @Property("score")
    public String score;
    @Property("likeCount")
    public Integer likeCount;

    public ReviewDTO(UUID id, String score, String reviewId, Integer
        likeCount) {
        this.id = id;
        this.reviewId = reviewId;
        this.score = score;
        this.likeCount = likeCount;
    }

    public void setId(UUID id) {
        this.id = id;
    }

    public void setScore(String score) {
        this.score = score;
    }

    public void setReviewId(String reviewId) {
        this.reviewId = reviewId;
    }
}
```

5.5.3 Game

```
@Node("Game")
public class GameDTO {
    @Id
    @GeneratedValue
    public UUID id;
    @Property("name")
    public String name;

    public GameDTO(String name){
        this.name = name;
    }

    public void setName(String name){
        this.name = name;
    }
}
```

Chapter 6

Queries

In this chapter we illustrate all of the non-trivial queries that are run by our application.

6.1 Mongo DB

6.1.1 Search

This query is run every time the user inserts a character in the search bar. It finds every user, game and company that have in their name all of the characters typed. It works in the same way for all three elements; the only difference is the collection and the attribute name the query is ran with. Below is reported the game version of the query as an example.

```
db.videogames.find(
   { "Name": { $regex: /escapedQuery/i } }
).sort(
   { "reviewCount": -1 }
).limit(10)
```

```
/*gamecritic/repositories/Game/CustomGameRepositoryImpl.java*/
public List<Game> search(String query){
        String escapedQuery = Pattern.quote(query);
        if (escapedQuery == null) {
            throw new IllegalArgumentException("The given query must not
                be null");
        }
        Criteria criteria = Criteria.where("Name").regex(escapedQuery, "
           i");
        Query q = new Query(criteria).limit(10).with(Sort.by(Sort.Order.
           desc("reviewCount")));

        List<DBObject> game_objects = mongoTemplate.find(q, DBObject.
            class, "videogames");
        return game_objects.stream().map(Game::new).toList();
    }
```

6.1.2 Users

Get top Users from Reviews

This query finds the top 10 users that wrote the most reviews in the last X months, with X being variable. It is executed whenever an administrator loads the Stats page.

```
db.reviews.aggregate([
    { $match: { "date": { $regex: regex } } },
    { $group: { _id: "$author", reviews: { $sum: 1 } } },
    { $sort: { reviews: -1 } },
    { $limit: 10 }
])
```

The regex indicates all dates that are included in the last X months.

```
/*gamecritic/repositories/User/CustomUserRepositoryImpl.java*/
public List<TopUserDTO> topUsersByReviews(Integer months) {
    if (months == null || months < 1) {
        throw new IllegalArgumentException("The given months must not be
            null nor less than 1");
    }
    Calendar d = Calendar.getInstance();
    Integer this_year = d.get(Calendar.YEAR);
    Integer this_month = d.get(Calendar.MONTH) + 1;
    String regex = "^(";
    for(int i = 0; i < months; i++)
    {
        Integer month = this_month - i;
        Integer year = this_year;
        if(month <= 0)
        {
            month += 12;
            year -= 1;
        }
        regex += year.toString() + "-" + String.format("%02d", month);
        if(i != months - 1)
        {
            regex += "|";
        }
    }
    regex += ")";

    Aggregation aggregation = Aggregation.newAggregation(
    Aggregation.match(Criteria.where("date").regex(regex)),
    Aggregation.group("author").count().as("reviews"),
    Aggregation.sort(Sort.Direction.DESC, "reviews"),
    Aggregation.limit(10)
    );
    List<DBObject> user_dbos = mongoTemplate.aggregate(aggregation, "
       reviews", DBObject.class).getMappedResults();
    List<TopUserDTO> users = user_dbos.stream().map(user -> new
       TopUserDTO(user.get("_id").toString(), (Integer)user.get("reviews
       "))).toList();
    return users;
}
```

Get top 3 Users' Reviews from Likes

This query finds the top 3 reviews by number of likes for each user, and updates their respective field Top3ReviewsByLikes. This is an expensive operation, thus we only execute it periodically (daily) and whenever an administrator requires it.

