UNIVERSITÀ DI PISA

Computer Engineering
Artificial Intelligence and Data Engineering

Large-Scale and Multi-Structured Databases

Social News Web Application
Project Specification

Academic Year: 2022/2023

Authors:

Matteo Biondi

Martina Burgisi

Federico Cristofani

Contents

1 INTRODUCTION

Social News is a web platform that connects readers and reporters with the purpose of spreading the news directly from the reporters to the common people and interacting with them.

For readers, Social News offers a wealth of information and resources to stay informed on current events. Our platform allows readers to easily search and follow their favourite reporters, as well as leave comments and engage in discussions about the articles and posts that interest them the most. Additionally, readers can discover new voices to follow from a wide-ranging group of influential and popular publishers.

For reporters, Social News provides a platform to share their work and engage with their readers. With the ability to easily post articles, and view statistics about their readers and posts, reporters have all the tools they need to handle the publishing of news. Additionally, reporters can view statistics about the most active moments of the day, as well as their most popular posts, to better understand their audience and tailor their content accordingly.

Considering the Administrator's side, Social News is a powerful tool for managing and moderating the platform. Administrators can easily register new verified reporters, search and delete registered users, and even delete reporter's posts or reader's comments. Additionally, administrators can view statistics about readers and reporters, giving them a complete overview of the platform's usage.

2 REQUIREMENTS

2.1 Functional requirements - Unregistered user

1. The system must allow an unregistered user to become a registered ("reader") user by registering his/her full name (first name and surname), email address, password, country and gender in independent way.

2.2 Functional requirements - Reader

- The system must allow a reader user to login (by email and password) to the system.
- The system must display to the reader a page with the followed reporters (a simplified view of each reporter will be available for the user) as homepage. After a click on one of these, the associated reporter's page will be shown.
- The system must allow a reader user to find reporters page or posts. In the search bar the user can select "Reporter Name" or "Hashtag" from a drop-down menu to search respectively reporters and posts.
- After a search, the system must display to reader user a list of simplified view of reporters (with at least the full name and the picture) or posts.
- After a click on one of these results, the system must display to reader user the page of the associated reporter with the reporter information and posts.
- The system must allow a reader user to comment a post.
- The system must allow a reader user to read comments about posts.
- The system must allow a reader user to report a post.
- The system must allow a reader user to delete a his/her comment about a post.
- The system must allow a reader user to follow or unfollow a reporter by clicking a button on reporter's page.
- The system must allow a reader user to see the most influential and popular publishers that he/she doesn't follow yet (see "Statistics and queries" section).
- The system must allow reader user to view his/her own profile.
- The system must allow reader user to perform logout process.

2.3 Functional requirements – Reporter

- The system must allow a reporter to login (by email and password) to the system.
- The system must display to the reporter a page with his/her profile with posts history, number of followers, profile information and a section to create a new article.
- The system must allow the reporter to post a new article.
- The system must allow the reporter to delete a past article.
- The system must allow the reporter to display statistic about the most active moment of the day (see "Statistics and queries" section).
- The system must allow the reporter to display statistic about its most popular post (see "Statistics and queries" section).
- The system must allow reader user to perform logout process.

2.4 Functional requirements – Administrator

- The system must allow administrator user to login (by email and password) to the system.
- The system must display statistics about readers and reporters (see "Statistics and queries" section).
- The system must allow administrator user to register a new "verified" reporter to the service. To register a new reporter, admin must indicate: full name (first name and surname), gender, location

(with street, address number, city, state, postcode, country), cell number, email, password, picture, and date of birth.

- The system must allow administrator to search (by email) and delete a registered user. A simplified view of each user will be available, with at least user's full name, email, and a button to delete the associated profile.
- In case of reporter users, the system must allow the administrator to view the associated profile page, with posts and profile information.
- In case of reporter users, the system must allow the administrator to view all the report associated to his/her post. Each report link to the associated post.
- The system must allow administrator to delete reports associated with a reporter user.
- The system must allow administrator to delete reporter's post or reader's comment.
- The system must allow administrator user to perform logout process.

2.5 Notes about functional requirements:

All the requirements associated to reader, reporter or administrator are related to a logged user.

2.6 Non-functional requirements

- The system must be a website application.
- The system must encrypt users' passwords.
- The system must communicate with any user by using a secure communication channel (HTTPS).
- The system must be developed by using an OOP language (i.e.: Java, Python, etc.)
- The system will be available 24 * 7.
- The system will ensure eventual consistency.
- The system must be intuitive to use for users without any specific training.
- The system must be able to run on at least the following main browsers: Google Chrome, Mozilla Firefox and Edge.

3 STATISTICS AND QUERIES

In the following will be listed the main queries needed to implement the system as described above. We have classified the operations in CRUD (Create, Read, Update and Delete) and statistics. The formers are those that allows retrieving and storing data in the database, so they are necessary to realize the core functionalities of the system. The statistics ones represent some complex queries that offers the possibility to extract some useful information from the database that will be displayed in the proper manner to the users.

Details about implementation of these operations will be presented later in the chapter 10

3.1 CRUD operations

1. ***Query*****:** Create new reader
   Side: Reader
   Description: The query allows a reader to sign up to the platform

2. ***Query*****:** Create new reporter
   Side: Admin
   Description: The query allows an admin to register a new reporter to the platform

3. ***Query*****:** Create new post
   Side: Reporter
   Description: The query allows a reporter to publish a post on its page

4. ***Query*****:** Create new comment
   Side: Reader
   Description: The query allows a reader to publish a comment on a post

5. ***Query*****:** Create new report
   Side: Reader
   Description: The query allows a reader to report a post

6. ***Query*****:** Create new following relationship
   Side: Reader
   Description: The query allows a reader to follow a reporter

7. ***Query*****:** All readers
   Side: Admin
   Description: The query allows the admin to retrieve all the readers registered to the website, possibly filtering by email

8. ***Query*****:** All reporters
   Side: Admin
   Description: The query allows the admin to retrieve all the reporters registered to the website, possibly filtering by email

9. ***Query*****:** Reader by email

Side: Reader, Admin
Description: The query allows to retrieve information about a reader and to show reader's profile

10. ***Query*****:** Reader by email and password
    Side: Reader
    Description: The query allows to authenticate a reader during the login phase

11. ***Query*****:** Reporter by email and password
    Side: Reporter
    Description: The query allows to authenticate a reporter during the login phase

12. ***Query*****:** Reporter by reporter id
    Side: Reporter
    Description: The query allows to load the reporter page

13. ***Query*****:** Reporter by full name
    Side: Reader
    Description: The query allows to search for a reporter by his\her full name

14. ***Query*****:** Admin by email and password
    Side: Admin
    Description: The query allows to authenticate an admin during the login phase

15. ***Query*****:** Post by post id
    Side: Admin
    Description: The query allows the retrieve the post by its unique identifier

16. ***Query*****:** Post(s) by reporter id
    Side: Reader
    Description: The query allows the retrieve the post(s) published by the specified reporter

17. ***Query*****:** Post(s) by hashtag
    Side: Reader
    Description: The query allows to search for post(s) that contains a specified hashtag

18. ***Query*****:** Comment(s) by post id
    Side: Reader, Reporter, Admin
    Description: The query allows to retrieve the comment(s) published on a specified post

19. ***Query*****:** Followed reporters by reader
    Side: Reader
    Description: The query allows to retrieve reporters followed by a reader

20. ***Query*****:** Count reporter's followers
    Side: Reader, Reporter, Admin
    Description: The query allows to retrieve the number of followers of a reporter

21. ***Query*****:** Reports by reporters
    Side: Admin
    Description: The query allows to retrieve all reports on a reporter's post

22. ***Query*****:** Delete reader
    Side: Admin
    Description: The query allows to remove a reader from the platform

23. ***Query*****:** Delete reporter
    Side: Admin
    Description: The query allows to remove a reporter from the platform

24. ***Query*****:** Delete post
    Side: Reporter, Admin
    Description: The query allows to remove a post from the platform

25. ***Query*****:** Delete comment
    Side: Reader, Admin
    Description: The query allows to remove a comment from the platform

26. ***Query*****:** Delete following relationship
    Side: Reader
    Description: The query allows a reader to unfollow a reporter

27. ***Query*****:** Delete report
    Side: Admin
    Description: The query allows to remove a report from the platform

3.2 Statistics

1. ***Statistic:*** Most active readers
   Side: Admin
   Description: The statistic shows the first 10 active[(1)] readers.

2. ***Statistic:*** Most popular reporters
   Side: Admin
   Description: The statistic shows the first 5 popular reporters[(2)].

3. ***Statistic:*** Gender statistic
   Side: Admin
   Description: The statistic shows a graph about the numbers of women and men readers.

4. ***Statistic:*** Nationality statistic
   Side: Admin
   Description: The statistic shows a graph about the nationalities of the readers.

5. ***Statistic:*** Hot posts in the period

Side: Reporter
Description: The statistic shows the 10 hottest posts[3] (of a certain period of time) of the reporter.

6. ***Statistic:*** Most active moment of the day
   Side: Reporter
   Description: The statistic shows the most active moment of the day[4].

7. ***Statistic:*** Suggested new reporters
   Side: Reader
   Description: The statistic shows most popular reporters (based on the number of followers) that the user doesn't already follow.

Legend:

(1) Active user: The Active level of a reader is computed as the number of comments for a certain period of time.
(2) Popular reporter: Reporter with a large number of followers.
(3) Hot posts: Posts (of a certain period of time) with a large number of comments left.
(4) Most active moment of the day: Time slot of the day when comments are posted by the readers most frequently.

4 BASE PAGES MOCK-UP

4.1 Base reader user page

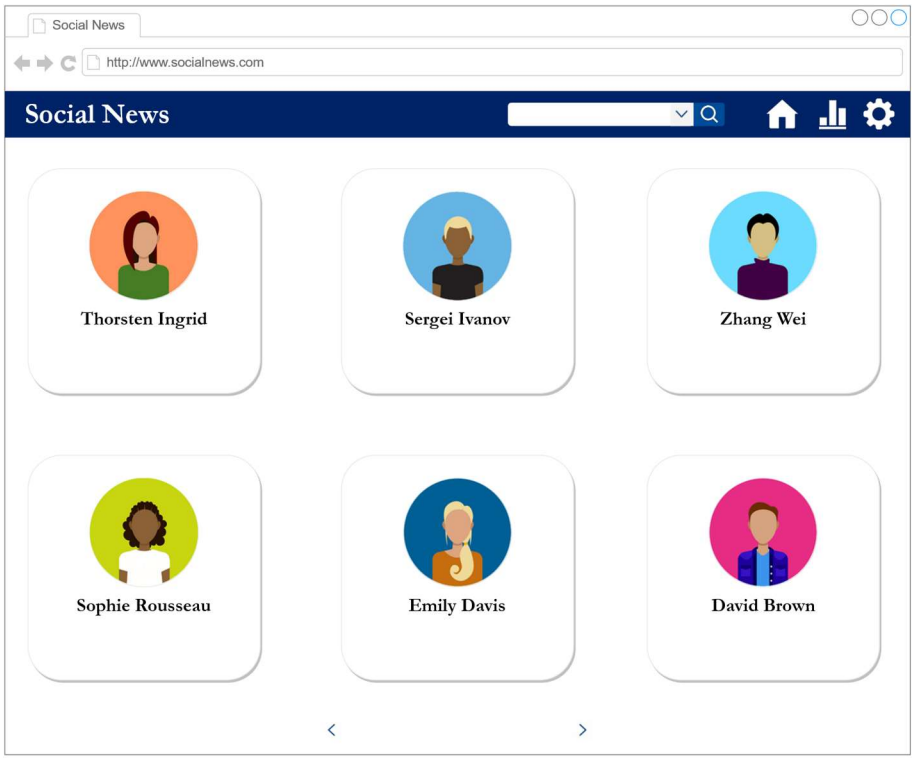

Figure 1 - Mock-up of the home page of the reader user. The reader sees a set of reporters that can be selected to check last news posted from him/her

4.2 Base reporter user page

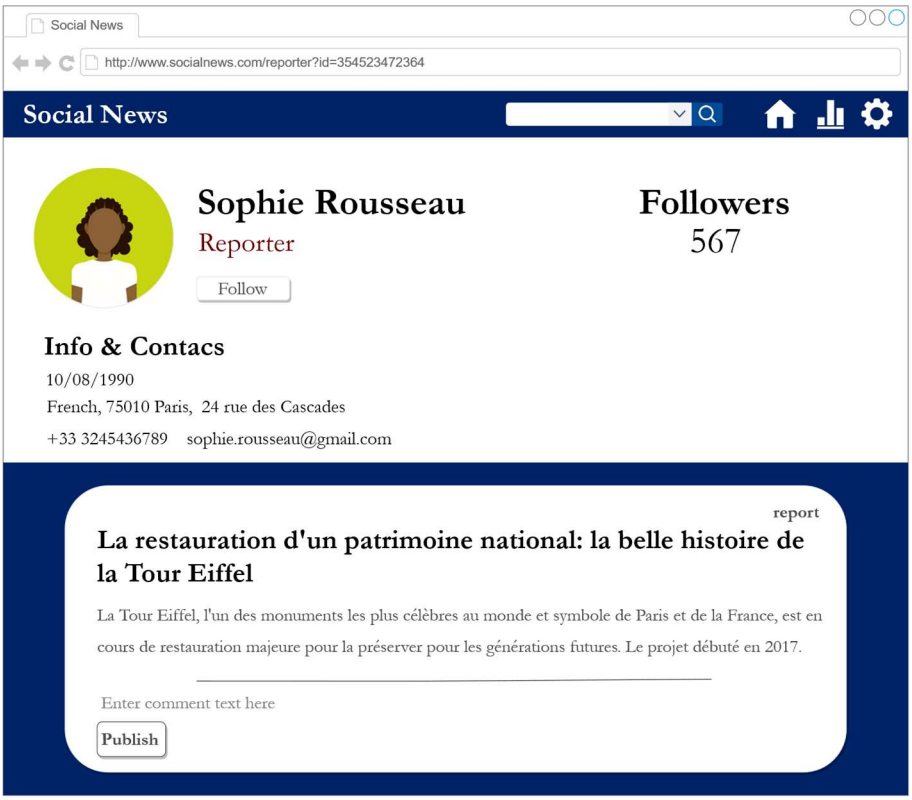

Figure 2 - Mock-up of the reporter profile seen by a reader user. The page shows information about the reporter and the news posted from him/her. Readers can also add comments below the posts.

4.3 Base admin user page

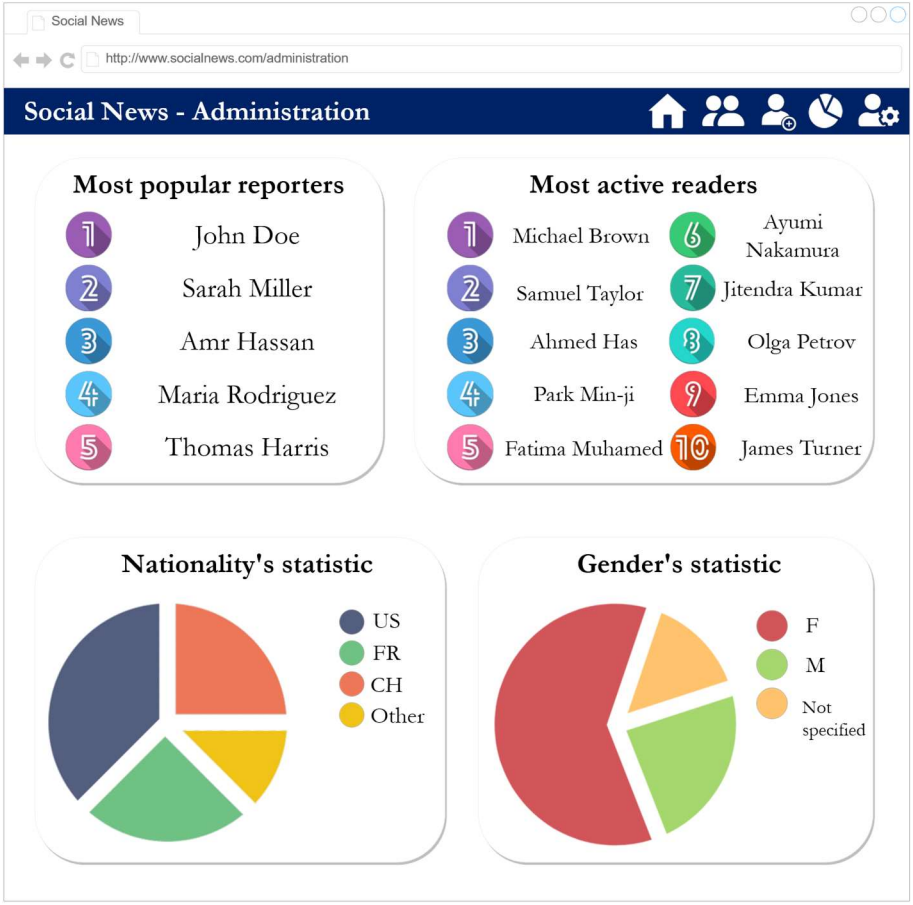

Figure 3 - Mock-up of the statistics page for administrator user. The administrator sees a set of statistics and rankings about the readers and the reporters.

