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.

- Allow users to retrieve various informations on videogames
- Allow users to review videogames and give them a score
- Allow users to follow other users, like and comment other users' reviews
- Give users recommendationson on who to follow and what games to play
- Allow videogame company managers to retrieve analytics on the sentiment of the reviewers for a particular game
- Allow the application manager to see which users are the most active on the site, so that he can promote them as the best reviewers

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. To do so, we used Python algorithms. *insert how algorithms work*

- **Variety** : we exploited two real different sources, having different formats
- **Velocity/Variability** :
  - Comments are frequently added/liked/responded to
  - Most active users change periodically

2.3 Raw Data

Data was retrieved from:

- MobyGames
- MetaCritic
- generated data

2.4 Scraping Algorithms

We used *Python* for scraping and data generation

2.4.1 Videogames Scraping

Data is scraped from *MobyGames* for videogames info

2.4.2 Users and Review Scraping

Some data is scraped from *Metacritic* for reviews, while other reviews are generated. Comment on reviews are generated using algorithms

2.5 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.
- *Reviewer (User)* : end-user of the application.
- *Company Manager* : it's a special kind of *user*, who is granted more benefits. Company Managers identify Game Producers/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 menage 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
- delete their reviews
- delete their comments

Company Manager Exclusive

Note: all of the following apply only to games made or produced 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
- 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 more attributes to a game they want to add, 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.

3.3 Use Case Diagram

We can look at the use case diagram of the application in Figure 3.1 A *guest user* is able to