```
db.reviews.aggregate([
  {
    $group: {
      _id: "$author",
      Top3ReviewsByLikes: {
        $push: {
          _id: "$_id",
          game: "$game",
          quote: "$quote",
          author: "$author",
          date: "$date",
          score: "$score",
          likes: "$likes"
        }
      }
    }
  },
  {
    $set: {
      username: "$_id"
    }
  },
  {
    $project: {
      _id: 0,
      username: 1,
      Top3ReviewsByLikes: {
        $slice: ["$Top3ReviewsByLikes", 3]
      }
    }
  },
  {
    $merge: {
      into: "users",
      on: "username",
      whenMatched: "merge",
      whenNotMatched: "discard"
    }
  }
]);
```

```
/*gamecritic/repositories/User/CustomUserRepositoryImpl.java*/
public void updateTop3ReviewsByLikes() {
    Aggregation aggregation = Aggregation.newAggregation(
    Aggregation.stage(
    "{\n" + //
        "  $group: {\n" + //
            "    _id: \"$author\",\n" + //
            "    Top3ReviewsByLikes: {\n" + //
                "      $topN: {\n" + //
                    "        n: 3,\n" + //
                    "        output: {\n" + //
                        "          _id: \"$_id\",\n" + //
                        "          game: \"$game\",\n" + //
                        "          quote: \"$quote\",\n" + //
                        "          author: \"$author\",\n" + //
                        "          date: \"$date\",\n" + //
                        "          score: \"$score\",\n" + //
                        "          likes: \"$likes\",\n" + //
                        "        },\n" + //
                    "        sortBy: {\n" + //
                        "          likes: -1,\n" + //
                        "        },\n" + //
                    "      },\n" + //
                "    },\n" + //
            "  },\n" + //
        "}"
    ),
    Aggregation.stage(
    "{\n" + //
        "  $set: {\n" + //
            "    username: \"$_id\",\n" + //
            "  },\n" + //
        "}"
    ),
    Aggregation.stage(
    "{\n" + //
        "  $project: {\n" + //
            "    _id: 0,\n" + //
            "    username: 1,\n" + //
            "    Top3ReviewsByLikes: 1,\n" + //
            "  },\n" + //
        "}"
    ),
    Aggregation.stage(
    "{\n" + //
        "  $merge: {\n" + //
            "    into: \"users\",\n" + //
            "    on: \"username\",\n" + //
            "    whenMatched: \"merge\",\n" + //
            "    whenNotMatched: \"discard\",\n" + //
            "  },\n" + //
        "}"
    )
    ).withOptions(Aggregation.newAggregationOptions().allowDiskUse(true)
       .build());

    Instant start = Instant.now();
    mongoTemplate.aggregate(aggregation, "reviews", DBObject.class).
       getMappedResults();
    logger.info("Finished updating top 3 reviews by likes for all users
       in " + (Instant.now().toEpochMilli() - start.toEpochMilli()) + "
       ms");
}
}
```

6.1.3 Videogames

Find Hottest Games

This query finds the games that have received the most reviews in the last 6 months, 10 at a time. It is executed whenever the Hottest page is loaded. The offset depends on how many games have already been loaded in the page.

```
db.videogames.aggregate([
  {
    $match: {
      "reviews.date": {
        $gte: 6MonthsAgo,
      },
    },
  },
  {
    $group: {
      _id: "$_id",
      Name: {
        $first: "$Name",
      },
      HotReviewCount: {
        $sum: {
          $size: {
            $filter: {
              input: "$reviews",
              as: "review",
              cond: {
                $gte: [
                  "$$review.date",
                  6MonthsAgo,
                ],
              },
            },
          },
        },
      },
      allAttributes: {
        $mergeObjects: "$$ROOT",
      },
    },
  },
  {
    $sort: {
      HotReviewCount: -1,
      Name: 1,
    },
  },
  {
    $skip: offset,
  },
  {
    $limit: 10,
  },
  {
    $replaceRoot: {
      newRoot: {
        $mergeObjects: ["$allAttributes"],
      },
    },
  },
]);
```