5 UML USE CASE DIAGRAM

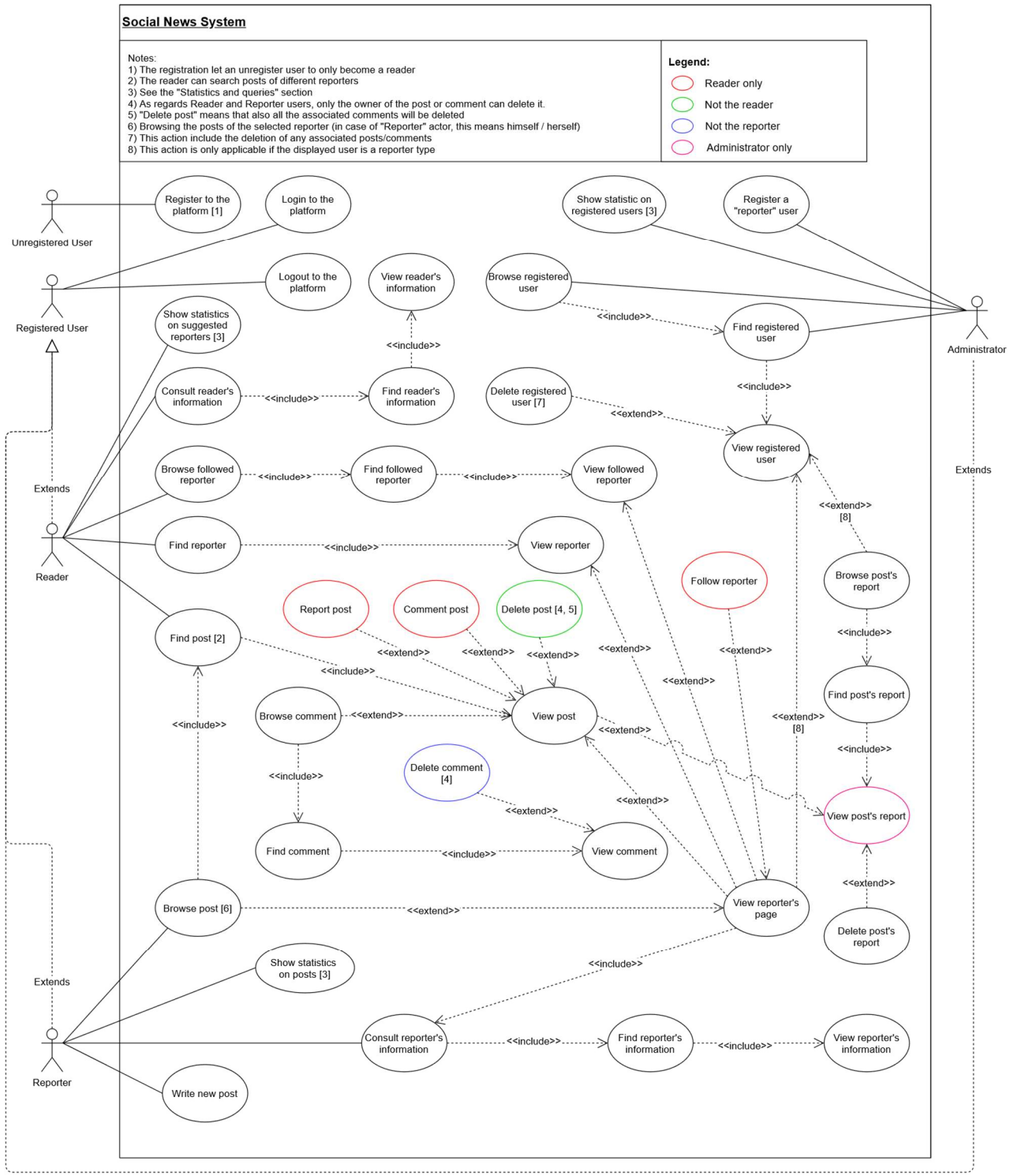

Figure 4 - The use case diagram, that represents the interactions between the actors Unregistered User, Registered User and consequently Reader, Reporter and Administrator, and the system through use cases

6 UML CLASS ANALYSIS

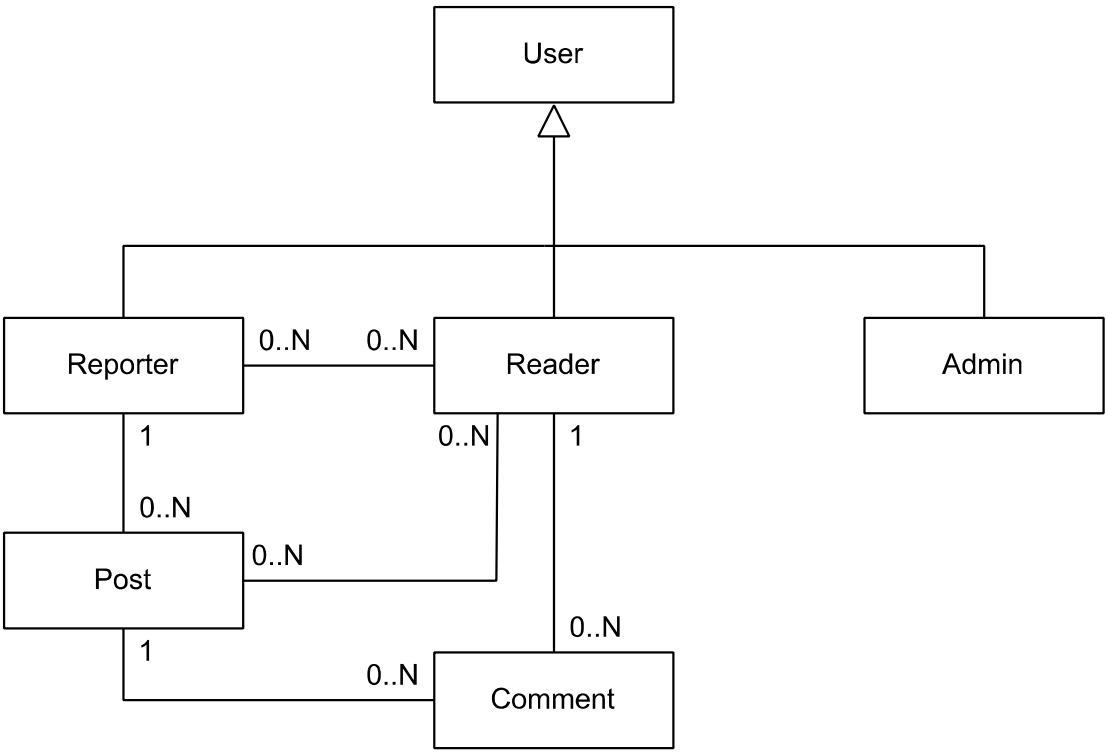

Figure 5 - UML Class Analysis

- **Reporter**, **Reader** and **Admin** classes extend the common class **User**
- **Post** and **Comment** classes handle news posts and interaction from the readers

7 UML CLASS DIAGRAM

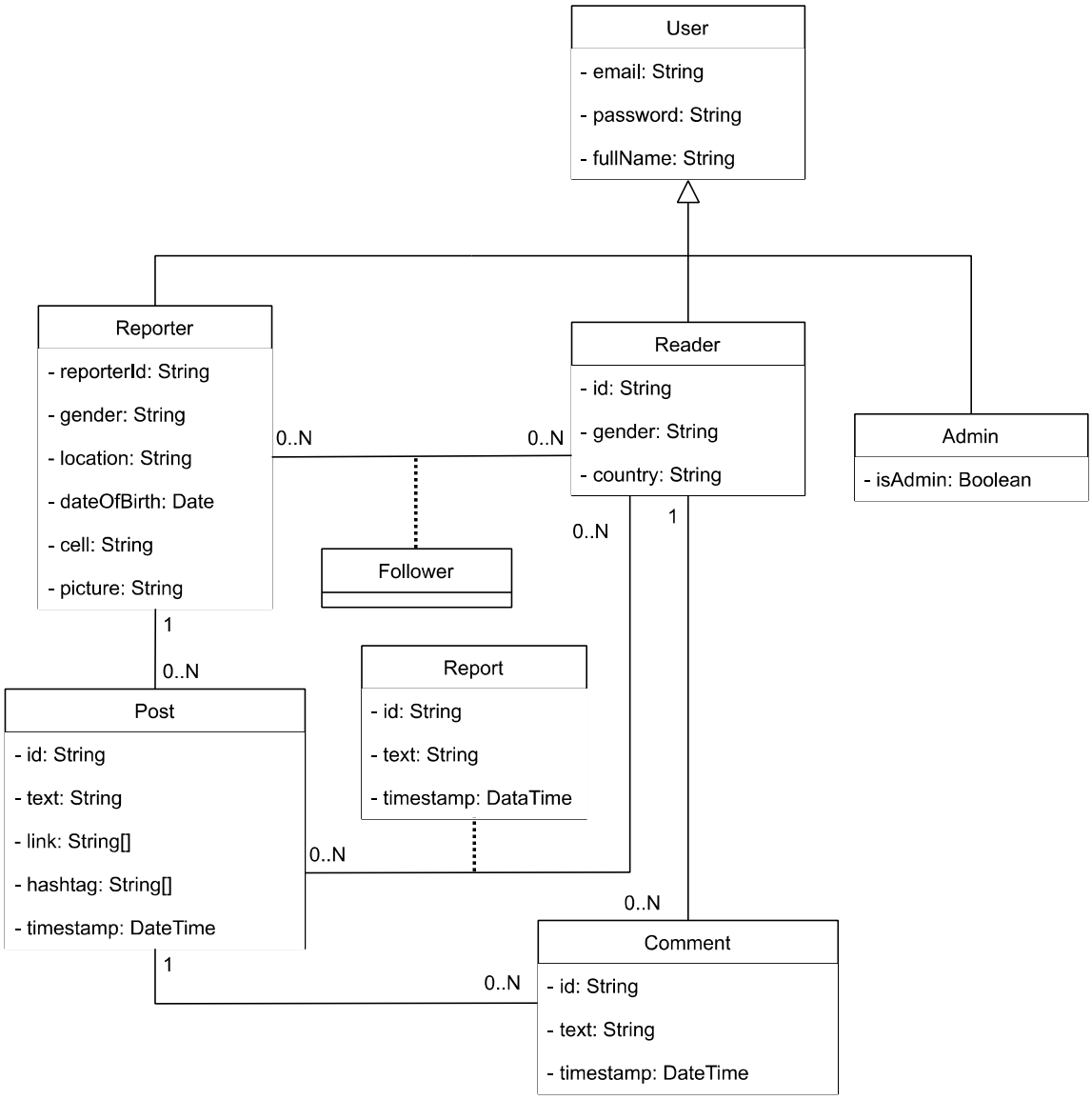

Figure 6 - UML class diagram with attributes

8 LOAD ESTIMATION

The architecture of the system, and more specifically the choice of database organization, is closely related to the workload the platform will have to handle. Thus, it is essential that subsequent design choices be made having a clear understanding of the expected workload, both in terms of quantity and type. The latter can be achieved by characterizing the work according to the most frequent operations.

Regardless of the absolute number of operations, we can state that:

- Number of readers will outperform the number of reporters.
- Number of comments will be much higher than the number of posts.
- Reading posts and comments will be the most frequent operation, so it is important to adopt choices that benefit a read-heavy system, such as the introduction of indexes described in a following section.

Regarding the load in terms of quantity, it is not easy to produce numbers that fit the actual amount of work the system will be subjected to. Moreover, fixing numbers can be misleading for some choices, risking forgetting the essential scalability property that must always be guaranteed.

Taking in mind this we still want to specify the orders of magnitude expected for the main entities in the system:

Entity	Order of magnitude (letters)	Order of magnitude (numbers)
Reader	Tens of millions	~10^7
Reporter	Tens of thousands	~10^4
Admin	Few hundreds	~10^2
Post	Few millions	~10^6
Comment	Hundreds of millions	~10^8

Table 1 - Expected volume size

9 DATABASE

The system must handle a large volume of data, so the choice of database platform has a significant impact on both system performance and ease of implementation, taking into account initial requirements.

The data that the system must handle are supposed to have a *heterogeneous format*, some fields may appear only rarely or have multiple values.

A relational database may not be suitable because of the lack of flexibility required by the type of data in the system. Moreover, if the system grows sufficiently, the relational database may not adequately support, at least economically and quickly, horizontal scalability achieved by distributing data across multiple servers.

The ability to handle complex data format in a very flexible manner, along with the ease of distribution and replication, that allows to achieve high availability and scalability requirements, lead to the choice of a document database.

Although all functionalities may be implemented by this NoSQL database, we noticed that some functionalities present properties that fit very well in a graph structure, such as the relationship between readers and reporters. This observation led us to introduce a new database architecture in our system, simplifying some recurring operations involving relationships between some entities, as will be better described in the dedicated section.

9.1 Document database

The entities that will be stored in document database are:

- User
  - Reader
  - Reporter
  - Admin
- Post
- Comment

9.1.1 Collections

The mapping of entities into database collections has been done taking into account two guidelines:

- Organize collections keeping in mind the queries that will be executed, removing dependencies between collections and structuring documents to simplify the workflow of each operation
- Reducing as much as possible the number of collections handled by the database

Following these guidelines, we have defined three collections:

- ***User***, stores info about both readers and admin(s)
- ***Reporter***, stores info about reporter and post
- ***Comment***, stores info about comments published by readers

User

The collection of users includes information on both readers and admins, basically some personal info and login credentials. The structure of each document is rather simple and quite similar in both types of users, so this led to the choice to include admins in this collection. Moreover, as will be described below, the collection reporter has a complex structure, so the choice to put together readers and admin(s), instead reporters and admin(s) allows to save this additional complexity to that collection.

Reporter

This collection maintains the information about each reporter registered in the system. The structure of each document is complex because in addition to the static personal info and contacts we have decided to embed the list of posts published by the reporter.

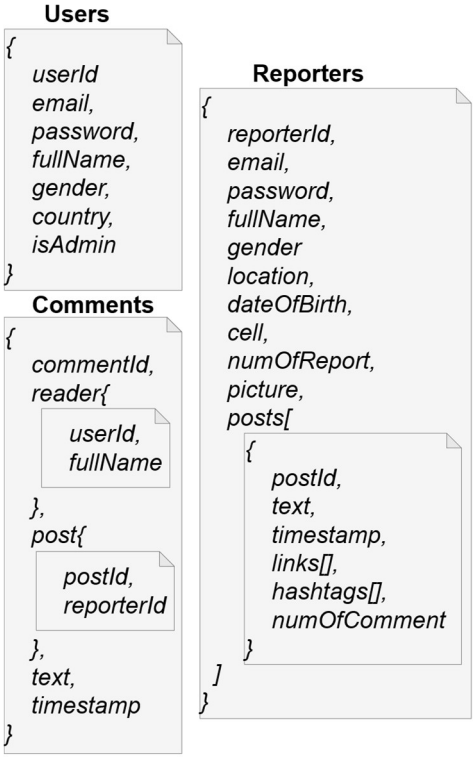

Figure 7 - Document database collections

Each post includes several fields, up to a maximum size of 5KB. This constraint has been introduced to limit the workload of operations on database and network and to guarantee that a reporter can publish a reasonable number of posts before reach the maximum size of document imposed by database[1]. However, we have decided not to limit the number of posts that a reporter can publish, handling the limit on document size in a transparent manner for users of the systems: when the document reach the maximum size, a new document is created, moving some static information from the old document to the new one, avoiding a useless repetition of such information. The system ensures the correctness of this transition from the old (no more active) document to the new one (the active one) through an ACID transaction. When a query is executed, the operation considers all documents, both active and not active. This ensures that the system retrieves all posts published by a reporter that match the condition, not only the active ones.

The main advantages of keeping the reporter's information and posts in the same collection are evident when accessing the reporter's page. In fact, it is possible to retrieve all the necessary information stored in the document database with a simple query.

Comment

The last collection stores info about comments posted by readers about posts. Each document represents a comment and is composed by a field containing the text message, some metadata and other fields that allow the comment to be associated with its author and the post on which it was published. The adopted strategies to set information about author and post are different:

- ***Post***, document linking
- ***Author***, document embedding

The choice is driven by the queries that the system must support, it's possible to notice that:

- The comments associated to a post are retrieved from database only when user selects a post, so the details about post are already loaded at that instant and it's possible to save a lot of disk space including only the post identifier rather than embedding full details
- The text of a comment and the full name of the reader who wrote that comment are fields that are always needed together, so it's necessary to put that information in the same document (and thus collection) to avoid costly join between collections

There is no widely adopted standard naming convention for the fields that make up a document. Therefore, we decided to adopt some guidelines published by Google on JSON documents.