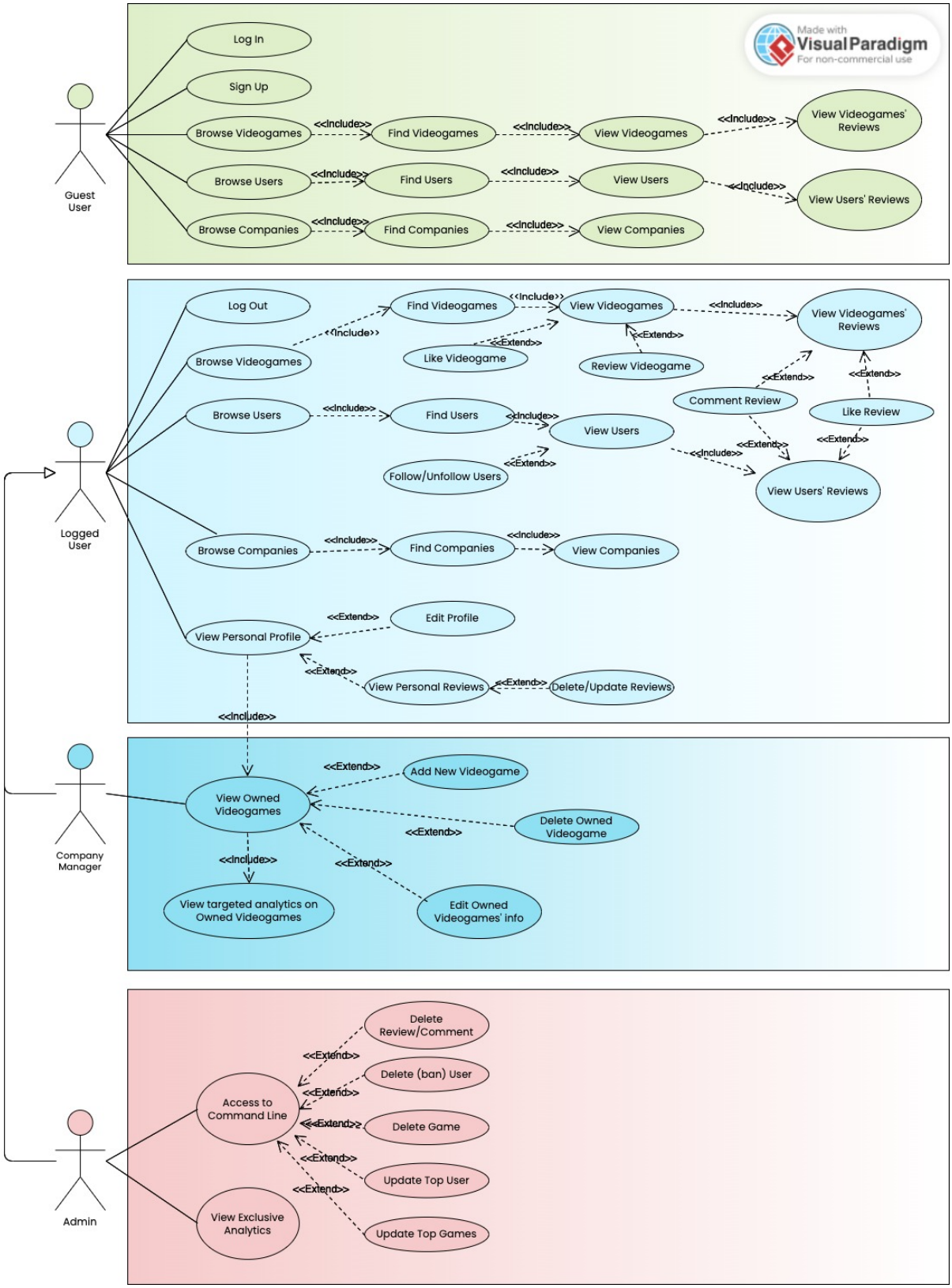

Figure 3.1: Actors and main supported functionalities

3.4 UML Class Diagram

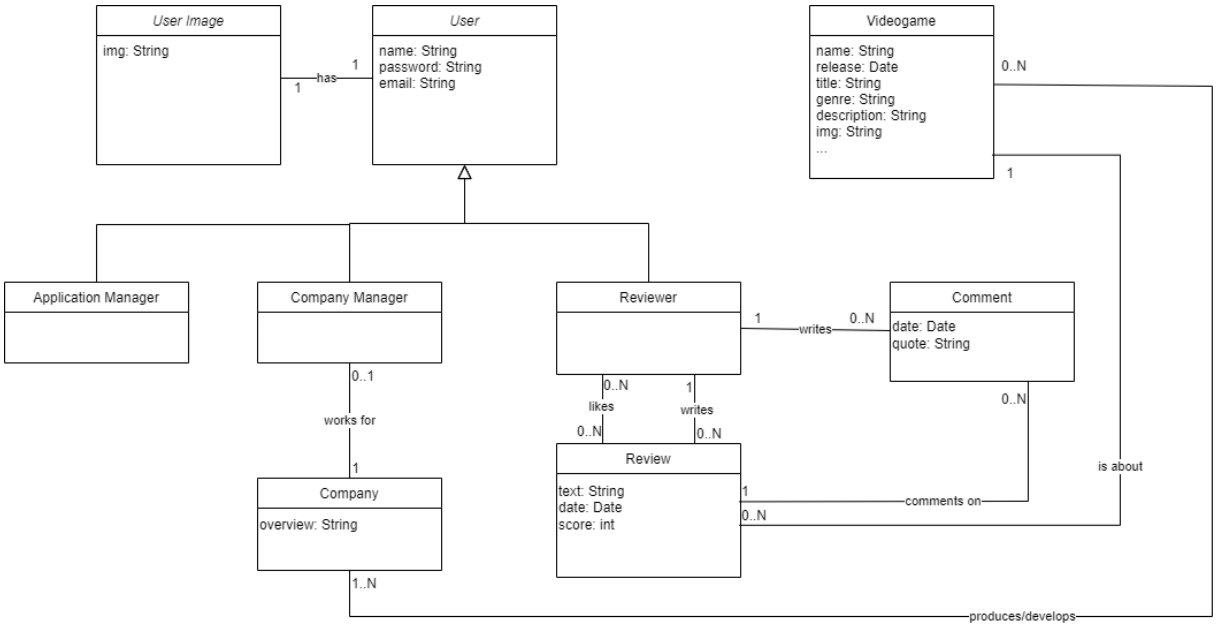

Figure 3.2: UML Class Diagram

The UML class diagram is reported in Figure 3.2

3.4.1 Classes

A *User* can be either be a *Reviewer*, a *Company Manager* or an *Administrator*.