```java
/*gamecritic/repositories/Game/CustomGameRepositoryImpl.java*/
    public List<Game> findHottest(Integer offset) {
        LocalDate currentDate = LocalDate.now();
        LocalDate ago;
        ago = currentDate.minusMonths(6);
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-
           MM-dd");
        String formattedDate = ago.format(formatter);
        Aggregation a = Aggregation.newAggregation(
                Aggregation.stage("{\n" +
                "    $match: {\n" +
                "      \"reviews.date\": {\n" +
                "        $gte: \""+formattedDate+"\",\n" +
                "      },\n" +
                "    },\n" +
                "  }"),
                Aggregation.stage("{\n" +
                        "    $group: {\n" +
                        "      _id: \"$_id\",\n" +
                        "      Name: {\n" +
                        "        $first: \"$Name\",\n" +
                        "      },\n" +
                        "      HotReviewCount: {\n" +
                        "        $sum: {\n" +
                        "          $size: {\n" +
                        "            $filter: {\n" +
                        "              input: \"$reviews\",\n" +
                        "              as: \"review\",\n" +
                        "              cond: {\n" +
                        "                $gte: [\n" +
                        "                  \"$$review.date\",\n" +
                        "                  \""+formattedDate+"\",\n" +
                        "                ],\n" +
                        "              },\n" +
                        "            },\n" +
                        "          },\n" +
                        "        },\n" +
                        "      },\n" +
                        "      allAttributes: {\n" +
                        "        $mergeObjects: \"$$ROOT\",\n" +
                        "      },\n" +
                        "    },\n" +
                        "  }"),
                Aggregation.stage("{\n" +
                        "    $sort:\n" +
                        "      {\n" +
                        "        HotReviewCount: -1,\n" +
                        "        Name: 1             \n"+
                        "      },\n" +
                        "  }"),
                Aggregation.stage("{" +
                        "$skip: " + offset + " }"),
                Aggregation.stage("{\n" +
                        "    $limit:\n" +
                        "      10,\n" +
                        "  }"),
                Aggregation.stage("{\n" +
                        "    $replaceRoot: {\n" +
                        "      newRoot: {\n" +
                        "        $mergeObjects: [\n" +
                        "          \"$allAttributes\",\n" +
                        "          \n" +
                        "        ],\n" +
                        "      },\n" +
                        "    },\n" +
                        "  }")
        );
        List<DBObject> game_objects = mongoTemplate.aggregate(a, "videogames", DBObject.class).getMappedResults();
        return game_objects.stream().map(Game::new).toList();
    }
```

Find Newest Games

This query orders games from newest to oldest[1], and returns 10 games at a time. As before, the offset depends on how many games have already been loaded in the page. It is executed whenever the Newest Page is loaded.

[1]Some videogames we scraped from MobyGames had no Release Date attribute, hence for those documents we set it to a default value ”Undated”.

```
db.videogames.find({
  "Released.Release Date": { $ne: "Undated" }
})
.sort({ "Released.Release Date": -1 })
.skip(offset)
.limit(10)
```

```java
/*gamecritic/repositories/Game/CustomGameRepositoryImpl.java*/
public List<Game> findLatest(Integer offset){
    Query query = new Query();
    query.addCriteria(Criteria.where("Released.Release Date").ne("
       Undated"));
    query.with(Sort.by(Sort.Order.desc("Released.Release Date"))).skip(
       offset).limit(10);
    List<DBObject> game_objects = mongoTemplate.find(query, DBObject.
       class, "videogames");
    return game_objects.stream().map(Game::new).toList();
}
```

Find Best Games

This query orders the games by best average score and most reviews, and again returns 10 games at a time, with the offset depending on how many games are already being shown in the page. It is executed whenever the Best page is loaded.

```
db.collection.find()
  .sort({ "user_review": -1, "reviewCount": -1 })
  .skip(offset)
  .limit(10)
```

```java
/*gamecritic/repositories/Game/CustomGameRepositoryImpl.java*/
public List<Game> findBest(Integer offset) {
    Query query = new Query();
    query.with(Sort.by(Sort.Order.desc("user_review"), Sort.Order.desc("
       reviewCount"))).skip(offset).limit(10);
    List<DBObject> game_objects = mongoTemplate.find(query, DBObject.
       class, "videogames");
    return game_objects.stream().map(Game::new).toList();
}
```

Get top Games from Average Score

This query finds the top games by average score, in the last X months, of a certain company. It is executed every time a Company manager accesses the Stats page for his company.

```
db.collection.aggregate([
  {
    $match: {
      $or: [
        { Developers: companyName },
        { Publishers: companyName },
        { Developers: { $elemMatch: { $eq: companyName } } },
        { Publishers: { $elemMatch: { $eq: companyName } } }
      ]
    }
  },
  { $unwind: "$reviews" },
  {
    $match: {
      "reviews.date": { $regex: regex },
      "reviews.score": { $exists: true }
    }
  },
  {
    $group: {
      _id: "$Name",
      averageScore: { $avg: "$reviews.score" },
      reviewCount: { $sum: 1 },
      img: { $first: "$img" }
    }
  },
  {
    $sort: {
      averageScore: -1,
      reviewCount: -1
    }
  },
  { $limit: limit }
]);
```

Again, the regex indicates every date in the last X months.