The field name is self-explanatory, but a few points need to be emphasized:

- Entities are uniquely identified by their id

[1] Latest version of MongoDB limits maximum document size to 16MB

- The field email in users' collection could guarantee uniqueness, however we decided to introduce an identifier for maintainability reasons, preserving the possibility to add the modification of email in future releases
- The passwords are hashed for security reasons
- The picture is stored directly in the database and has a maximum size constraint
- Some fields are a composition of several sub-fields, such as "*location*", but we keep it as a single field because in the application the information will be always displayed together
- "*numOfComment*" and "*numOfReport*" are redundant fields that allows to simplify some operation. More details about redundancies can be found in the following.

9.1.2 Queries

For clarity we briefly report the queries that refers to entities stored in the document database and that will be implemented leveraging the query language offered by database. Details about the queries are already provided on previous chapter.

CRUD operations

- **CREATE**
  - Reader
  - Reporter
  - Comment
- **READ**
  - All readers/reporters
  - Reader/Reporter by email
  - Reader/Reporter/Admin by email and password
  - Reporter by reporter id
  - Reporter by full name
  - Post by post id
  - Post(s) by reporter id
  - Post(s) by hashtag
  - Comment(s) by post id
- **UPDATE**
  - Add new post to reporter
  - Delete post from reporter
- **DELETE**
  - Reader
  - Reporter
  - Comment

Note that the creation and deletion of posts are classified as update operation because a post is embedded inside a reporter document.

Statistics queries

The complex query needed to implement some of the statistics offered by the system reinforce the choice to use document database to store the information. The following statistics will be executed on data stored in document database:

- Most active readers
- Gender's statistic
- Nationality's statistic
- Hot posts of a certain period

- Most active moment of the day

9.2 Graph database

The Social News platform will utilize a graph database to store and manage the relationships between readers, reporters and posts. The graph database will consist of three types of nodes that represent readers, reporters, and posts. An edge will connect the first two nodes, representing the "FOLLOW" unidirectional relationship from a “Reader” to a “Reporter”. Another edge will connect a “Reader” to a “Post” with relationship called “REPORT”, and the last one connects a “Reporter” to a “Post” with a “WRITE” operation.

Specifically, in the graph database, each “Reader” will be represented by a node with the following properties:

- readerId: a unique identifier for the “Reader”

Each “Reporter” will be represented by a node with the following properties:

- reporterd: a unique identifier for the “Reporter”
- fullName: the full name of the “Reporter”
- picture: the “Reporter” 's profile picture

Each “Post” will be represented by a node with the following properties:

- postId: a unique identifier for the “Post”

The relationship between a “Reader” and a “Reporter” will be represented by the edge of type "FOLLOW" which have no properties, and which represent the follow relationship between a “Reader” and a “Reporter”.

The relationship between a “Reader” and a “Post” will be represented by the edge of type “REPORT” with the following properties:

- reportId: a unique identifier for the “REPORT” relationship
- text: the text associated to the report written by a “Reader”
- timestamp: the time and date associated to the “REPORT”

The relationship between a “Reporter” and a “Post” will be represented by the edge of type “WRITE” which have no properties, and which represent the write operation of a “Post” from a “Reporter”.

For the specific operations see the paragraph with the operations list.

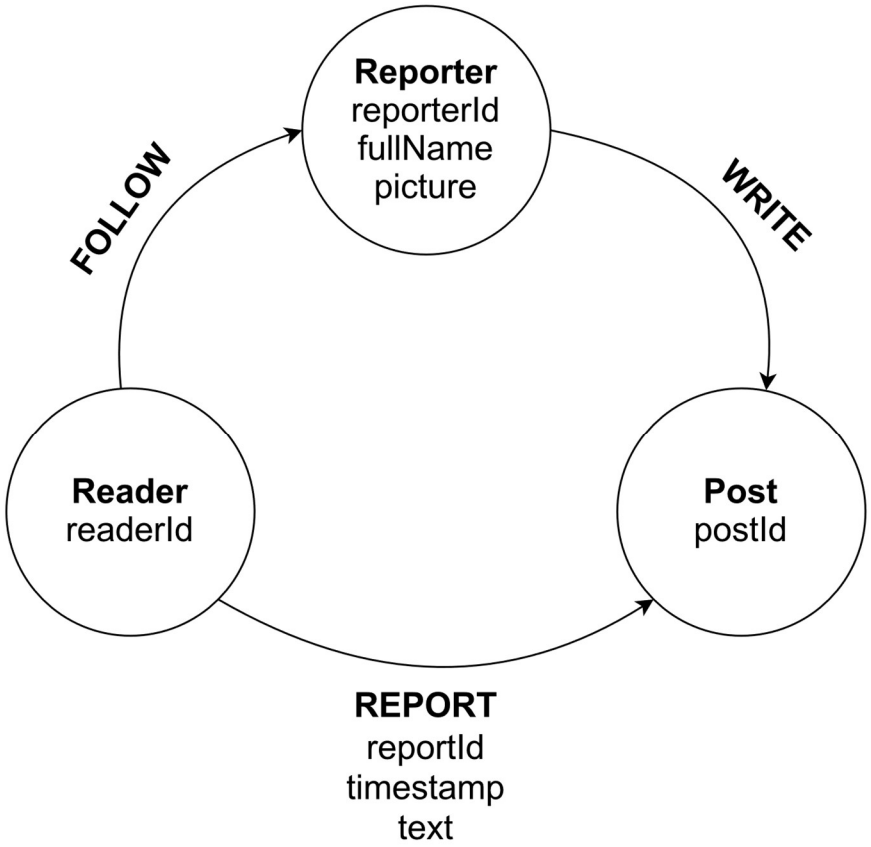

Figure 8 - Graph diagram with proper nodes and edges, with associated attributes

9.2.1 Queries

For clarity we briefly report the queries that refers to entities stored in the graph database and that will be implemented leveraging the query language offered by database.

CRUD operations

CREATE	
Domain-specific	**Graph-centric**
A new reader type user is registered	Add new reader vertex
Admin registers a new reporter type user	Add new reporter vertex
A reporter publishes a new post	Add new post vertex and “WRITE” edge with the reporter as the owner
A reader follows a reporter	Add an edge directed from a reader to a reporter
A reader reports a post	Add an edge directed from a reader to a post

READ	
Domain-specific	**Graph-centric**
Which reporters are followed by a specific reader?	Which “FOLLOW” edges are outgoing from a specific reader vertex?
How many followers does a specific reporter have?	How many “FOLLOW” edges are incident to a specific reporter vertex?
How many reports are associated to a specific reporter?	How many “REPORT” edges are incident to post vertexes that are linked to the reporter vertex with “WRITE” relationship?

DELETE	
Domain-specific	**Graph-centric**
Admin removes a reader account	Remove a reader vertex and all its outgoing edges
Admin removes a reporter account	Remove the “Reporter” node, all “Post” nodes linked with “WRITE” relationship and all ingoing edges (to deleted posts and reporter)
Admin user removes a reporter’s post or reporter removes his/her own post	Remove a post vertex and all its ingoing edges
Reader stops following a reporter	Remove the edge directed from reader to a reporter
Reader removes a report on a post	Remove the edge directed from a reader to a post

Statistics queries

STATISTICS	
Domain-specific	**Graph-centric**
Which reporters are the most popular?	Reporter vertexes with highest number of "FOLLOW” edges
Which reporters are suggested to a specific reader?	Reporter vertexes with highest number of incoming edges that haven’t yet a direct edge with the reader for which the statistic is computed

10 REDUNDANCIES

The performance of the application is a critical consideration, thus when making decisions, priority was given to fast query times over reduced memory consumption by implementing redundancies to minimize join operations. The following section will detail the redundancies implemented, as well as the mechanisms put in place to maintain consistency.

10.1 Number of comments' redundancy

The "number of comments" redundancy in the reporter's collection is designed for each of its posts (and saved inside the post embedded document) to avoid calculating the number of comments per post by retrieving the posts from the reporter collection and comments from the comments collection and then performing a join. It has been implemented to improve the performance of the system by avoiding the need to perform expensive join operations on large datasets. Instead, the number of comments for each post is stored directly in the reporter collection, providing quick and easy access to this information without the need for additional computation. This will be useful in the case of the process of the "Hot Post" statistic. The price we have to pay for this redundancy is to increment/decrement the value of the number of comments for each published/deleted comment about a certain post.

10.2 Number of reports' redundancy

The "number of reports" redundancy in the reporter's collection is designed to avoid building the page with the list of users by loading information first from MongoDB and then from Neo4J. Additionally, it helps to decrease the load when calculating the number of reports when an admin views the number of reports per reporter, in order to improve the performance of the system even in this case. The number of reports for each reporter is stored directly in the reporter collection, allowing for immediate and effortless access to this information without any additional computation. The cost of this redundancy is the need to increment/decrement the value of the number of reports for each report submitted/deleted.

10.3 Updating Redundancies

To keep redundancies up-to-date and thus have a consistent system, the application is provided with a redundancy update mechanism based on log files and periodic tasks. In particular, for each operation of publishing or deleting comments or reports, we write the operations performed on separate log files (one for comments and one for reports). Having separate files allows us to apply synchronization mechanisms by distinguishing the subject of the operation, thus implementing a finer-grained synchronization and obtaining a thread-safe mechanism for the logs. Periodically, the log files are read, and their contents are extracted. These operations must be processed to reduce the number of operations in the database. Finally, for each report and comment affected by operations, the relative redundancy is updated (in 'reporter' and 'post' respectively). Unsuccessful update operations must be kept in the log in order to be retried in a subsequent attempt (at the next execution of the periodic task).

10.4 Redundancy Update Queries

To complete what was presented in chapter 3, we define the queries required to update redundancies.

1. ***Query*:** Update comments counter
   Side: System
   Description: The query allows to update the redundant counter of comments on a post

2. ***Query*:** Update reports counter
   Side: System
   Description: The query allows to update the redundant counter of reports on a reporter's document

11 DOCUMENT DATABASE INDEXES DESIGN

The introduction of indexes is mandatory to improve performance of reading operations, especially in a read-heavy system as the Social News one. However, is not possible to introduce an arbitrary number of indexes and just hope to obtain better performance. First of all, the fields on which the indexes may be exploited by the queries that the system must implement must be accurately selected. Then it's necessary to take into account that the indexes require structure that must be stored in memory and updated each time the associated data are modified. Such structures involve a non-negligible overhead for the database engine that must be considered in the overall analysis.

The fields on which an index may be exploited by the queries are reporter in the following tables, jointly to the operations that may be disadvantage by the same index.

Users		
Query	**Field(s)**	**Disadvantage operations**
All readers	isAdmin fullName id	Create a new reader Delete a reader
Reader by email	email	Create a new reader Delete a reader

Table 2 – Tentative index fields

Reporters		
Query	**Field(s)**	**Disadvantage operations**
All reporters	fullName reporterId	Create a new reporter Delete a reporter
Reporter by reporter id	reporterId	Create a new reporter Delete a reporter
Reporter by full name	fullName reporterId	Create a new reporter Delete a reporter
Post by post id	posts.id reporterId	Create a new post Delete a post
Post by reporter id	reporterId posts.timestamp posts.id	Create a new post Delete a post
Post by hashtag	posts.hashtag post.timestamp posts.id	Create a new post Delete a post

Table 3 - Tentative index fields

Comments		
Query	**Field(s)**	**Disadvantage operations**
Comments by post id	post.id timestamp id	Create a new comment Delete a comment

Table 4 - Tentative index fields

It's worth noting that many fields used by the queries are not strictly related to the filter used to retrieve the data, but to the necessity to realize a pagination to avoid loading a huge amount of data with a single query.

At design level it's not easy to definitively understand which indexes may provide an effective advantage, however, it's possible to state that the operations that are damaged from indexes are far less frequent than the ones that are advantaged. Moreover, it's possible to ignore the field "isAdmin" from the ones that may compose an index because the number of admins in the system is insignificant compared to the number of readers, so the index wouldn't provide a real advantage.

From a design perspective the indexes on the remaining fields may lead to a performance boost, but some tests are required to state such statement.

Until now the discussed indexes are performance-related, but other types of indexes may be defined to guarantee some properties in the system. A desired property is the uniqueness of certain fields, necessary for the correctness of some operations. Such fields are:

- User id
- User email
- Reporter email
- Post id
- Comment id

On such fields a special index to guarantee the uniqueness must be defined, regardless the performance implications. For those fields the uniqueness is required because queries rely on them to retrieve an entity from the database, taking for granted that such entity is univocally identified by the field.

12 GRAPH DATABASE INDEXES AND CONSTRAINTS DESIGN

In the design phase, a careful assessment was made as to which indexes should be introduced into the graph database, for query performance issues, and which constraints should be defined in order to obtain a database with consistent content. A database index is introduced in order to make searching for related data more efficient. This comes at the cost of additional storage space and slower writes, to keep them up to date, so a trade-off is needed. For our purpose it is therefore sufficient, taking into account the queries described above, to introduce indices that allow us to identify nodes and relations often based on equality operators. Against this, the implementation of four indices is proposed for the ids of the nodes and relations contained in the graph. Details on their implementation and tests for their effectiveness are given in a later chapter of the documentation. The introduction of indexes of this type is justified, given the read-heavy tendency of the application and its use of the graph database.

Entity Type	**Entity Name**	**Indexed Attribute**	**Advantaged operations**	**Disadvantaged operations**
Node	Reporter	reporterId	Search reporter by id	Insertion and deletion of reporter
Node	Reader	readerId	Search reader by id	Insertion and deletion of reader
Node	Post	postId	Search post by id	Insertion and deletion of post
Relationship	Report	reportId	Search report by id	Insertion and deletion of report

Table 5 - Designed index on graph database

Assuming a load in the system equal to that reported in chapter 8, the motivation behind the addition of this indexes is that the number of searches of these nodes/relationship by means of their identifier is much greater than the number of their additions and removals.

In addition to identifying the indexes, during the design phase, the need arose to establish constraints guaranteeing certain characteristics of the content of our database, e.g. uniqueness constraints or constraints on the existence of certain properties. Since these constraints depend heavily on the graph database used, their implementation will be explained in more detail in a dedicated chapter.

13 SYSTEM ARCHITECTURE

The decision about the architecture to adopt plays a vital role in the development of the application. First it will be described in general terms, referring to the components without tying them to any specific implementation. Afterwards, the components actually used will be provided. This separation is due to the fact that the architecture is intended to be as general as possible, not strictly dependent on an implementation that may no longer be suitable for the application in the future.

13.1 General description

The system architecture was designed to meet the requirements described in detail in the previous chapters, trying at the same time to minimize the programming effort leveraging some components able to implement common functionalities.

The fundamental building block that represents the core of the application is the design choice regarding the databases. The data organization has been extensively discussed above, however the databases distribution and replication deserve to be analysed, even if the final configuration strictly depends on the available resources and will be presented in a dedicated section. The document database is organized as a cluster of three replicas, each of which maintains the same data, to guarantee the availability of the service. In contrast, the graph database in present only one instance, no replication is designed. The lack of redundancy of data on graph database may represent a bottleneck for the application, in future releases this problem should be addressed to overcome this limitation. Some insights will be presented at the end of the documentation to extend the work on this platform, especially with regard to the possibility of data sharding.

Although database architecture and organization are the key part of the design choices it's still necessary to define how to implement the logic of the application and how to expose its functionalities to the end users. Both tasks can be fulfilled via a web-application[2], a very common solution adopted nowadays for the implementations of applications intended for large scale utilization.

Many frameworks are currently available that allow the programmer to implement only the business logic of the application, without taking into account all the problems related to the standard implementation of application structure. The "socialNews" application, at least in the first version that we are presenting, doesn't need advanced features, so it's worth to select a basic framework that don't require a complex configuration. More specifically, the application logic is hosted on an application server, i.e. the platform that provides the environment that allows both the execution of the application logic and access by end users, also indicated as clients.

All the user requests are directed to the application server, that acts in the middle between end users and databases. The requests are handled by the application server following the defined business logic, usually composed by an access to the databases to retrieve/store information. The message exchange flow follows the client/server paradigm, i.e. is unidirectional from clients toward server. The requests are served in a synchronous manner, at least from the application server perspective, because the client can adopt some asynchronous technique to properly handle the user interface.