3.5 Data Models and Implementation

3.5.1 Document DB Collections

Videogames

Whenever a new videogame is inserted, the following attributes are mandatory:

- Videogames
  - Videogames
- Users
- Reviews
- Comments
- userImages

3.5.2 Document DB Relationships

3.5.3 Document DB Examples


```
    {
  "Name": "(Almost) Total Mayhem",
  "Released": "January 14th, 2011 on 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": "description",
  "user_review": 6.0,
  "reviews": [
  {
    "score": 8,
    "quote": "stunning graphhcs!",
    "author": "Yusoreqa",
    "date": "2023-10-30",
    "source": "random",
    "comments": [
    {
      "author": "Hojosu",
      "quote": "I see that Yusoreqa agrees with me.",
      "date": "2023-12-06"
    }
    ]
  }
  ]
}
```


3.5.4 Graph DB - Neo4j

We handled the following entities via GraphDB (figure 3.3) :

- Users
- Videogames
- Reviews

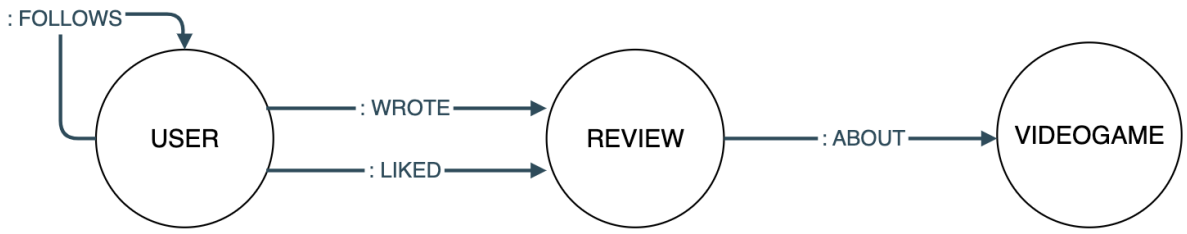

Figure 3.3: Entities handled by *GraphDB* and their relationships

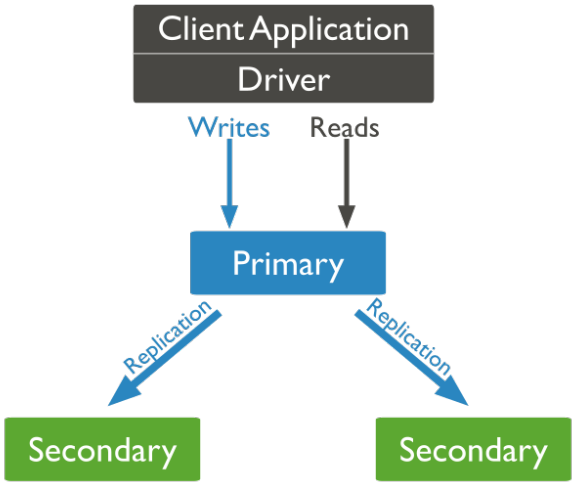

Figure 3.4: Replica Set

3.6 Distributed Database Design

Eventual consistency is sufficient, enforcing strong consistency would be too costly.

3.6.1 Replicas

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 3.4). 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.

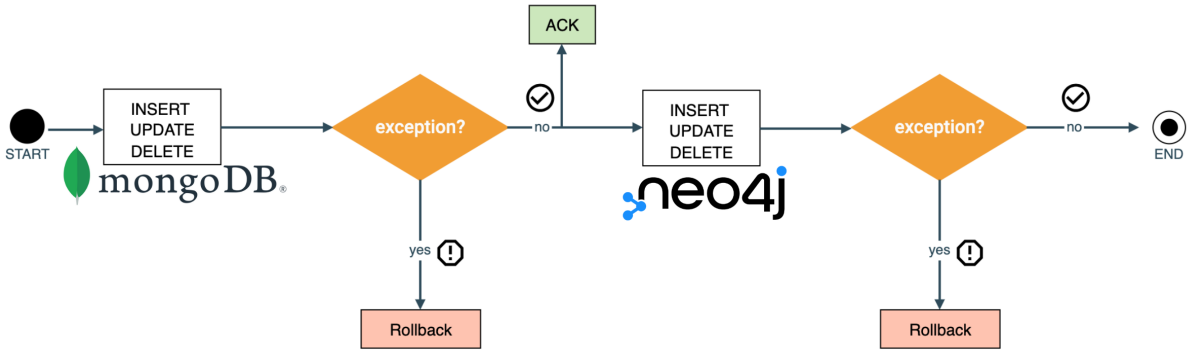

Figure 3.5: Handling consistency between *MongoDB* and *Neo4j*

Write Concern

Write concern describes the level of acknowledgment requested from MongoDB for write operations to Replica sets (TO DO: relation with Async)

3.6.2 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. To implement sharding, 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. We can implement sharding for Reviews and Comments, using their *IDs*. We shard users and reviews