```java
/*gamecritic/repositories/Game/CustomGameRepositoryImpl.java*/
public List<TopGameDTO> topGamesByAverageScore(Integer months, String
    companyName, Integer limit) {
     if (months == null || months < 1) {
         throw new IllegalArgumentException("The given months must not be
             null nor less than 1");
     }
     Calendar d = Calendar.getInstance();
     Integer this_year = d.get(Calendar.YEAR);
     Integer this_month = d.get(Calendar.MONTH) + 1;
    String regex = "^(";
    for(int i = 0; i < months; i++)
    {
        Integer month = this_month - i;
        Integer year = this_year;
        if(month <= 0)
        {
            month += 12;
            year -= 1;
        }
        regex += year.toString() + "-" + String.format("%02d", month);
        if(i != months - 1)
        {
            regex += "|";
        }
    }
    regex += ")";

    Aggregation aggregation = Aggregation.newAggregation(
    Aggregation.match(new Criteria().orOperator(
    Criteria.where("Developers").is(companyName),
    Criteria.where("Publishers").is(companyName),
    Criteria.where("Developers").elemMatch(new Criteria().is(companyName
        )),
    Criteria.where("Publishers").elemMatch(new Criteria().is(companyName
        ))
    )),
    Aggregation.unwind("reviews"),
    Aggregation.match(new Criteria().andOperator(
    Criteria.where("reviews.date").regex(regex),
    Criteria.where("reviews.score").exists(true)
    )),
    Aggregation.group("Name").avg("reviews.score").as("averageScore").
        count().as("reviewCount").first("img").as("img"),
    Aggregation.sort(Sort.Direction.DESC, "averageScore").and(Sort.
        Direction.DESC, "reviewCount"),
    Aggregation.limit(limit)
    );
    List<DBObject> games_dbos = mongoTemplate.aggregate(aggregation, "
        videogames", DBObject.class).getMappedResults();
    List<TopGameDTO> games = games_dbos.stream().map(game -> new
        TopGameDTO(game.get("_id").toString(), ((Double)game.get("
        averageScore")).floatValue(), game.get("img") != null?game.get("
        img").toString():null)).toList();
    return games;
}
```

Get top 3 Videogames' Reviews from Likes

This query finds the top 3 reviews of a game by number of likes received, and embeds the results in the Top3ReviewsByLikes attribute of the respective videogame document. This is another expensive query, so we also run this periodically (daily) and whenever an administrator asks.

```
db.reviews.aggregate([
  {
    $group: {
      _id: "$game",
      Top3ReviewsByLikes: {
        $topN:{
          n:3,
          output: {
            _id: "$_id",
            game: "$game",
            quote: "$quote",
            author: "$author",
            date: "$date",
            score: "$score",
            likes: "$likes",
          },
                sortBy:{
                        likes:-1
          }
              }
      }
    }
  },
  {
    $set: {
      Name: "$_id"
    }
  },
  {
    $project: {
      _id: 0,
      Name: 1,
      Top3ReviewsByLikes: 1
    }
  },
  {
    $merge: {
      into: "videogames",
      on: "Name",
      whenMatched: "merge",
      whenNotMatched: "discard"
    }
  }
]);
```