13.2 Frameworks and components

The application architecture can be divided in three levels: web interface, application logic and database. The first is realized adopting common web technologies, that allows to build advanced interface with low effort for the programmer. No framework has been adopted, but high-level libraries has been extensively used, including Bootstrap, JQuery, ZingChart, ZingGrid …

[2] The term web-application refers to any distributed application that can be accessed via the web

The server-side of the web application is realized leveraging the Java EE platform in its basic form, without adopting complex frameworks. As already anticipated this choice is due to the fact that there is no need for advanced features. This choice has led to the selection of Apache Tomcat as application server to host the logic because it provides the minimal environment to runs a Java EE web-application. The environment includes support for dynamic web pages built with JSP and Servlet technologies.

Finally, MongoDB and Neo4J are the implementation respectively of document and graph databases chosen for the system. The choice is driven by the popularity and set of functionalities that both databases can provide.

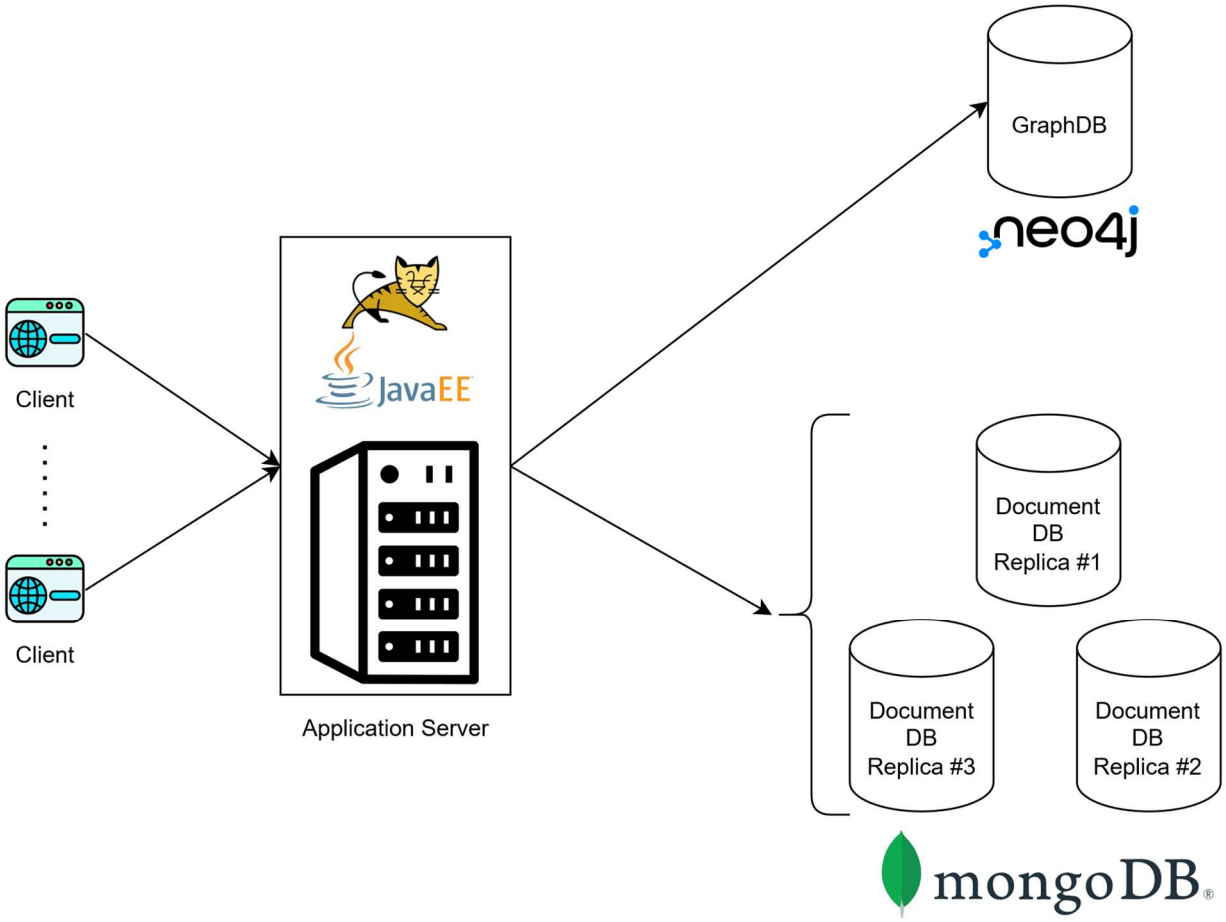

Figure 9 - System architecture

14 IMPLEMENTATION

The implementation of the system follows what was described in the design phase. Although the code has been appropriately commented, it is still appropriate to describe the structure adopted to simplify understanding by third parties. In the following the structure will be presented first in terms of the high-level modules, describing the functionalities they implement and the interactions with other modules and then in terms of the actual code entities implemented to realize such functionalities. Finally, the most relevant queries will be analysed and a focus on how redundancies are managed will be reported.

14.1 Main modules

Configuration

The module is responsible for the initialization of the application. It reads all configuration parameters from local file or the environment and distribute them to the other modules across the system in order to guarantee a proper configuration. It is also responsible for setting up connections with databases, building the necessary objects that will be used by other modules that perform action on the stored data. Finally, it constructs the singleton instance of some components that have special tasks in the application, such as the custom threading pooling manager responsible for the execution of the operations that require parallelization and the database redundancy manager.

All these operations are carried out during the deploy of the application. In case of failure the process stops, and the application won't be available until the errors will be fixed and the process successfully repeated.

In case of application shutdown some clean-up tasks must be performed, such as closing the connection with the databases. In order to avoid leaks of resources the module ensure the execution of such operation before stopping the application.

Data access

The data access module encapsulates all the interaction between application and databases, exposing a simple interface on which each method can be smoothly mapped to one of the queries previously discussed. The complexity of queries and connection management are hidden in order to provide an abstract view to other modules, basically implementing the DAO pattern. The modules that leverage the exposed functionalities are unaware of the database organization, they don't even know about the typologies of underlying databases platform. This is due, as already anticipated, to the fact that the interface presents the available queries, but provides no details about the interactions with the databases. The abstraction achieved is a powerful tool for code maintainability because the database layer of the system can potentially be redesigned from scratch without affecting other components.

The transaction management implemented by the module deserves special mention because it represents one of the most critical functionalities that the system has to manage. The database as explained above are organized in such a way to minimize the need of multiple accesses to complete an operation, however there are some cases where this cannot be avoided and therefore operations are performed within a transaction. If all the operations complete successfully, e.g. the entity is deleted on both databases, then the transaction is committed, otherwise if at least one operation fails the transaction will be aborted and the operations already performed will be reverted via a rollback mechanism. The ability to hide such complex functionality to other modules clearly shows the effectiveness of the division into modules.

Since the adoption of Java technologies and thus a strict version of the object-oriented programming paradigm to simplify the integration between database and application all operations on database returns custom defined objects that resembles the entities of the data model. The mapping between data and

custom object well supported by the database drivers, at least for the part of MongoDB[3], allows to solve without effort for the programmer the problem of the data conversion from the format on database to a format that can be manipulated by the application logic. It's worth to notice that even though entity-to-object mapping is an elegant solution to handle the problem of data representation conversion it may lack flexibility in certain operations. For example the result of queries that compute the statistics, the result won't be a well-defined entity of the database, but just a bunch of information that represent the wanted statistic. In such cases the approach adopted is slightly different because the retrieved data won't be mapped on a custom object, but will be represented by a flexible object that follows the JSON structure.

Due to the complexity and heterogeneity of the functionalities offered by this module, internally it is implemented as a composition of the sub-modules:

- MongoDB sub-module
- Neo4J sub-modules

The set of operation that must be performed are very different due to the different platform, however the sub-modules are able to hide all the details and thus they implement the core of the data access module. The only the task left the main module is only to merge the functions of the two sub-modules, implementing a clear and simple common interface that can be used by the other modules.

In this module it's also implemented the handling of database redundancies to guarantee the consistency of the data. In the following a dedicated section will highlight the flow of operations required to achieve such goal.

Service

The 'Service' module contains the business logic of the application and encapsulates the implementation of the various services offered. In order to make use of it, an interaction based on an interface is adopted, which makes use independent of the actual implementation. A change in the implementation does not, in general, also entail a change in the interface and thus in the code that makes use of it. Moreover, the user sees logical functionalities and not directly the related database operations. This, therefore, makes it possible to abstract from both the implementation and the actual organization and design. The 'Service' module acts as a link between the user, and thus the interface and servlets that handle the user's requests, and the databases with their logic for accessing and interacting with the data. The services offered are arranged in separate files according to the application domain. Services for the exclusive use of one type of user are enclosed in a file dedicated to that type of user. Services shared by several types of users are contained in files dedicated to the entity on which that service operates. In order to transfer the data resulting from the execution of a service, use is made of serializable Data Transfer Objects (DTOs), defined in such a way as to contain the information of interest. In particular, DTOs do not map the entities contained in the databases but are used to contain the information as it is then used in the business logic and for the interface, so as to be more consistent with its use within the application. The use of DTOs instead of entity objects allows easy access to information while maintaining transparency of the database organization. For a simple conversion from database entity to DTO and vice versa, a mapper class was implemented. In the 'Service' module, use is also made of a thread pool to parallelise computation. In particular, this technique is adopted to compute the statistics of an admin user and the components of a reporter page in parallel. This makes it possible to reduce the time for obtaining the content by avoiding the sequencing of operations and operating if necessary on two databases separately in an independent and parallel manner.

[3] Actually, Neo4J drivers support the mapping on custom objects, but they can't guarantee the desired flexibility, so a custom conversion has been implemented.

Servlet

Servlet module is responsible for handling a request coming from the client, returning to the latter a dynamically created web page or encoded data (e.g. in JSON).

Some servlet filters are also implemented in the project. These are applied so that requests received can be logged and we can check that:

- Requests to pages other than the login and signup pages come from registered and logged-in users. If this is not the case, these users will be redirected to the login page.
- Requests to the login and signup page come from users who are not logged in. If this is not the case, these users will be redirected to the homepage.
- Logged user is not trying to access to pages for which they do not have authorisation. If yes, redirect the user to the relative homepage.

Servlets expose endpoints to which to forward the request. The operation to be performed is identified by the URL mapping and request parameters.

The 'Servlet' module makes use of the 'Service' module, which implements the business logic and interaction with the module for accessing data and thus the database. The role of the 'Servlet' module is therefore to act as a bridge, allowing clients to make use of the services offered by interacting with them indirectly. This approach makes the implementation more transparent to the client.

Webapp

The 'Webapp' module contains resources of different types, which are used by the user when navigating the application. In particular, it contains the stylesheet files used for the graphics of the application, the JavaScript files for managing the user's local interaction with the pages, and the Jakarta Server Pages (JSP for short) files representing the pages that the client accesses and whose content is dynamically calculated at the time of the request. Also contained in the module is the deployment descriptor file 'web.xml', which contains various information, including the first page displayed, the Web application display name and the JSP page to be shown to the user in the event of an error. In the 'Webapp' module we can find resources for the realisation of the GUI components of the web application. Through the graphical interface it implements, the user is then able to visualise and interact with the application by using a browser and accessing the specific URL of the application. Some of the requests will be handled asynchronously (e.g. paging) while others are synchronous (e.g. navigation from one page to another). The reason why some requests are handled asynchronously is to achieve a smoother interaction.

14.2 Adopted patterns and techniques

Some aspects of the implementation that were not covered or were only mentioned during the description of the various modules deserve to be highlighted more in detail because they play an important role in the application.

Service-Locator pattern

The service-locator design pattern is extensively used to manage the dependencies and interactions between different modules. In particular, the modules that expose their functionalities via this pattern are the data access and the service modules. The main advantage coming from the adoption of this pattern is the decoupling of the service[4] implementation from its interface. The client of the service is aware only of the interface of that service, it doesn't need any information about the implementation because the instantiation

[4] The term service in this context refers to a generic operation offered by a module, not only to the business logic implemented in the service module.

is a responsibility of the "Service Locator", i.e. the third component specified by the pattern which must take care of the service lookup, given the interface, and returns the actual object that will be used by the client.

Actually the "SocialNews" application doesn't present a complex logic that deserves a properly distribution of the services to handle scalability, so the lookup operation is a simple instantiation of the object that implements the desired interface.

Singleton pattern

The singleton pattern is used by many components of the system that can correctly handle the access by multiple threads at the same time. Some components must be instantiated only once across the entire lifecycle of the application, such as the objects that manage the connections with the databases as specified by their documentation, so the singleton pattern fit very well to ensure the unicity in the application. Other components instead could be theoretically instantiated each time they are required, resulting in multiple instances at the same time, possibly one per running operation, used in different parts of the system. This situation could lead to a huge amount of resources occupied by the same object, so the adoption of the singleton pattern could ensure a more efficient usage of the resource, forcing to reuse the same object shared across all the application each time it's invoked. The components that fit very well the latter possibility are the ones that implements the services offered via the service locator patter, i.e. the DAO objects and the service objects that implement the business logic. Obviously, the single instance may represent a bottleneck, so further analysis may be necessary in case of performance problems.

Parallel execution

The operations associated with a client request are normally executed by the thread that the application server selects from its pool, adopting the thread-per-request approach. Following this approach, the execution of operations will be sequential and the response will be sent to the client as the last step. This execution model works well in most cases because the workload is given by the amount of requests to be served, not by the single request that usually requires few computational resources. So, the parallelization of the requests via the thread pool managed by the application server seems to be sufficient to achieve good results. However, there are still cases where this model is not a good one because may experience a high latency to produce the response. The latency of a response is mainly due to the time needed to traverse the network from client to application server and from application server to database. If the cost is paid once per direction, the latency may be acceptable, but in cases where the response needs multiple interactions with the database to be produced, paying multiple times to traverse the network from the application server to the database, the cost is too high even if the client can adopt asynchronous techniques to mitigate the problem.

The solution in the latter cases is to introduce a custom thread pool, managed directly by the application[5] and invoked whenever multiple access to the database should be performed inside the same thread. A typical example of this situation is the construction of the admin statistic dashboard that requires four multiple queries to build all the statistics shown on the same page. If the queries are executed in parallel the cost to be paid is equivalent to the slowest of the multiple operation executed, greatly reducing the overall latency. The configuration of the pool is critical from a performance point of view and therefore needs to be deeply investigate. However, to minimize the run-time cost of thread spawning the selected pool is a "cached thread pool", i.e. the spawned thread stay alive for a certain amount of time after the execution of the assigned task in order to be reused by subsequent tasks without the need for spawning new threads. The cached thread pool is suitable for executing a large number[6] of short duration tasks, which is exactly the use case for which it was introduced into the system. To keep control over the pool a manager is provided in single instance,

[5] Note that it is not usually advisable to create a custom thread pool within the application server, but Tomcat provides no solution to parallelize operations in response to a request.

[6] The term "large number" strictly depends by the available resources and the overall configuration of the pool.

following the singleton design pattern, ensuring that an unexpected number of incoming requests wouldn't explode the number of active threads, risking a congestion of the entire system.

Finally, the pool is also designed to fulfill another task, related to the execution of tasks triggered by a request, but that are needed for system maintainability and not for building the associated response. In such cases the task is assigned to a new thread in order to avoid an additional waiting by the client. An example of this use case is the submission of the redundancy task to the manager that will update the data periodically, more details can be found on the dedicate section.

14.3 Project organization

The main structure is given by the adopted Java EE technology, i.e. one folder is dedicated to the code that implements the logic executed by the application servers and another folder contains all the resources that defines the client-side behaviour. More in details the first directory will contain all the Java files organized in custom packages, while the second one, named webapp, will contain all files (scripts, styles …) that implement the page displayed by the browser to the end user. The inner structure of the webapp folder doesn't deserve mentioning because it is a simple subdivision of files into subfolders as is appropriate for small websites that are not implemented by adopting a well-defined framework. In contrast, it is interesting to report the organization of the java folder as it provides a more concrete view of the modules described above.

Packages

The structure of the packages on which the Java code is organized closely resembles the modules that were introduced. All the packages belong to a common package root, named "*it.unipi.lsmsd.socialnews*" and then are differentiated following the modules structure.

In the following for each module will be reported the packages that provides the implementation.

- **Configuration**, implemented by the "*config*" package in which can be found the main class bound to the lifecycle of the application and two other utility classes, in a dedicated package, that allow to retrieve the value of environment variables.
- **Data access**, implemented by the "*dao*" package in which reside a large part of the server-side application. In this package is defined the interface used to access the database, split into multiple files in order to enclose all the operations related to an entity in a single object, avoiding the definition of a super-object in which can be found all the available operations. Inside the package there are other sub-packages, as already anticipated there is a package dedicated to the Mongo interaction and one for the Neo4j interactions. The classes within these two packages implement most of the available operations, that will be coordinated to provides the final interfaces. The package also contains the sub-package "*model*" in which are defined the classes that are mapped to the database entities and the package containing the implementation of the mechanism for the redundancy update. Finally in the package can be found a class that implements the locator party of service-locator pattern and two custom exceptions thrown in case of errors.
- **Service**, implemented by the namesake package adopts a division of the interfaces based on system actors plus the post entity because many operations are shared by the actors, abstracting completely from the database organization. The package, other than the implementation of the interfaces, contains a custom exception, the locator party of the service-locator design pattern and a utility sub-package in which can be found some enumerates used to simplify the code readability and a class containing all the methods capable of an automatic conversion between model entity coming from data access module and DTO objects used as return object by the service module. The implementation of the DTO is not within the service package but reside in an external dedicated package in order to reduce dependencies between packages that use such classes.