3.6.3 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, whenever a write operation is performed successfully on *MongoDB*, an ACK (*acknowledgement*) is sent to the user. Data on *GraphDB* will be eventually consistent: if any exception occurs when the updates over data on *Neo4j* are attempted, than a rollback operation will be executed in order to bring back both the databases in a consistent state (just as shown in figure 3.5). Whenever the systems performs an operation that leads to changes (insertions or updates) over the data stored in *Neo4j* we handle possible exceptions, testing for errors in the execution of the code as follows: if an error occurs in the *try* block, we wait for a certain amount of time and we try to re-execute the operations (into the *catch* statement) until it is carried out correctly. We do this up to a defined maximum number of times $N = 10$, and at each try the time that the system will wait to make the next attempt will increase. Hopefully, by doing this, even when the server is down the operations will still be carried out once the server will get back up. If N or the chosen time in between every try, turn out to be not sufficient for the operation to be carried out correctly, than an error will be reported in the *log* file.

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.

Chapter 4

Implementation

Application:

- *Java* with *Maven*
- *SpringBoot*
- *Javascript, HTML, CSS*

4.1 Spring Boot

4.1.1 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.

4.1.2 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.

4.2 Java Entities

4.2.1 Users

Simple User - Reviewer

Company Manager

Admin

4.2.2 Videogames

4.2.3 Reviews and Comments

Chapter 5

Queries

5.1 Mongo DB

5.1.1 Users

Get top Users from Reviews

```
/*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 += ")";
```

```
26
27      Aggregation aggregation = Aggregation.newAggregation(
28      Aggregation.match(Criteria.where("date").regex(regex)),
29      Aggregation.group("author").count().as("reviews"),
30      Aggregation.sort(Sort.Direction.DESC, "reviews"),
31      Aggregation.limit(10)
32      );
33      List<DBObject> user_dbos = mongoTemplate.aggregate(aggregation, "
            reviews", DBObject.class).getMappedResults();
34      List<TopUserDTO> users = user_dbos.stream().map(user -> new TopUserDTO(
            user.get("_id").toString(), (Integer)user.get("reviews"))).toList();
35      return users;
36  }
```

Get top 3 Users' Reviews from Likes

```
1   /*gamecritic/repositories/User/CustomUserRepositoryImpl.java*/
2   public void updateTop3ReviewsByLikes() {
3       Aggregation aggregation = Aggregation.newAggregation(
4       Aggregation.stage(
5       "{\n" + //
6           "  $group: {\n" + //
7               "    _id: \"$author\",\n" + //
8               "    Top3ReviewsByLikes: {\n" + //
9                   "      $topN: {\n" + //
10                      "        n: 3,\n" + //
11                      "        output: {\n" + //
12                          "          _id: \"$_id\",\n" + //
13                          "          game: \"$game\",\n" + //
14                          "          quote: \"$quote\",\n" + //
15                          "          author: \"$author\",\n" + //
16                          "          date: \"$date\",\n" + //
17                          "          score: \"$score\",\n" + //
18                          "          likes: \"$likes\",\n" + //
19                          "        },\n" + //
20                      "        sortBy: {\n" + //
21                          "          likes: -1,\n" + //
22                          "        },\n" + //
23                      "      },\n" + //
24                  "    },\n" + //
25              "  },\n" + //
26          "}"
27      ),
28      Aggregation.stage(
29      "{\n" + //
```

```
30          "  $set: {\n" + //
31              "    username: \"$_id\",\n" + //
32              "  },\n" + //
33          "}"
34      ),
35      Aggregation.stage(
36      "{\n" + //
37          "  $project: {\n" + //
38              "    _id: 0,\n" + //
39              "    username: 1,\n" + //
40              "    Top3ReviewsByLikes: 1,\n" + //
41              "  },\n" + //
42          "}"
43      ),
44      Aggregation.stage(
45      "{\n" + //
46          "  $merge: {\n" + //
47              "    into: \"users\",\n" + //
48              "    on: \"username\",\n" + //
49              "    whenMatched: \"merge\",\n" + //
50              "    whenNotMatched: \"discard\",\n" + //
51              "  },\n" + //
52          "}"
53      )
54      ).withOptions(Aggregation.newAggregationOptions().allowDiskUse(true).
            build());
55
56      Instant start = Instant.now();
57      mongoTemplate.aggregate(aggregation, "reviews", DBObject.class).
            getMappedResults();
58      logger.info("Finished updating top 3 reviews by likes for all users in
            " + (Instant.now().toEpochMilli() - start.toEpochMilli()) + " ms");
59  }
60  }
```