```
/*gamecritic/repositories/Game/CustomGameRepositoryImpl.java*/
public void updateTop3ReviewsByLikes()
{
    Aggregation aggregation = Aggregation.newAggregation(
    Aggregation.stage(
    "{\n" + //
        "  $group: {\n" + //
            "    _id: \"$game\",\n" + //
            "    Top3ReviewsByLikes: {\n" + //
                "      $topN: {\n" + //
                    "        n: 3,\n" + //
                    "        output: {\n" + //
                        "          _id: \"$_id\",\n" + //
                        "          game: \"$game\",\n" + //
                        "          quote: \"$quote\",\n" + //
                        "          author: \"$author\",\n" + //
                        "          date: \"$date\",\n" + //
                        "          score: \"$score\",\n" + //
                        "          likes: \"$likes\",\n" + //
                        "        },\n" + //
                    "        sortBy: {\n" + //
                        "          likes: -1,\n" + //
                        "        },\n" + //
                    "      },\n" + //
                "    },\n" + //
            "  },\n" + //
        "}"
    ),
    Aggregation.stage(
    "{\n" + //
        "  $set: {\n" + //
            "    Name: \"$_id\",\n" + //
            "  },\n" + //
        "}"
    ),
    Aggregation.stage(
    "{\n" + //
        "  $project: {\n" + //
            "    _id: 0,\n" + //
            "    Name: 1,\n" + //
            "    Top3ReviewsByLikes: 1,\n" + //
            "  },\n" + //
        "}"
    ),
    Aggregation.stage(
    "{\n" + //
        "  $merge: {\n" + //
            "    into: \"videogames\",\n" + //
            "    on: \"Name\",\n" + //
            "    whenMatched: \"merge\",\n" + //
            "    whenNotMatched: \"discard\",\n" + //
            "  },\n" + //
        "}"
    )
    ).withOptions(Aggregation.newAggregationOptions().allowDiskUse(true)
        .build());

    Instant start = Instant.now();
    mongoTemplate.aggregate(aggregation, "reviews", DBObject.class).
        getMappedResults();
    logger.info("Finished updating top 3 reviews by likes for all games
        in " + (Instant.now().toEpochMilli() - start.toEpochMilli()) + "
        ms");
}
}
```

Company Score Distribution

This query calculates the distribution of the average score for the games of a given company. It is executed every time a Company Manager accesses the Stats page for his company.

```
db.videogames.aggregate([
  {
    $match: {
      $or: [
        {
          Developers: companyName,
        },
        {
          Publishers: companyName,
        },
        {
          Developers: {
            $elemMatch: {
              $eq: companyName,
            },
          },
        },
        {
          Publishers: {
            $elemMatch: {
              $eq: companyName,
            },
          },
        },
      ],
    },
  },
  {
    $unwind: "$reviews",
  },
  {
    $match: {
      "reviews.score": {
        $exists: true,
      },
      "reviews.score": {
        $gt: 0,
      },
    },
  },
  {
    $group: {
      _id: "$reviews.score",
      count: {
        $sum: 1,
      },
    },
  },
  {
    $densify:
      {
        field: "_id",
        range: {
          step: 1,
          bounds: [1, 11],
        },
      },
  },
]);
```

```java
/*gamecritic/repositories/Game/CustomGameRepositoryImpl.java*/
public List<Float> companyScoreDistribution(String companyName)
{
    Aggregation aggregation = Aggregation.newAggregation(
        Aggregation.match(new Criteria().orOperator(
        Criteria.where("Developers").is(companyName),
        Criteria.where("Publishers").is(companyName),
        Criteria.where("Developers").elemMatch(Criteria.where("$eq").is(
            companyName)),
        Criteria.where("Publishers").elemMatch(Criteria.where("$eq").is(
            companyName))
    )),
    Aggregation.unwind("reviews"),
    Aggregation.match(new Criteria().andOperator(
        Criteria.where("reviews.score").exists(true),
        Criteria.where("reviews.score").gt(0)
    )),
    Aggregation.group("reviews.score").count().as("count"),
    DensifyOperation.builder().densify("_id").range(Range.bounded(1, 11)
        .incrementBy(1)).build(),
    Aggregation.sort(Sort.Direction.ASC, "_id")
    );
    List<DBObject> games_dbos = mongoTemplate.aggregate(aggregation, "
        videogames", DBObject.class).getMappedResults();
    List<Float> games = games_dbos.stream().map(game ->
    {
        if(game.get("count") != null)
        {
            return ((Integer)game.get("count")).floatValue();
        }
        else
        {
            return 0f;
        }
    }).toList();
    return games;
}
```

6.2 Neo4j

For these queries, the Cypher text version is immediately visible inside the @Query annotation.

6.2.1 Reviews

Users

Get Followers

```
/*gamecritic/repositories/User/UserRepositoryNeo4J.java*/
@Query(
    "MATCH (u:User {username: $username})\n"+
    "MATCH (u)<-[:FOLLOWS]-(f:User)\n"+
    "RETURN f"
)
List<UserDTO> findFollowers(
    @Param("username") String username
);
```

Get Following

```
/*gamecritic/repositories/User/UserRepositoryNeo4J.java*/
@Query(
    "MATCH (u:User {username: $username})\n"+
    "MATCH (u)-[:FOLLOWS]->(f:User)\n"+
    "RETURN f"
)
List<UserDTO> findFollowed(
    @Param("username") String username
);
```

6.2.2 Suggestions

Games

This query finds at most 4 games to suggest to a user, based on what his followed users have reviewed and ordered by average score and total number of reviews.