- **Servlet**, implemented by the namesake package follow a division in sub-package by the actors of the system. The division is not so strict because many endpoints can be used by more than one actor, so in such cases no sub-package has been defined. The rule followed in broad terms is to dedicate one servlet for each page of the web application, but in some cases the rule is bypassed to guarantee a better organization of the code.

There are other packages that can't be reconducted to the implementation of one specific module because they implement system functionalities that are not foreseen in the module division. In such category there are the "*threading*" and "*listener*" package. The former implements the custom thread pool deeply discussed above, while the latter implements operations that must be executed when a client session is destroyed, such as the system check on the maximum dimension of the reporter collection on Mongo database.

Main classes

In this section a brief description of the main classes implemented, grouped by the package which they belong. The interfaces won't be listed, but the description of the relative implementation will highlight which interface they implement.

CONFIG		
Class	**Sub-Package**	**Brief description**
ConfigListener	-	Implements initialization and shutdown actions, its methods are bound to the application lifecycle
MongoEnvironment	environment	Retrieves and stores the Mongo configuration from environment variables
Neo4JEnvironment	environment	Retrieves and stores the Neo4J configuration from environment variables

Table 6- Config package

DAO		
Class	**Sub-Package**	**Brief description**
MongoConnection	mongodb	Open the connection with the Mongo cluster and offers the possibility to other classes to get a reference to issue queries. The class adopts the singleton pattern to avoid multiple connections to the database[7]
MongoDAO	mongodb	Abstract class from which derives all the Mongo DAO classes, it implements the common operations needed to select the appropriate collection customized via the "generics" mechanism provided by Java.
MongoAdminDAO	mongodb	Implements all the operations regarding the admin database entity on MongoDB
MongoReporterDAO	mongodb	Implements all the operations regarding the reporter database entity on MongoDB
MongoReaderDAO	mongodb	Implements all the operations regarding the reader database entity on MongoDB

[7] The Mongo drivers already offer a connection pool toward the database, for this reason is a good practise to avoid multiple instances of such drivers.

MongoPostDAO	mongodb	Implements all the operations regarding the post database entity on MongoDB
MongoCommentDAO	mongodb	Implements all the operations regarding the comment database entity on MongoDB
Neo4jConnection	neo4j	Open the connection with the Neo4J database and offers the possibility to other classes to get a reference to issue queries. The class adopts the singleton pattern to avoid multiple connections to the database[8]
Neo4jReporterDAO	neo4j	Implements all the operations regarding the reporter database entity on Neo4J
Neo4jReaderDAO	neo4j	Implements all the operations regarding the reader database entity on Neo4J
Neo4jPostDAO	neo4j	Implements all the operations regarding the post database entity on Neo4J
Neo4jReportDAO	neo4j	Implements all the operations regarding the report database entity on Neo4J
AdminDAOImpl	implement	Implements the AdminDAO interface by providing all operations related to the admin database entity, leveraging the operations of MongoAdminDAO
ReporterDAOImpl	implement	Implements the ReporterDAO interface by providing all operations related to the reporter database entity, merging the operations of MongoReporterDAO and Neo4jReporterDAO.
ReaderDAOImpl	implement	Implements the ReaderDAO interface by providing all operations related to the reader database entity, merging the operations of MongoReaderDAO and Neo4jReaderDAO.
PostDAOImpl	implement	Implements the PostDAO interface by providing all operations related to the post database entity, merging the operations of MongoPostDAO and Neo4jPostDAO.
CommentDAOImpl	implement	Implements the CommentDAO interface by providing all operations related to the comment database entity, merging the operations of MongoCommentDAO and the redundancy package
ReportDAOImpl	implement	Implements the ReportDAO interface by providing all operations related to the report database entity, merging the operations of Neo4jReportDAO and the redundancy package
BaseEntity	model	Empty class used as common root for all entities hierarchy

[8] The Neo4J drivers already offer a connection pool toward the database, for this reason is a good practise to avoid multiple instances of such drivers.

User	model	Contains all the attributes common to all types of users in the system and their getters and setters
Admin	model	Extends the parent class User providing admin unique attributes and their getters and setters
Reporter	model	Extends the parent class User providing reporter unique attributes, including the list of Post and their getters and setters
Reader	model	Extends the parent class User providing reader unique attributes and their getters and setters
Post	model	Contains all post attributes and their getters and setters
Comment	model	Contains all comment attributes and their getters and setters
Report	model	Contains all report attributes and their getters and setters
RedundancyUpdater	redundancy	Contains all the operations required to manage the logging of redundancy operations and the updating of redundancies on databases.
RedundancyTask	redundancy	Implements the Serializable interface and represents the performed operation, the quantity, and the identification of the interested entity
TaskType	redundancy	Enumerate that represents the types of operation performed on post and reporter that are of interest for redundancies attributes
SocialNewsDataAccessException	exception	Custom exception thrown in case of errors during the execution of the queries
SocialNewsRedundancyTaskException	exception	Custom exception thrown in case of errors during the redundancy update
DAOLocator	-	Implements the locator party of service-locator pattern

Table 7 - Dao package

SERVICE		
Class	**Sub-Package**	**Brief description**
AdminServiceImpl	implement	Implements the AdminService interface, providing all the available admin's operations (excluding the one related to posts)
ReporterServiceImpl	implement	Implements the ReporterService interface, providing all the available reporter's operations (excluding the one related to posts)
ReaderServiceImpl	implement	Implements the ReaderService interface, providing all the available reader's operations (excluding the one related to posts)
PostServiceImpl	implement	Implement the PostService interface, providing all the available operations on the posts

Util	util	Implements a set of utility methods such as automatic conversion from entity classes and DTO classes and vice-versa
Page	util	Enumerate that represents the navigation direction for the pagination mechanism
Statistic	util	Enumerate that represents the desired statistics to be computed
SocialNewsServiceException	exception	Custom exception thrown in case of errors during the execution of a service
ServiceLocator	-	Implements the locator party of service-locator pattern

Table 8- Service package

DTO		
Class	**Sub-Package**	**Brief description**
BaseDTO	-	Empty class used as common root for all DTOs hierarchy
UserDTO	-	Contains all the attributes common to all types of users in the system and their getters and setters
AdminDTO	-	Extends the parent class UserDTO providing admin unique attributes and their getters and setters
ReporterDTO	-	Extends the parent class UserDTO providing reporter unique attributes and their getters and setters
ReaderDTO	-	Extends the parent class UserDTO providing reader unique attributes and their getters and setters
PostDTO	-	Contains all post attributes and a reference to the author and their getters and setters
CommentDTO	-	Contains all comment attributes and their getters and setters
ReportDTO	-	Contains all report attributes and their getters and setters
ReporterPageDTO	-	Contains the ReporterDTO, list of PostDTO, the number of followers and a flag that determines if the current reader already follows the target reporter
StatisticPageDTO	-	Contains the information needed to build the admin dashboard
JSONConverter	util	Implements an automatic conversion from DTO class to JSON formatted string and vice-versa

Table 9 - Dto package

SERVLET		
Class	**Sub-Package**	**Brief description**
LoginServlet	-	Implements login endpoint
LogoutServlet	-	Implements logout endpoint
SignUpServlet	-	Implements signup endpoint
AuthenticationFilter	-	Implements authentication check
RequestLoggingFilter	-	Implements the log of received requests
UnauthorisedAccessFilter	-	Implements checks to prevent unauthorised access to resources
ReporterPageServlet	-	Implements reporter homepage endpoint for reader and admin users
PostHandlingServlet	-	Implements the reporter posts paging, insertion and deletion functions endpoint

CommentHandlingServlet	-	Implements the reader comments paging, insertion and deletion functions endpoint
HomepageServlet	admin	Implements admin homepage endpoint
UsersServlet	admin	Implements the admin user lists endpoints
AddReporterServlet	admin	Implements the admin add report function endpoints
DashboardServlet	admin	Implements the admin dashboard endpoints
ReportServlet	admin	Implements the admin report handling endpoints
HomepageServlet	reporter	Implements the reporter homepage endpoint for reporter user itself
StatisticsServlet	reporter	Implements the reporter statistics endpoints
HomepageServlet	reader	Implements the reader homepage endpoint
ProfileServlet	reader	Implements the reader profile endpoint
SearchServlet	reader	Implements the reader posts/reporters search endpoint
StatisticsServlet	reader	Implements the reader statistics endpoints
ReportPostServlet	reader	Implements the reader post reporting endpoint

Table 10 - Servlet package

THREADING		
Class	**Sub-Package**	**Brief description**
ServiceWorkerPool	-	Implements the thread pool manager, offering methods to submit tasks on the pool

Table 11 - Threading package

LISTENER		
Class	**Sub-Package**	**Brief description**
SessionListener	-	Web listener tied to the destruction of the client session that fires the max size collection handling system functionality

Table 12 - Listener package

14.4 Queries

In the following the code of the most relevant queries will be reported, explaining the step by step how the final result is obtained.

Document database

The majority of the available operations concern the document database, but many of them are simple CRUD operations that don't need a detailed analysis. However, other operations are quite complex, especially the ones that retrieve statistics data, so a deeper look is needed for the comprehension of the operations performed.

Most active readers

The most active readers query should return an ordered list of readers, ranked according to the number of comments published in the last period, i.e. since a past threshold instant. The first stage of the aggregation is the filtering out of the comments published before the specified threshold, reducing dramatically the amount of data to be processed in subsequent stages. Then the remaining comments are grouped by the reader, outputting for each group the "fullName" of the considered reader and the number of comments that are part of the group, obtained via the count operator to perform the aggregation. Finally, the groups are ordered by the number of comments and the top 10 readers are returned along with the comment counter.

```
db.comments.aggregate([
  {$match:{'timestamp':{$gte:threshold}}},
  {$group:{_id:'$reader._id',
      fullName:{$first:'$reader.fullName'}, numOfComment:{$count:{}}}},
  {$sort:{numOfComment:-1}},
  {$limit:10}
])
```

Figure 10 - Most active readers query language

```
try{
    List<Bson> stages = new ArrayList<>();
    stages.add(Aggregates.match(Filters.gte("timestamp", from)));
    stages.add(Aggregates.group(
           "$reader._id",
            Accumulators.first("fullName","$reader.fullName"),
            Accumulators.sum("numOfComment",1)));
    stages.add(Aggregates.sort(Sorts.descending("numOfComment")));
    stages.add(Aggregates.limit(topN));
    stages.add(Aggregates.project(Projections.exclude("_id")));

    List<Document> docs = new ArrayList<>();
    getRawCollection("comments").aggregate(stages).into(docs);

    return new ObjectMapper().valueToTree(docs);
}
catch (MongoException me){ /* error handling*/ }
```

Figure 11 - Most active readers Java code

Gender statistic

The gender statistic groups the readers into three categories: “male”, “female” and “other” in case no info is given during the signup phase. The implemented aggregation firstly filters the admin user from the readers, checking the “isAdmin” attribute. Then the bucket stage is responsible to create three buckets, determined by numeric boundaries, each assigned to one of the desired categories. To assign each reader to the correct bucket the “gender” attribute is mapped into a number via switch operator. The bucket stage outputs for each bucket the gender label and the sum of readers that fall into that category. Note that the bucket stage has been preferred to the group one, even if it could simplify the query from a coding point of view, because the desired result is composed by only the three categories specified. Using the group stage would not be possible to summarize multiple labels into the “other” label.

```
db.users.aggregate([
  {$match:{'isAdmin':{$exists:false}}},
  {$bucket: {
    groupBy: {'$switch': {branches: [
      {case: {'$eq':[{$toLower: '$gender'}, 'male']}, then: 0 },
      {case: {'$eq':[{$toLower: '$gender'}, 'female']}, then: 1 }],
        default: -1}},
    boundaries: [0, 1, 2], default:-1,
    output: {
      'count': { $sum: 1 },
      'gender': {$first: {$cond: {if:{$in:[{$toLower:'$gender'},
                ['male','female']]}, then:'$gender', else:'Other'}}}}}},
  {$project:{_id:0}}
])
```

Figure 12 - Gender statistic query language

```
try{
    List<Bson> stages = new ArrayList<>();
    stages.add(Aggregates.match(Filters.exists("isAdmin", false)));
    stages.add(Aggregates.bucket(Document.parse(
        "{$switch: { branches: [" +
        "    {case: {$eq:[{$toLower: '$gender'}, 'male']}, then: 0 }," +
        "    {case: {$eq:[{$toLower: '$gender'}, 'female']}, then: 1 }]," +
        "    default: -1}}}"),
        List.of(0,1,2),
        new BucketOptions().defaultBucket("-1").output(List.of(
            new BsonField("count", Document.parse("{ $sum: 1 }")),
            new BsonField("gender", Document.parse("{$first: {$cond: " +
                    "{if:{$in:[{$toLower:'$gender'},['male','female']]}, " +
                    "then:'$gender', else:'Other'}}}"))))));
    stages.add(Aggregates.project(Projections.exclude("_id")));

    List<Document> docs = new ArrayList<>();
    getRawCollection("users").aggregate(stages).into(docs);
    ObjectMapper mapper = new ObjectMapper();
    ObjectNode obj = mapper.createObjectNode();

    for(Document doc : docs){
        obj.put(doc.getString("gender"), doc.getInteger("count"));
    }
    return obj;
}
catch (MongoException me){ /* error handling */ }
```

Figure 13 - Gender statistic Java code

Nationality statistic

The nationality statistic on contrary can be easily implemented via the group stage, preceded by the same filtering to remove admins from the statistic. The group stage sum together all the readers that share the same “country” attribute, analysed after the application of the operator toLower to avoid annoying possible duplicates of nationality and outputs for each country the counter of readers that reside to that country. Finally, the result is sorted by counter in descending order.

```
db.users.aggregate([
    {$match:{'isAdmin':{$exists:false}}},
    {$group: {_id: {$toLower:'$country'},
        'country':{$first:'$country'},'count': { $sum: 1 }}},
    {$project:{_id:0}},
    {$sort:{'count':-1}}
])
```

Figure 14 - Nationality statistic query language

```
try{
    List<Bson> stages = new ArrayList<>();
    stages.add(Aggregates.match(Filters.exists("isAdmin", false)));
    stages.add(Aggregates.group(Document.parse("{$toLower: '$country'}"),
            Accumulators.first("country","$country"),
            Accumulators.sum("count",1)));
    stages.add(Aggregates.project(Projections.exclude("_id")));
    stages.add(Aggregates.sort(Sorts.descending("count")));

    List<Document> docs = new ArrayList<>();
    getRawCollection("users").aggregate(stages).into(docs);
    return new ObjectMapper().valueToTree(docs);
}
catch (MongoException me){ /* error handling */ }
```

Figure 15 - Nationality statistic Java code

Hot post in the period

The hot post in the period retrieves the most commented post in the last period among the one published by a target reporter. The first stage keeps all the documents related to target reporter and having at least one post, reducing dramatically the amount of data to handle by subsequent stages. Note that more than one document may be returned by the first stage, according to the mechanism for managing the maximum size of a document. The subsequent stage is responsible for keeping for each document only the “reporterId” attribute and the posts that present at least one comment, deducted leveraging the redundancy “numOfComment”, and published after the threshold in time specified. Then the documents are merged grouping on the common “reporterId”, unique for construction among the documents under analyses, and pushing all the post arrays (one for document) into the same array via the push operator, resulting in a document composed by the reporterId and an array of arrays containing all the selected posts. Finally, the array of selected posts is reduced applying the concatArrays operator in order to obtain a plain array of posts and ordered by number of comments. The top 10 posts will be returned as result.

```
db.reporters.aggregate([
  {$match:{$and:[{reporterId:reporterId},{posts:{$exists: true}}]}},
  {$project:{_id: 0, reporterId:1, posts: {"$filter":
      {input: "$posts", as: 'posts', cond: {$and:[
          {$gte: ['$$posts.timestamp', threshold]},
          {$gte: ["$$posts.numOfComment", 1]}]}}}}},
  {$group:{_id:'$reporterId', reporterId:{$first:'$reporterId'},
      posts:{$push:'$posts'}}},
  {$project: {
    reporterId:1,
    posts: {
      $filter: { input: {
          $sortArray:{ input: {
              $reduce: {
                input:'$posts', initialValue: [],
                in: {$concatArrays: ['$$value', '$$this']}}},
            sortBy:{'numOfComment':-1, timestamp:-1, _id:-1}}},
    cond:{}, limit:10}}}
])
```