5.1.2 Videogames

Get top Games from Average Score

```
1  /*gamecritic/repositories/Game/CustomGameRepositoryImpl.java*/
2  public List<TopGameDTO> topGamesByAverageScore(Integer months, String
       companyName, Integer limit) {
3      if (months == null || months < 1) {
4          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();
```

```
45      return games;
46  }
```

Get top 3 Videogames' Reviews from Likes

```
1   /*gamecritic/repositories/Game/CustomGameRepositoryImpl.java*/
2   public void updateTop3ReviewsByLikes()
3   {
4       Aggregation aggregation = Aggregation.newAggregation(
5       Aggregation.stage(
6       "{\n" + //
7           "  $group: {\n" + //
8               "    _id: \"$game\",\n" + //
9               "    Top3ReviewsByLikes: {\n" + //
10                  "      $topN: {\n" + //
11                      "        n: 3,\n" + //
12                      "        output: {\n" + //
13                          "          _id: \"$_id\",\n" + //
14                          "          game: \"$game\",\n" + //
15                          "          quote: \"$quote\",\n" + //
16                          "          author: \"$author\",\n" + //
17                          "          date: \"$date\",\n" + //
18                          "          score: \"$score\",\n" + //
19                          "          likes: \"$likes\",\n" + //
20                          "        },\n" + //
21                      "        sortBy: {\n" + //
22                          "          likes: -1,\n" + //
23                          "        },\n" + //
24                      "      },\n" + //
25                  "    },\n" + //
26              "  },\n" + //
27          "}"
28      ),
29      Aggregation.stage(
30      "{\n" + //
31          "  $set: {\n" + //
32              "    Name: \"$_id\",\n" + //
33              "  },\n" + //
34          "}"
35      ),
36      Aggregation.stage(
37      "{\n" + //
38          "  $project: {\n" + //
39              "    _id: 0,\n" + //
40              "    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

```
/*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;
}
```

5.2 Neo4j

5.2.1 Reviews

Get more liked Reviews

```
/*gamecritic/repositories/Review/ReviewRepositoryNeo4J.java*/
@Query(
    "MATCH (user:User {username: $username})-[:WROTE]->(review:Review)\n" +
    "OPTIONAL MATCH (review)<-[like:LIKED]-()\n" +
    "WITH review, COUNT(like) AS likeCount\n" +
    "ORDER BY likeCount DESC\n" +
    "LIMIT 3\n" +
    "RETURN review, likeCount;"
)
List<ReviewDTO> findMostLikedReviewsForUsers(
    @Param("username") String username
);

@Query(
    "MATCH (game:Game {name: $name})<-[:ABOUT]-(review:Review)\n" +
    "OPTIONAL MATCH (review)<-[like:LIKED]-()\n" +
    "WITH review, COUNT(like) AS likeCount\n" +
    "ORDER BY likeCount DESC\n" +
```

```
    "LIMIT 3\n" +
    "RETURN review, likeCount;"
)
List<ReviewDTO> findMostLikedReviewsForGames(@Param("name") String name);
```

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
);
```

5.2.2 Friends' suggestions

From users' relationships

From common interests

5.3 Indexes

5.3.1 MongoDB

MongoDB				
Query	**Collection**	**Index Keys**	**Documents examined**	**ms**
find("Name":name)	videogames	- Name	69553 1	38 *ms* 0 *ms*
find("author":author)	reviews	- author	250981 1	92 *ms* 3 *ms*
Query3	reviews	- {author, game}	numDoc numDoc	*ms* *ms*
find("game":game)	reviews	- game	250981 5	92 *ms* 3 *ms*
find("reviewId":reviewId)	comments	- game	625435 3	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*

5.3.2 Neo4j

Neo4j					
Query	**Index Name**	**Nodes' Label**	**Property**	**Hits**	**(ms)**
Query1	- *username_index*	- User	- username	numHits numHits	*ms* *ms*
Query2	- *game_index*	- Game	- name	numHits numHits	*ms* *ms*
Query3	- *review_index*	- Review	- reviewId	numHits numHits	*ms* *ms*

Chapter 6

Chapter