```
/*gamecritic/repositories/User/GameRepositoryNeo4J.java*/
@Query(
        "match (u1:User {username: $username})-[:FOLLOWS]->(u2:User)-[:
           WROTE]->(r:Review)-[:ABOUT]->(g:Game)\n"+
        "where not (u1)-[:WROTE]->(:Review)-[:ABOUT]->(g)\n"+
        "with g,r,u2, sum(r.score)/count(u2) as avgScore, count(r) as
           reviewCount\n"+
        "return g\n"+
        "order by avgScore desc, reviewCount desc limit 4"
    )
    List<GameDTO> findSuggestedGames(
        @Param("username") String username
    );@Param("username") String username
);
```

Users

This query finds at most 4 users that have given scores that are the most similar to those of the active user, thus suggesting them as recommended users to follow.

```
/*gamecritic/repositories/User/UserRepositoryNeo4J.java*/
@Query(
        "match (u1:User {username: $username})-[:WROTE]->(r1:Review)-[:
           ABOUT]->(g:Game)\n"+
        "<-[:ABOUT]-(r2:Review)<-[:WROTE]-(u2:User)\n"+
        "where u2.username <> u1.username and not (u1)-[:FOLLOWS]->(u2)\
           n"+
        "with u2, r1, r2, sum(abs(r1.score-r2.score))/count(r1) as
           avgDelta, count(r1) as reviewCount\n"+
        "return u2\n"+
        "order by avgDelta asc, reviewCount desc limit 4"
    )
    List<UserDTO> findSuggestedUsers(
        @Param("username") String username
    );
```

6.3 Inserting and Deleting Elements

What follows are the non-trivial methods for inserting and deleting elements of both the MongoDB and Neo4J databases.

6.3.1 Videogames

Insertion

Whenever a videogame is inserted, we add a new document in the ”videogames” MongoDB collection and a Node in Neo4J.

Deletion

Whenever a videogame is deleted, the following actions are performed (not in this order):

- We delete every review associated with the game in MongoDB and in Neo4J

- We delete every comment associated with every one of those reviews
- We delete the game from the ”videogames” collection in MongoDB
- We delete the game’s Node from Neo4J.

6.3.2 Reviews

Insertion

Whenever a review is inserted, the following steps are performed:

- We add a new document in the ”reviews” collection
- We embed a simplified version inside the videogame document the review is for, in the ”videogames” collection
- We increase the ”reviewCount” attribute in the videogame document by one
- We update the ”user_review” attribute in the videogame document by recalculating the average score
- We add a new review Node in Neo4J, and connect it to the author with a `[:WROTE]` relationship

Deletion

Whenever a review is deleted, the following steps are performed:

- We find all comments associated with that review and delete them from MongoDB
- We find the embedded version of the review inside the game the review is about and delete it
- We update the reviewCount and user_score fields of the game the review is about
- We delete the review from Neo4J
- We delete the review from the ”reviews” collection in MongoDB.

6.3.3 Users

Insertion

Whenever a new User is created, we simply insert the new user document inside the ”users” collection in MongoDB and then we create a new User Node in Neo4J.

Deletion

Whenever a User is deleted, the following steps are performed:

- We delete every comment made by the user
- We delete every review made by the user
- We delete the image associated to the user in the ”user_images” collection
- We delete the user’s document in the ”users” collection
- We delete the user’s Node from Neo4J.

6.3.4 Likes

Insertion

Whenever a ”like” is dropped on a review, the following steps are performed:

- The attribute ”likes” inside the document of the review is increased by one
- A new `[:LIKED]` relationship is created between the user and the review inside Neo4J

Deletion

Whenever a user removes a ”like”, the following steps are performed:

- The attribute ”likes” inside the document of the review is decreased by one
- The `[:LIKED]` relationship between the user and the review inside Neo4J is deleted

6.3.5 Comments

The insertion and deletion of comments is very simple, since it involves only adding or deleting a single document in the ”comments” collection.

6.4 Indexes

6.4.1 MongoDB

We adopted a series of indexes on our MongoDB collections to improve performance. Table 6.1 shows this improvement for some of the most frequently executed queries.

MongoDB				
Query	**Collection**	**Index Keys**	**Documents examined**	**ms (avg)**
`find("Name":name)`	videogames	- `Name`	69553 1	38 *ms* 0 *ms*
`find("author":author)`	reviews	- `author`	250981 1	92 *ms* 3 *ms*
`find("game":game)`	reviews	- `game`	250981 5 (avg)	92 *ms* 3 *ms*
`find("reviewId":reviewId)`	comments	- `game`	625435 3 (avg)	681 *ms* 0 *ms*