Figure 16 - Hot post in the period query language

```
try{
    List<Bson> stages = new ArrayList<>();

    stages.add(Aggregates.match(Filters.and(
            Filters.eq("reporterId", reporterId),
            Filters.exists("posts", true)))
    );
    stages.add(Aggregates.project(Projections.fields(
            Projections.excludeId(),
            Projections.include("posts", "reporterId"),
            Projections.computed("posts",Document.parse(String.format(
                "{$filter: {" +
                    "input: '$posts'," +
                    "as: 'posts'," +
                        "cond: {$and:[" +
                        "{$gte: ['$$posts.timestamp', new Date(%d)]}, " +
                        "{$gte: ['$$posts.numOfComment', 1]}]}}}",
from.getTime()
            ))))));
    stages.add(Aggregates.group("$reporterId",
            Accumulators.first("reporterId","$reporterId"),
            Accumulators.push("posts","$posts")));
    stages.add(Aggregates.project(Projections.fields(
            Projections.excludeId(),
            Projections.include("reporterId"),
            Projections.computed("posts", Document.parse(String.format(
                "{$filter: {" +
                "input:{" +
                "    $sortArray:{" +
                "      input:{$reduce: {input:'$posts', initialValue: []," +
                "          in: {$concatArrays: ['$$value', '$$this']}}}," +
                "      sortBy:{'numOfComment':-1, timestamp:-1, _id:-1}}}," +
                "cond:{}, limit: %d}}", nTop)))))
    );

    Reporter reporter = getCollection().aggregate(stages).first();

    if(reporter == null)
        return new ArrayList<>();// Empty list
    else
        return reporter.getPosts();
}
catch (MongoException me){ /* error handling */ }
```

Figure 17 - Hot post in the period Java code

Most active moment of the day

The most active moment of the day is a query that clearly demonstrates the powerful of the MongoDB query language by constructing a histogram in which the buckets represent time windows over a day, such as a 3-hour window, and the associated values are the number of comments published in the interval each day from a selected time threshold. The first stage of the aggregation filters out all the comments published before the time threshold, dramatically reducing the amount of data handle by the subsequent stages. Then the bucket stage groups into the correct time window each comment, discriminating by the hour extracted from the timestamp via the hour operator. For each bucket the stage outputs the number of comments that fall within it, applying the sum operator. Then to ensure that each time window is part of the final document, even if no comment falls in that window, a densify stage plus a set stage are added to the aggregation. The

former adds the missing windows to the list of windows into the produce document, while the latter adds the attribute “count” sets its value to zero for artificially added windows. Finally, a project stage is responsible for producing a document in which each element is an object containing upper\lower bounds and the counter of comments, producing an information ready processed by the application without further manipulation.

Note that the buckets are hard coded in the query language example, however the actual implementation in Java code allows to customize the buckets size.

```
db.comments.aggregate([
  {$match:{'timestamp':{$gte:threshold}}},
  {$bucket: {groupBy: {$hour:'$timestamp'},
    boundaries: [0,6,12,18,24], default:-1,
    output: {'count': { $sum: 1 }}}
  },
  {$densify:{field:'_id', range:{step:6, bounds:[0,24]}}},
  {$set: {count: {$cond: [{$not: ['$count']}, 0, '$count']}}},
  {$project: {_id: 0, 'count': 1, 'lowerBound': {$mod:['$_id',24]},
      'upperBound':{$mod:[{$add:['$_id',6]}, 24]}}}
])
```

Figure 18 - Most active moment of the day query language

```
try{
    List<Integer> boundaries = new ArrayList<>();
    IntStream.iterate(0, n -> n + windowSize).limit(24/ windowSize + 1)
            .forEach(boundaries::add);

    List<Bson> stages = new ArrayList<>();
    stages.add(Aggregates.match(Filters.gte("timestamp", from)));
    stages.add(Aggregates.bucket(
            Document.parse("{$hour:'$timestamp'}"),
            boundaries,
            new BucketOptions().output(
                    new BsonField("count", Document.parse("{ $sum: 1 }")))));
    stages.add(Aggregates.densify(
        "_id", DensifyRange.rangeWithStep(0,24, windowSize))
    );
    stages.add(Aggregates.set(new Field<>("count",
        Document.parse("{$cond: [{$not: ['$count']}, 0, '$count']}")))
    );
    stages.add(Aggregates.project(Projections.fields(
        Projections.exclude("_id"),
        Projections.include("count"),
        Projections.computed("lowerBound",
            Document.parse("{$mod:['$_id',24]}")),
        Projections.computed(
            "upperBound",
            Document.parse(String.format("{$mod:[{$add:['$_id',%d]},24]}",
            windowSize))
    ))));

    List<Document> docs = new ArrayList<>();
    getRawCollection("comments").aggregate(stages).into(docs);

    return new ObjectMapper().valueToTree(docs);
}
catch (MongoException me){ /* error handling */ }
```

Figure 195 - Most active moment of the day Java code

Pagination

All the queries that can produce a large amount of data as result are implemented in such a way to paginate the result, i.e. the user can navigate across pages loading only a small number of items each time. This approach is necessary not only because allows to improve the response time for the client, but even because wouldn't be feasible to download a consistent part of the database each time a request that regards all items of an entity is issued.

In mongoDB a standard way to realize such functionality is the adoption of the "skip-limit" approach in which an offset determines how many items to skip, the ones already loaded by the client, and limit the subsequent items to the desired page size. This approach suffers from a performance point of view because even if only a small number of items are returned for each request, the query internally has to select all the items before applying the skip and limit mechanism. To overcome this problem is possible to adopt a slightly more complex approach based on filtering a field able to ensure a unique ordering of the items, such as a timestamp. The items are first filtered based on the value of an offset given by ordering field of the last item in the previous page and then it's possible to apply the limit to return only a desired quantity of items. To better understand such mechanism the query to retrieve the next page of posts given a reporter id will be discussed in the following.

The query must return a list of posts ordered starting from the most recent, published by a target reporter before a given temporal offset. The aggregation is composed by a preliminary filter that keeps only the documents that contains data about the target reporter and the "posts" attribute. Then the posts published after the timestamp offset, is given by the last item in the previous page, embedded in the remaining documents are filtered out. Note that two or more posts may accidentally present to same timestamp, so in such cases the ordering is given by the identifier, that is guaranteed to be unique. The filter allows to reduce the amount of data to be processed in the subsequent stages. The remaining stages are due to the fact that multiple documents related to the same reporter can be returned and thus it's necessary to merge them into a single document, applying the same grouping and reduction already discussed. The last operation needed to complete the pagination mechanism is the limit of number of items returned after their ordering by timestamp.

If the previous page must be returned, then it's necessary to apply some modification to the query:

- The offset is given by the first item in the next page
- The filtering criteria must be reverted
- The limit must be applied to a reverse ordered array, that will be correctly ordered after the reduction of the number of items

```
db.reporters.aggregate([
  {$match: {$and:[
      {reporterId: targetReporterId},
      {'posts.timestamp': {$exists: true}}]}
  },
  {$project:{ reporterId:1,
    posts: {"$filter":
      {input: "$posts", as: "posts", cond:
        {"$or": [
          {"$and": [{"$eq": ["$$posts.timestamp", postOffsetNext.timestamp]},
                  {"$lt": ["$$posts._id", postOffsetNext.id]}]},
          {"$lt": ["$$posts.timestamp", postOffsetNext.timestamp]}]}}}}},
  {$group:{_id:'$reporterId', posts:{$push: '$posts'}}},
  {$project: {reporterId:1,
    posts:{$filter: {
      input:{
        $sortArray:{
          input:{$reduce: {input:'$posts', initialValue: [],
              in: {$concatArrays: ['$$value', '$$this']}}},
          sortBy:{'timestamp':-1}}},
      cond:{}, limit:pageSize}}}
])
```

Figure 20 - Posts by reporter id with pagination (next page) query language

```
try{
    List<Bson> stages = new ArrayList<>();
    stages.add(Aggregates.match(Filters.and(
            Filters.eq("reporterId", reporterId),
            Filters.exists("posts", true))));

    Bson filter = Document.parse(String.format(
        "{$filter: {" +
            "input: '$posts'," +
            "as: 'posts'," +
            "cond: {" +
            "   $or: [" +
            "       {$and: [{$eq:['$$posts.timestamp', new Date(%d)]}," +
            "                 {$lt:['$$posts._id', '%s']}]}," +
            "       {$lt: ['$$posts.timestamp', new Date(%d)]}]}}}",
        offset.getTimestamp().getTime(), offset.getId(),
        offset.getTimestamp().getTime()));

    stages.add(Aggregates.project(Projections.fields(
            Projections.include("reporterId"),
            Projections.computed("posts", filter))));

    stages.add(Aggregates.group("$reporterId",
            Accumulators.push("posts", "$posts")));
    stages.add(Aggregates.project(
        Projections.fields(
            Projections.include("reporterId"),
            Projections.computed("posts", Document.parse(String.format(
                "{$filter: {" +
                    "input:{" +
                    "   $sortArray:{" +
                    "     input:{$reduce: {input:'$posts',initialValue:[]," +
                    "       in: {$concatArrays: ['$$value', '$$this']}}}," +
                    "     sortBy:{'timestamp':-1}}}," +
                    "cond:{}, limit:%d}}}", pageSize))))));

    Reporter reporter = getCollection().aggregate(stages).first();
    return reporter == null ? new ArrayList<>():reporter.getPosts();
}
catch (MongoException me){ /*error handling */ }
```

Figure 21 - Posts by reporter id with pagination (next page) Java code

Graph database

Below are some of the queries previously described. Only the main, more complex operations are reported and explained in detail here. The missing operations can be easily understood from what is explained below, either by similarity or because they are extremely simple.

Reporting a reporter's post

The query under consideration allows us to report a post owned by a reporter. In particular, we first locate in the database the node representing the reporter who owns the post. The node associated with the reporter must already be present in the graph and is uniquely identified by its identifier. Once this has been done, we go on to search for the node representing the reader who wants to publish a report and the node representing the reported post owned by the aforementioned reporter. If one or both of the reader and post nodes are not present, we will go on to create the missing ones, also specifying their unique identifier (information contained in MongoDB). If, on the other hand, the nodes are present, we will match them. This method of introducing nodes implements the "lazy insertion" we will explain later. Reader and post nodes

are inserted into Neo4J when they are needed. Finally, once the necessary nodes have been matched, we create a relationship between the reader and the post in the last step. The report type relationship will contain information about the report (Id, timestamp and text). To execute this query via the java drivers, we make use of "writeTransaction" and manipulate the result by reading the contents of the counters and returning the number of relations created. Transaction functions are also able to handle connection problems and transient errors using an automatic retry mechanism.

```
MATCH (reporter:Reporter {reporterId: <reporterId>})
MERGE (post:Post {postId: <postId>}) <-[:WRITE]- (reporter)
MERGE (reader:Reader {readerId: <readerId>})
CREATE (reader) -[report:REPORT {reportId:<reportId>, timestamp: <timestamp>,
        text: <text>}]-> (post)
```

Figure 22 - Report a reporter's post, query language

```
try(Session session = neo4jConnection.getNeo4jSession()){
    Query query = new Query(
        "MATCH (reporter:Reporter {reporterId: $reporterId})" +
        "MERGE (post:Post {postId: $postId}) <-[:WRITE]- (reporter) "+
        "MERGE (reader:Reader {readerId: $readerId}) " +
        "CREATE (reader) -[report:REPORT {reportId:$reportId, timestamp:
                $timestamp, text: $text}]-> (post) ",
        parameters(
            "readerId", readerId,
            "reporterId", reporterId,
            "postId", postId,
            "reportId", report.getReportId(),
            "timestamp", report.getTimestamp().getTime(),
            "text", report.getText())
    );

    return session
        .writeTransaction(tx -> tx.run(query))
        .consume().counters().relationshipsCreated();
}
catch (Exception e){/*Exception handling */}
```

Figure 23 - Report a reporter's post, Java code

Get number of a reporter's followers

The query under consideration makes it possible to identify the number of followers of a reporter and also to establish whether the user requesting it is one of his followers. In particular, we first of all identify in the database the node representing the reporter of interest and the relative "FOLLOW" relations with "Reader" nodes. The node associated with the reporter must already be present in the graph and is uniquely identified by its identifier. Since this information is especially useful when a user views the page of a reporter, once the first match has been made, we check whether the node representing the current user has a "FOLLOW" relationship with the reporter of interest. If this is verified, it would mean that the user is a reader and also one of the followers of the reporter. The use of "OPTIONAL MATCH" is justified by the fact that the latter relationship may or may not be present, but this must not affect the calculation of the number of followers. When compared to the traditional 'MATCH', the difference is that if no matches are found, OPTIONAL MATCH will use a null for missing parts of the pattern. Finally, two values are returned: the first counting the number

of 'FOLLOW' relationships found in the first point, the second counting the 'FOLLOW' relationship with the current user (we obtain 0 or 1 which can be interpreted as 'false' and 'true' respectively). To perform the count we go to use the 'count' function. To execute this query via the java drivers, we make use of 'readTransaction'. Transaction functions are also able to handle connection problems and transient errors using an automatic retry mechanism.

```
MATCH (reporter:Reporter {reporterId: <reporterId>}) <-[:FOLLOW]-
      (reader:Reader)
OPTIONAL MATCH (reader:Reader {readerId: <readerId>}) -[follow:FOLLOW]->
               (reporter)
RETURN count(reader) as numFollowers, count(follow) as follow
```

Figure 24- Number of reporter's followers, query language

```
try(Session session = neo4jConnection.getNeo4jSession()){
    Query query = new Query(
        "MATCH (reporter:Reporter {reporterId: $reporterId}) <-[:FOLLOW]-
               (reader:Reader) " +
        "OPTIONAL MATCH (reader:Reader {readerId: $readerId}) -
                  [follow:FOLLOW]-> (reporter) " +
        "RETURN count(reader) as numFollowers, count(follow) as follow",
        parameters(
            "reporterId", reporterId,
            "readerId", readerId)
    );

    Record result = session.readTransaction(tx -> tx.run(query).single());
    /*Result manipulation and return*/
}
catch (Exception e) {/*Exception handling */}
```

Figure 25 - Number of reporter's followers, Java code

Get most popular reporters

The query under consideration allows us to identify a ranking containing the <limit> (parameter) most popular reporters, based on the number of their followers. First, we identify the representative nodes of the reporters within the database. Next, using 'OPTIONAL MATCH' we go on to identify any 'FOLLOW' relationships that link these 'Reporter' nodes to other nodes. These relationships may or may not be present. The next step in the query is to calculate the number of followers associated with each reporter. Finally, since we are only interested in obtaining a ranking of the <limit> most popular, the results (reporter - follower pairs) are returned, which are then sorted by descending order of followers, truncating the result at <limit> records. In the event of a tie, the results are arbitrarily sorted.

The use of 'OPTIONAL MATCH' is justified by the fact that we want to be able to return reporters who do not yet have any followers, should the need arise. To go and execute this query via the java drivers, we make use of "readTransaction". Transaction functions are also able to handle connection problems and transient errors using an automatic retry mechanism.

```
MATCH (reporter:Reporter)
OPTIONAL MATCH (reporter) <-[follow:FOLLOW]- ()
WITH reporter, count(follow) as numFollowers
RETURN reporter, numFollowers
ORDER BY numFollowers DESC
LIMIT <limit>
```

Figure 26 - Most popular reporters, query language

```
try(Session session = neo4jConnection.getNeo4jSession()){
    Query query = new Query(
            "MATCH (reporter:Reporter) " +
            "OPTIONAL MATCH (reporter) <-[follow:FOLLOW]- () "+
            "WITH reporter, count(follow) as numFollowers " +
            "RETURN reporter, numFollowers "+
            "ORDER BY numFollowers DESC " +
            "LIMIT $limit",
            parameters("limit", limitTopRanking));

    List<JsonNode> result = session.readTransaction(tx -> {
        Result queryResult = tx.run(query);
        /*Result manipulation*/
    });
    /*Result manipulation and return*/
}
catch (Exception e) {/*Exception handling */}
```

Figure 27 - Most popular reporters, Java code

Get reports of a reporter

The query under consideration makes it possible to identify the reports associated with posts by a particular reporter. Specifically in this case, the reports are paginated and then returned in blocks of at most <limit> records. First of all we identify, by means of a "MATCH", the nodes representing reporters, those representing posts and those representing readers (with the corresponding "WRITE" and "REPORT" relations). Then filter the reporter nodes to only the node representing the reporter of interest, identified via its "reporterId". Specifying this condition in the "MATCH" or in a "WHERE" clause makes no semantic or performance difference (given the presence of the query optimiser). The returned result is then finally composed of triads reader id - report (with id, timestamp and text information) - post id. These triads, to implement pagination, are ordered by timestamp of the reports and the pages are identified by <offset>, offset of reports to be discarded because they belong to previous pages, and <limit>, maximum limit of reports for each page. To execute this query via the java drivers, we make use of 'readTransaction'. Transaction functions are also able to handle connection problems and transient errors using an automatic retry mechanism.

```
MATCH (reporter:Reporter) -[:WRITE]-> (post:Post) <-[report:REPORT]-
      (reader:Reader)
WHERE reporter.reporterId = <reporterId>
RETURN reader.readerId as readerId, report, post.postId as postId
ORDER BY report.timestamp DESC
SKIP <offset>
LIMIT <limit>
```

Figure 268 - Reports by reporter, query language

```
try(Session session = neo4jConnection.getNeo4jSession()){
    Query query = new Query(
        "MATCH (reporter:Reporter) -[:WRITE]-> (post:Post) <-
               [report:REPORT]- (reader:Reader) " +
        "WHERE reporter.reporterId = $reporterId " +
        "RETURN reader.readerId as readerId, report, post.postId as postId "+
        "ORDER BY report.timestamp DESC " +
        "SKIP $offset " +
        "LIMIT $limit",
        parameters(
            "reporterId", reporterId,
            "offset", offset,
            "limit", limit));

    return session.readTransaction(tx ->
        tx.run(query).list( record -> {/*Result manipulation*/})
    );
}
catch (Exception e) {/*Exception handling */}
```

Figure 29 - Reports by reporter, Java code

Suggest reporters to a reader

The query under consideration makes it possible to identify reporters to be suggested to the reader requesting them. Based on what has already been specified in the query that returns the most popular reporters, what differs this query from the one previously described is the need to specify a "WHERE" clause to check that the reader is not already a follower of the reporter. In fact, as previously specified, we want to suggest to the reader a set of the most popular <limit> reporters that it does not yet follow. To perform this query via the java drivers, we make use of "readTransaction". Transaction functions are also able to handle connection problems and transient errors using an automatic retry mechanism.

```
MATCH (reporter:Reporter)
WHERE NOT (:Reader {readerId: <readerId>}) -[:FOLLOW]-> (reporter)
OPTIONAL MATCH (reporter) <-[follow:FOLLOW]-()
WITH reporter as suggestedReporters, count(follow) as NumFollower
RETURN suggestedReporters
ORDER BY NumFollower DESC
LIMIT <limit>
```

Figure 30 - Suggest reporters to a reader, query language

```
try(Session session = neo4jConnection.getNeo4jSession()){
    Query query = new Query(
        "MATCH (reporter:Reporter) " +
        "WHERE NOT (:Reader {readerId: $readerId}) -[:FOLLOW]-> (reporter) "+
        "OPTIONAL MATCH (reporter) <-[follow:FOLLOW]-() " +
        "WITH reporter as suggestedReporters, count(follow) as NumFollower "+
        "RETURN suggestedReporters " +
        "ORDER BY NumFollower DESC " +
        "LIMIT $limit",
        parameters("readerId", readerId, "limit", limitListLen));

    return session.readTransaction(tx ->
        tx.run(query).list( record -> /*Result manipulation*/)
    );
}
catch (Exception e) {/*Exception handling */}
```

Figure 31 - Suggest reporters to a reader, Java code

Delete a reporter

The query under consideration allows a reporter to be deleted from the Neo4J database. With it, the related posts, the writing relationships that join them and the associated reports must also be deleted. In particular, there are precedence constraints in deletion whereby a node can only be deleted when it is not involved in any relationship with another node. It is therefore necessary to go and delete relationships before nodes. In the query under consideration we first identify the node representing the reporter of interest. Next, any pending relationships with other nodes 'Post' or 'Reader' are identified. Since the posts owned by the reporter will also be deleted, it is necessary to perform a check on these nodes as well, making sure that these nodes in turn are not involved in "REPORT" type relationships with "Reader" nodes. The presence of relationships of these types is not taken for granted, hence the choice of using "OPTIONAL MATCH", which in the case of a relationship not present, allows to maintain the elements already matched. Finally, it is necessary to first delete any pending relationships (report, follow and write) and then the post and reporter nodes. As a result, the number of deleted nodes is returned, making use of the statistical counters associated with the query execution. To execute this query via the java drivers, we make use of "writeTransaction". Transaction functions are also able to handle connection problems and transient errors using an automatic retry mechanism.

```
MATCH (r:Reporter {reporterId: <reporterId>})
OPTIONAL MATCH (r) -[w:WRITE]-> (p:Post)
OPTIONAL MATCH (p) <-[report]- ()
OPTIONAL MATCH (r) <-[follow]- ()
DELETE report
DELETE follow
DELETE w
DELETE p
DELETE r
```

Figure 32 - Delete a reporter, query language

```
try(Session session = neo4jConnection.getNeo4jSession()){
    Query query = new Query(
            "MATCH (r:Reporter {reporterId: $reporterId}) " +
            "OPTIONAL MATCH (r) -[w:WRITE]-> (p:Post) " +
            "OPTIONAL MATCH (p) <-[report]- () "+
            "OPTIONAL MATCH (r) <-[follow]- () "+
            "DELETE report "+
            "DELETE follow "+
            "DELETE w " +
            "DELETE p "+
            "DELETE r ",
            parameters("reporterId", reporterId)
    );

    return session
        .writeTransaction(tx -> tx.run(query))
        .consume().counters().nodesDeleted();
}
catch (Exception e) {/*Exception handling */}
```

Figure 33 - Delete a reporter, Java code

15 SYSTEM OPERATIONS

The operations implemented to compose the set of functionalities available to end users are only a part of the overall application. There are some hidden operations that do not directly offer a functionality to the end user, but play a crucial role in the correctness of the whole application. These operations are collectively referred as system operations, to differentiate them from those that provide a service to the user.

15.1 Database consistency

The consistency of the entire database, even if can be relaxed due to the nature of the application, as stated by the initial requirements and how will be discussed in a dedicated chapter, is still a property that must be guaranteed to some extent.

Redundancies

Referring to what has already been presented, the implementation of the mechanism for updating redundancies is now briefly presented. In order to be able to handle the update in the best possible way, four possible types of operations affecting the updating of redundancies have been defined as enumerations:

- ADD_COMMENT,
- REMOVE_COMMENT,
- ADD_REPORT,
- REMOVE_REPORT

Next, a class ('RedundancyTask') was implemented to represent the task to be performed. In particular, it keeps track of the type of task performed, an identifier of the post or reporter concerned and a counter to determine the quantity added or removed. Finally, a class capable of keeping track of each operation performed was defined. In particular, for each operation on the MongoDB database (for adding/removing comments) or Neo4j (for adding/removing reports), an instance of the class representing the tasks is created. Using the thread pool described above, a thread extracted from the pool is assigned the task of writing to one of the two log files used to maintain this information (refer to 10.3 "Updating Redundancies"). Writing to the log is done in a thread-safe manner by managing synchronization in accessing resources and avoiding inconsistencies. Writing to the log file is done using a method provided by the 'RedundancyUpdater' class which makes use of the Singleton-Pattern.

Finally, this last class is the heart of the mechanism. It contains methods for reading and writing log files and for updating redundancies on databases. In particular, the class provides an instance of 'ScheduledExecutorService' which is used to periodically extract the content from the log file in order to carry out the processing and perform the relevant update operations on the databases when necessary. Among the attributes of the class, there are two instances of 'Map' (one for reports and one for comments) in which, once the contents of the log file have been extracted, a counter is maintained for each type of entity (comment or report), trying to summarise different tasks referring to the same post or reporter (respectively) to a single operation. In fact, if there are comments inserted and deleted for the same post, the operation of summarising to a single task allows only one counter to be kept, representing the value to be applied to the redundancy to update it. This limits the number of database accesses making it more efficient. Once the tasks have been summarised in the relevant maps and the update operations have been performed, the logs are emptied of the successful operations, but a trace is kept of the update operations that have produced errors of any kind, thus not allowing the update. Writing these tasks to the logs (in their summarised version with the summary counter of previously logged operations) allows the server to keep track of pending updates to be performed to make redundancy consistent. Such operations will be retried the next time the periodic thread is executed or if the application is terminated.

It is important to emphasise the fact that transactions are exploited in order to update the value of the comment and report number redundancies in the database based on the values contained in the appropriate

maps. In particular, the transactions contain the operations of updating the values of the redundancies in the database, removing the relevant key from the summary map and updating the contents of the log files. Furthermore, update operations on the database are only performed if the operations summary map contains, for a given post or reporter, a non-zero number of comments or reports (respectively). If, on the other hand, the insertion and removal operations were balanced, no update of the redundancies would be necessary. This update management mechanism thus saves many database operations. A transaction is committed if the operations to renew the contents of the log file and delete the contents of the map at the end of a redundancy update are successful, otherwise the transactions are aborted. Different handling is used for database operations. If it is not possible to apply an update to a database redundancy, instead of aborting the entire transaction it is chosen to continue with the update of the next redundancy, writing an error message and keeping the correspondence of this unsuccessful update in the log file for a later attempt at the next periodic event.

Lazy update

Some entities have to be stored on both databases, usually replicating a small fraction of the information stored in the document database even in the graph database. This is necessary because the graph database needs a reference to the entities to model their relationship as described in the dedicated section.

Regarding the reader and post entities is worth to notice that the insertion inside the graph database is not strictly necessary to be carried out in the same instant in which it is created. If no relationship involves such entities, as is at the creation time, the information on the graph database is completely ignored by every operation. Exploiting such observation is possible to perform the insertion of the new entities in the graph database in a lazy fashion, i.e. the insertion is issued when is strictly necessary to complete an operation that requires the creation of a relationship between missing entities.

The lazy insertion allows to limit the number of nodes stored in the graph database, in such a way to reduce the load that it must handle. At the same time, it guarantees to the end user a reduced response time since the system hasn't to perform twice the insertion of the new entity.

From a more practical point of view, the lazy insertion of an entity in Neo4J is possible due to the merge operator offered by the graph database itself. The MERGE clause either matches existing node patterns in the graph and binds them or, if not present, creates new data and binds that. In this way, it acts as a combination of MATCH and CREATE that allows for specific actions depending on whether the specified data was matched or created. In our case, the merge is used to create nodes representing Post and Reader entities. Such nodes have, on Neo4J, only their ID information. This information is, however, already known at the time of the query, thus excluding the need to retrieve this information on MongoDB before creating the corresponding nodes on Neo4J, which would have caused non-negligible delays in the execution of the operation.

CRUD operations and transactions

Finally, there are some simple CRUD operations that cannot exploit any tricks to ensure consistency between databases other than execution within transactions, such as:

- Creation of a new reporter
- Deletion of reporter, reader and post entities

The reasons why they need a transaction are slightly different.

In the former case the entity must be stored in the document database to allows all the operations regarding the retrieval of information about a reporter and his posts. However, if a user tries to follow that reporter the entity should be store even in the graph database, but this time is not possible to exploit a lazy insertion because not all the needed information to be duplicated in the second database are available at the time of

the “follow” query, i.e. the name and the picture attributes. They may be retrieved by the system executing a query on document database, but it would be necessary only if the entity is not yet stored in the graph database. This approach seems neither simple to implement nor efficient, thus the solution adopted is to immediately duplicate the necessary information on both database at the creation time, ensuring that both succeed via a transaction.

In the deletion case the transaction is applied to guarantee that the target entity is removed by both databases to ensure consistency and to remove other entities directly linked to the one to be removed. For example, if a reader is removed then all his comments should be removed as well or if a post is removed then all the comments on it should be removed along with the same entity on graph database. All this chained operation must be guaranteed within a transaction avoid annoying inconsistency in the database that may arise in case of errors in the middle of operations.

15.2 Maximum document size management

Document databases store data within documents, but they have a limit to the maximum size that can be reached for performance reasons. Although this limit can be modified acting on the database configuration, is not a good idea because the solution would only be a kind of problem hiding. As previously described the adopted solution follows a completely different approach, producing multiple physical documents that can be seen as the same entity to bypass the problem.

The operations to be performed to switch from a saturated document to a new document are:

1. Check the current size of the active document and, if it approaches the maximum size considering a certain margin, perform the next steps
2. Retrieve all the information about the target reporter, obviously excluding the list of posts
3. Remove unnecessary field to avoid duplication of information between the documents
4. Create the new empty document for the target reporter containing only the personal information

These operations, as already mentioned, are quite simple, but must be performed within a transaction to ensure that an error in the middle may cause a loss of information.

Another important aspect is deciding when to perform this mechanism. A periodic check may solve the problem, but it may result in an unnecessary waste of resources to perform a check on all active documents. For this reason, the check is performed in response to the expiry of the reporter session, i.e. each time he logs out of the application. This makes sense because it allows to check only those documents that might actually approach the limit size due to the possible new posts published by the reporter during the session just expired.

These are the reasons behind the introduction of the listener in the application server that fires the just described operations each time the reporter session expires.

16 PERFORMANCE TEST

The final system has been tested to assess some of the choices made during the design and implementation phase. To contextualise the tests so that they can be reproduced and have some validity for readers of the documentation in the following will be described the data on which the test has been performed and the configuration of the mongo cluster used before analysing the obtained results.

16.1 Dataset description

The data stored into the database to carry out the tests faithfully assume the expected characteristics by the real data that will be produced by the application once made available to the end users, except of course for the amount of data, even if the proportion between entities has been respected to some extent.

The data comes from three different sources:

1. Collection of Facebook posts and associated comments on news outlets pages
2. Collection of tweets that have been published with the “breaking” hashtag by standard users
3. Generation of fake users via API provided by an online service

The different origins lead to data characterized by variety a common property of big data.

The scripts used to clean, merge and upload the data can be found in the repository of the code, however it’s not so relevant gives a detailed description of steps performed to fill the database. The final data are organized as follows:

- Around 150.000 readers
- Around 270.000 comments
- Around 165.000 posts
- Around 150 reporters

The sources of the data can be found at the end of documentation.

16.2 Application deployment

The components that are part of application have been already discussed in the section dedicated to the system architecture, however the deployment of such components must be taken into account in the overall results.

The cluster available for the test deployment is a virtual one, composed by three virtual machines. Given the fact that the available virtual machines are less than the components of the application some of them must be deployed on the same machine.

The Mongo cluster is composed by three replicas and they must be deployed in different machines, otherwise the communication between replicas may be negligible respect to a real deployment distorting the final results. Thus, the remaining components, application server and Neo4J instance, should be deployed in two different machines to guarantee the fairest load balancing. The deployment of the last two component is pretty simple because the most suitable choice is to dedicate to the primary instance of the Mongo cluster an entire machine, since it is plausible that it should handle a heavier load than the other replicas.

The final deployment can be summarized as follows:

Component	Virtual Machine	Ip address
Apache Tomcat	Profile2022LARGE4	172.16.5.20
Mongo instance #1	Profile2022LARGE5	172.16.5.21
Mongo instance #2	Profile2022LARGE4	172.16.5.20
Mongo instance #3	Profile2022LARGE6	172.16.4.22

Neo4J	Profile2022LARGE6	172.16.5.22

Table 13 - Virtual cluster deployment

16.3 Test results

The test performed regards to different aspects of the document database, for sure the keystone for enhance the performance of the system.

Mongo cluster configuration

The configuration of the cluster has a heavy impact on the performance of the queries that are issued and determines the consistency level guaranteed for the application. In particular, the configuration aspects that are taken into account are the "*write concern*", "*read concern*" and "*read preference*".

Write concern

The "write concern" describes the level of acknowledgment requested from MongoDB for write operations, i.e. the number of replicas that must successfully complete[9] the write operation before the response from primary replica can be sent back to the client. Note that each write operation must be directed to the primary and then it will decide how may acknowledges wait before send back the response.

The write concern is specified by three parameters, even if the last one is not taken into consideration by the test:

- ***"W" (write)***, requests acknowledgment that the write operation has propagated to a specified number of replicas
- ***"J" (journal)***, requests acknowledgment that the write operation has been written to on-disk journal or only into main memory
- ***"Wtimeout"***, specifies a timeout, in milliseconds, for the write concern to receives the acks

Before describing the test and show the obtained result is worth to recall that the acks are a mechanism to avoid possible roll backs of the operation in case of failure of one or more replicas within the cluster. If no acks are expected by the primary node in case of failure of the latter before replication is confirmed by secondaries is possible that the operation will be rolled back even if the client receives a success response. To minimize that issue the write parameter should be set to "majority", i.e. the primary should wait for the acks coming by the majority of replicas belonging to the cluster, primary included.

The tests want to show the impact that the configuration of write and journal parameters have on the insert time of a document into a collection in the database. The configuration considered are summarized in the following table.