`find("Name":  Name)`	companies	- `reviewId`	6556 1	10 *ms* 1 *ms*
`find("username":  username)`	users	- `username`	170594 1	80 *ms* 1 *ms*
`find("username":  username)`	userImages	- `username`	170594 1	78 *ms* 1 *ms*

Table 6.1

6.4.2 Neo4j

For Neo4J we implemented an index for each of our Nodes, on their only attribute. While the improvement for read queries wasn't very noticeable, the improvement for write queries was substantial. The initial setup of the database initially took up to an hour to complete, but after adding indexes, it only took a few seconds.

Chapter 7

User Manual

7.1 Index Page

The index page is the home page of the application. At the top there is a search bar, that can be used to find users, games, or companies. On the top right there are the buttons for login and signup. Once a user is logged in, the top right side of the application will show the name and profile picture of the user. By clicking on it, they will be able to access various pages, depending on the type of user they are.
By default, the index page shows the Hottest Games, but one can switch to Newest or Best Games through the respective buttons. Clicking on a game will take the user to that game's page. Scrolling further down, more games will be loaded, and the suggested games and users will be shown (if any are available).

7.2 Game Page

In this page all the information about a game is shown, along with the top 3 reviews (by number of likes) for that game. The user can view all the reviews for the game by clicking on ”SHOW ALL”. The names in ”Developers” and ”Publishers” have links that lead to that company's page.

7.3 Reviews Page

The Reviews page shows all reviews for a particular game. On the right is a chart showing the distribution of the score. To add a review, a user has to click on the ”+” icon that is at the bottom of the game's page or the one that is above all reviews in the Reviews page.

7.4 User's profile

The user can like a review by clicking on the heart, and he can view all comments for that review by clicking the "Comments" button. To comment on a review, the user has to click on the "comics cloud" shaped icon. Every user has a profile page that shows their info, along with the top 3 reviews they wrote that received the highest amount of likes. A symbol next to the name will indicate if the user is an administrator or a company manager. One can see the Followers or Followed users of an account by clicking on the respective buttons. Once a user is logged in, they can modify their info by clicking on the edit button.

7.5 Company Panel

This page is only accessed by Company Managers. It allows them to publish new games, edit existing ones or delete them. It also shows the statistics for the games of the company. To choose the operation to perform, the user clicks on one of the buttons below the search bar.

7.5.1 Publishing a Game

To publish a game, the following attributes are mandatory:

- Name
- Description
- Image URL
- Release Date
- Platform
- Publishers
- Developers
- Genre

Name, Description and Image URL simply have to be written inside the input element. Every other attribute must be added by clicking on the "+" button. The user can add any number of values for the attributes with the "+" button, and in any format, *except* for Release Date. Only one Release Date can be inserted, and it *must* be in the following format: `yyyy-mm-dd`. The last input element lets the user add his own attributes. They must write the name of the attribute on the left, the value of the attribute on the right, and then click the "+" button. To delete an attribute value, the user can click on the "X" button next to it.

7.5.2 Editing a Game

To edit a game, the user must first select one through the dropdown menu. Then they can modify or add new attribure values, the same way it is done for the publishing.

7.6 Control Panel

This page can only be accessed by administrators. Here they can view statistics on the Stats tab, and access the administrator's terminal on the Terminal tab.

7.6.1 Terminal

Here are the available commands:

- `help` : shows all available commands.
- `ban <username>` : bans (deletes from database) a user
- `delete <review|comment|game> <id|name>` : deletes any specified review, comment, or game
- `update <games|users|companies|all>`
  - `games` : it runs the method that updates the top 3 reviews by number of likes, for every game
  - `users` : it runs the method that updates the top 3 reviews by number of likes, for every user
  - `companies`: it runs the method that updates the top 3 games by average score, for every company
  - `all` : runs all 3 methods.