Write	Journal	Description
1	1	The primary stores the operation to on disk-journal and returns the response to the client
1	0	As above, but the operation is saved only in main memory before returns the response
Majority	1	The primary stores the operation to on disk-journal, waits for the first ack coming from one of the other two replicas and returns the response to the client. The ack from other replicas is sent just after the operation is saved on journal on disk, the primary waits for only the first ack because the operation must be acknowledged by the majority of the replicas, i.e. 2 out of 3 in the configured cluster.
Majority	0	As above, but the operation is saved only in main memory on each replica

[9] Actually, for performance reasons the operation is not immediately executed by the secondary replicas, but the ack is sent just after the operation is saved on the journal and then it will be completed in asynchronous fashion

3	1	The primary stores the operation to on disk-journal, waits for acks coming from the other two replicas and return the response to the client. The operation is stored in journal on disk even for the other replicas before they can send the ack
3	0	As above, but the operation is saved only in main memory on each replica

For each configuration the time needed for insert a document of 512B has been evaluate, repeating the operation 1000 times. The large number of repetitions is due to the high variance that the repetitions present, mainly due to network instability. To further reduce the impact of the network, which may affect the final, the average insertion time was calculated by filtering out values exceeding the 75th percentile of the obtained series.

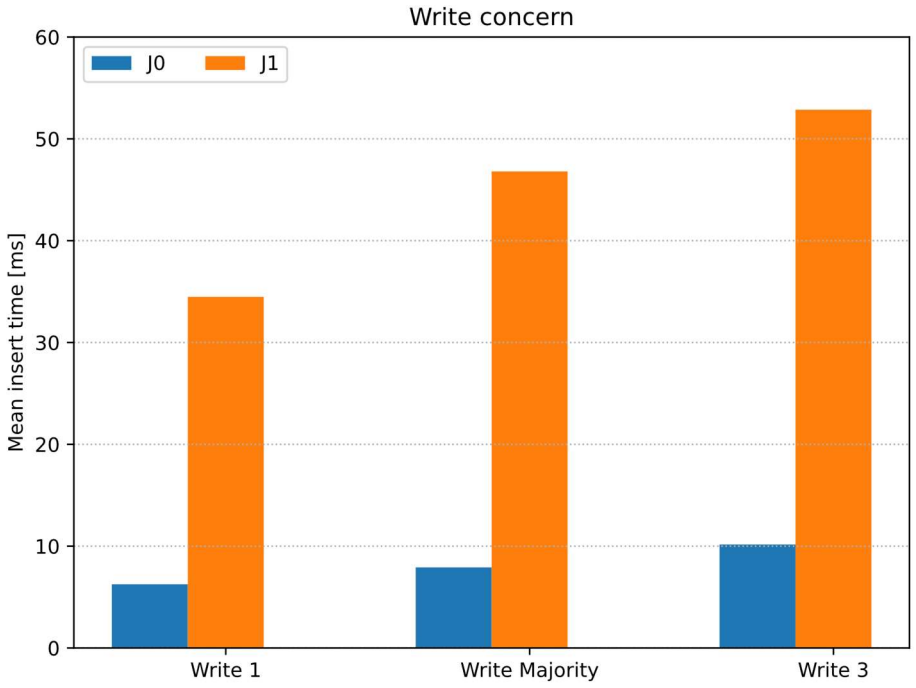

Figure 374 - Write concern test results

As expected, the best performance is achieved without waiting for acks from secondaries and if the operation is saved only in main memory before the primary can return the response. The test allows to extract two interesting results:

- The journal parameter as a huge impact on the insert time experienced by the application
- The write parameter still has an impact on the insert time, but is not so significant as the journal one

Even if the write parameter could be set to “1” to obtain the best performance is worth to pay a penalty and set it at least to “majority” in order to minimize the possibility to experience annoying rollbacks. Regarding the journal parameter if the operation is not stored to on-disk journal a rollback may occurs even if the primary receives all the acks. It seems that the choice should be to set journal to "1" to avoid the same problems that would occur if there were no acks, but before any hasty conclusion, it is good to analyse what scenarios may lead to a rollback setting journal to “0”. A possible rollback may occur in the event of a transient loss of a majority of nodes in a given replica set. This cannot be ruled out a priori, but if replicas are spread across different sites, as they should be to maximize data safety, it sounds more like a catastrophe than a failure. Considering the nature of the application, the high penalty paid setting the journal to “1” and the low probability of rollbacks the optimal choice is to set the write concern to write majority and journal 0.

Read concern

The read concern configuration determines the level of consistency of the read operations, in particular the possibility to read data that are not already durable and may be roll backed in case of failures. In principle may be necessary to solve a trade-off between performance and consistency, but as state in the official documentation the read concern level "majority" in a replica set cluster allows to avoid reading of data that may be rollbacked paying a cost similar to other options that indeed are not able to offer guarantees about durable data. The choice thus is to specify a read concern "majority", i.e. a read operation can return only data already acknowledged by the majority of the replicas in the cluster.

Read preference

The read preference describes how MongoDB clients route read operations to the members of a replica set. That decision has a huge impact on the application because if the operations are not directed to the primary then they might return stale data and not the most updated one. The reason is given by the fact that the replication mechanism is asynchronous and thus a datum already present on the primary may be not yet available on a secondary. The advantage to allow the possibility of read even from secondaries is a load balance between the replicas, which would otherwise provide only fault-tolerant properties.

The eventual consistency is tolerated by the nature of "SocialNews" application, as also state in the initial requirements, so the choice is to permit the reading on the secondaries specifying the read preference "nearest", which lets the client decide to route the request to the replica that experiences the lowest latency.

Indexes and constraints

The implementation of the indices and any constraints previously discussed is presented below, giving, where possible, an example proof of the effectiveness of the indices introduced.

MongoDB indexes

In the design some indexes have been selected as candidates to enhance the reading operations for the data stored in MongoDB. The implementation of such indexes has been carried out taking into account that query planner of MongoDB rarely adopts the index intersection strategy as stated by the official documentation, i.e. if a query specifies multiple conditions on different fields even if on each field is defined an index only one will be used. To obtain the best results in such cases is necessary to define compound indexes formed jointly by all the field that are considered in the query under analysis.

The indexes that have been designed and then tested are reported in the following table.

Collection	Index name	Field(s)	Type(s)
Users	sortByFullNamePagination	fullName, _id	Compound
Users	emailUnique	email	Single, unique
Reporters	searchByFullNamePagination	fullName, reporterId	Compound
Reporters	filterByReporterId	reporterId	Single
Reporters	filterByPostIdReporterId	posts.id, reporterId	Compound, sparse
Reporters	filterByReporterIdPagination	reporterId, posts.timestemp, posts._id	Compound, sparse
Reporters	filterByPostHashtag	posts.hashtags, posts.timestamp, posts.id	Single, sparse
Reporters	emailActiveReporterUnique	email	Single, unique, sparse
Comments	searchByPostSortByTimestampPagination	post.id, timestamp, _id	Compound

Table 14 - Tentative implemented indexes

Note that the unique indexes on "_id" added automatically by MongoDB is not reported in the table.

The sparse typology means that the field may be not present in some documents, while the additional index on reporter's email has been included to guarantee uniqueness of the email. Such index is sparse due to the mechanism of maximum collection size that may define multiple documents for the same reporter, but the email field will be present only in the active document.

The test performed consist in the execution of the queries considered to extract the initial fields used to define indexes, exploiting the "explain" functions provided by MongoDB to profile the query execution.

For some indexes the test can be skipped:

- The "emailUnique" index is mandatory for the uniqueness of the field, however, is also possible to assume that it can guarantee a direct access to the searched document during the login phase, providing benefits for the system
- The "searchByFullNamePagination" index is tested only with the reporters by full name because for the "all reporters" query is possible to assume the results will be the same of the "all readers" query with the appropriate index built in the same manner

Query	Index	Keys examined	Docs examined	Returned docs
allReaders	sortByFullNamePagination	25	25	25 of 149.754
reporterByReporterId	filterByReporterId	1	1	1 of 151
reportersByFullName	searchByFullNamePagination	49	5	5 of 151
postByPostId	filterByPostIdReporterId	1	1	1 of 151*
postByReporterId	filterPostsByReporterIdPagination	1	1	1 of 151*
postsByHashtag	filterByPostHashtag	1	1	1 of 151*
commentsByPostId	searchByPostSortByTimestampPagination	40	40	25 of 271.670

Table 15 - Indexes test results

* *The value refers to the number of reporters, not to the number of posts because the latter are embedded in reporter's documents.*

The results obtained clearly shows that the introduction of indexes reduce the computation needed to retrieve data from the database for queries regarding readers, reporters and comments. The results about queries on posts instead deserve a deeper analysis because the number of documents examined is not an exhaustive metric of comparison due to the fact that the posts are embedded in the reporter's documents, the ones counted by such metric:

- The "postByPostId" query leverages the "filterByReporterId" index to access directly to the document containing the desired post, instead of "filterByPostIdReporterId", but then the filtering of posts can't exploit any index. However, the direct access allows to filter out all other documents without performing a scanning
- The "postByReporterId" is not able to exploit the "filterPostsByReporterIdPagination" index, however the access to the reporter's document is direct using the "filterByReporterId" index. The benefits are thus the same as above
- The "postsByHashtag" is able to use the "searchByPostSortByTimestampPagination" index, however, is possible to define a single index only on the field "hashtags" obtaining the same results with a lower amount of memory needed

Overall considered the final indexes for the MongoDB are:

Collection	Index name	Field(s)	Type(s)
Users	sortByFullNamePagination	fullName, _id	Compound
Users	emailUnique	email	Single, unique
Reporters	searchByFullNamePagination	fullName, reporterId	Compound
Reporters	filterByReporterId	reporterId	Single
Reporters	filterByPostHashtag	posts.hashtags	Single, sparse
Reporters	emailActiveReporterUnique	email	Single, unique, sparse
Comments	searchByPostSortByTimestampPagination	post.id, timestamp, _id	Compound

Table 16 - MongoDB indexes

Neo4J database

With regard to indexes to be introduced into the graph database, several types of indexes can be created on Neo4J:

- Range index
- (Token) Lookup index
- Text index
- Point index
- Full-text index

As stated in the documentation, all types of indexes can be created and dropped using Cypher, and they can be used to index both nodes and relationships. The only default index present in the database is the token lookup index. Range, point, text, and full-text indexes map property values to entities (nodes or relationships). However, token lookup indexes are different as they map labels to nodes or relationship types to relationships, rather than mapping properties to entities. Token lookup indexes are used to look up nodes with a specific label or relationships of a specific type. Therefore, a database can have a maximum of two token lookup indexes - one for nodes and one for relationships. Token lookup indexes are the most important indexes as they significantly speed up the population of other indexes. It's important to note that node indexes and relationship indexes operate in the same way. Additionally, based on the described query types, it is expected that indexes are most commonly used for "MATCH" and "OPTIONAL MATCH" clauses that combine a label/relationship type predicate with a property predicate.

Furthermore, some of these indexes allow specifying the index on a single field or multiple fields. Specifically:

- An index created on a single property for a given label or relationship type is called a single-property index.
- An index created on more than one property for a given label or relationship type is called a composite index.

Here is an overview of the index types to justify the chosen index type and its subsequent implementation:

- **Range index**: In combination with node labels and relationship type predicates, range indexes support most types of predicates (equality check, list membership check, existence check, range search, etc.). They can be used for exact lookups, range scans, full scans, and prefix searches. Range indexes are the most general purpose of property indexes as they support all value types and a wide range of operations.
- **Lookup index**: Indexes presented by default. They solve only node label and relationship type predicates (e.g., MATCH (n:Label), MATCH ()-[r:REL]->()), unlike range indexes.
- **Text index**: In combination with node labels and relationship type predicates, Text indexes only solve predicates operating on strings (e.g., STARTS WITH, ENDS WITH, CONTAINS, IN, etc.). They are a type

of single-property index and only index properties with string values. Text indexes are specifically designed to deal with ENDS WITH or CONTAINS queries efficiently. They are used through Cypher and they support a smaller set of string queries. Even though text indexes do support other text queries, ENDS WITH or CONTAINS queries are the only ones for which this index type provides an advantage over a range index. Text indexes only index single property strings.

- **Point index**: In combination with node labels and relationship type predicates, Point indexes only solve predicates operating on points (e.g., property point value, within bounding box, distance). They are a highly specialized, single-property index and only index properties with Point values, unlike range indexes. Point indexes are designed to speed up spatial queries, specifically the distance and bounding box queries.
- **Full-text index**: Full-text indexes are optimized for indexing and searching text. Unlike text indexes, which only index single-property strings, full-text indexes can index any kind of string data.

Comparing these index types, we can conclude that:

- TEXT indexes are used over RANGE and POINT indexes for CONTAINS and ENDS WITH.
- POINT indexes are used over RANGE and TEXT indexes for distance and within a bounding box.
- RANGE indexes are preferred over TEXT and POINT indexes in all other cases.
- LOOKUP indexes are not defined in this order since they solve a different set of predicates compared to the other indexes.

For our purpose, considering the described queries, it is sufficient to introduce range indexes for the node and relationship IDs in the graph. These indexes will allow us to efficiently identify nodes and relationships based on equality operators. By creating range indexes on the ID properties, we can ensure faster retrieval of nodes and relationships based on their unique identifiers.

From a practical point of view, a best practice is to name the index when it is created. If the index is not named explicitly, it gets an automatically generated name. The name of the index must be unique between indexes and constraints. Index creation is by default non-idempotent and an error will be generated if an attempt is made to create the same index twice. Using the keyword "IF NOT EXISTS" makes the command idempotent and no error will be generated if an attempt is made to create the same index twice. The creation of an index is performed with the command "CREATE ... INDEX ...'. If no index type is specified in the command, an index of type Range will be created. The creation of an index requires the "CREATE INDEX" privilege. The new index is not immediately available but is created in the background. Please refer to the "neo4jDBConfig.md" file in the project folder for commands to create indexes.

Before proceeding with the creation of the indexes, it should be mentioned that, in addition to introducing indexes, during the design phase the need arose to insert constraints to guarantee certain characteristics of the content of our database. The types of constraints that can be introduced in Neo4J are (taken from official documentation):

1. Unique node property constraints: Unique node property constraints, or node property uniqueness constraints, ensure that property values are unique for all nodes with a specific label. For property uniqueness constraints on multiple properties, the combination of the property values is unique.
2. Unique relationship property constraints: (This feature was introduced in Neo4j 5.7, so is not appliable for us) Unique relationship property constraints, or relationship property uniqueness constraints, ensure that property values are unique for all relationships with a specific type. For property uniqueness constraints on multiple properties, the combination of the property values is unique.
3. Node property existence constraints: (Enterprise Edition, so is not appliable for us) Node property existence constraints ensure that a property exists for all nodes with a specific label. Queries that try

to create new nodes of the specified label, but without this property, will fail. The same is true for queries that try to remove the mandatory property.

4. Relationship property existence constraints: (Enterprise Edition, so is not appliable for us) Relationship property existence constraints ensure that a property exists for all relationships with a specific type. All queries that try to create relationships of the specified type, but without this property, will fail. The same is true for queries that try to remove the mandatory property.
5. Node key constraints: (Enterprise Edition, so is not appliable for us) Node key constraints ensure that, for a given label and set of properties:
    - All the properties exist on all the nodes with that label.
    - The combination of the property values is unique.
6. Relationship key constraints: (This feature was introduced in Neo4j 5.7 for Enterprise Edition only so is not appliable for us) Relationship key constraints ensure that, for a given type and set of properties:
    - All the properties exist on all the relationships with that type.
    - The combination of the property values is unique.

Creating a constraint has the following implications on indexes:

- Adding a node key, relationship key, or property uniqueness constraint on a single property also adds an index on that property, and therefore, an index of the same index type, label/relationship type, and property combination cannot be added separately.
- Adding a node key, relationship key, or property uniqueness constraint for a set of properties also adds an index on those properties, and therefore, an index of the same index type, label/relationship type, and properties combination cannot be added separately.
- Cypher® will use these indexes for lookups just like other indexes.
- If a node key, relationship key, or property uniqueness constraint is dropped and the backing index is still required, the index need to be created explicitly.

Additionally, the following is true for constraints:

- A given label or relationship type can have multiple constraints, and uniqueness and property existence constraints can be combined on the same property.
- Adding constraints is an atomic operation that can take a while — all existing data has to be scanned before Neo4j DBMS can turn the constraint 'on'.
- Best practice is to give the constraint a name when it is created. If the constraint is not explicitly named, it will get an auto-generated name.
- The constraint name must be unique among both indexes and constraints.
- Constraint creation is by default not idempotent, and an error will be thrown if you attempt to create the same constraint twice. Using the keyword IF NOT EXISTS makes the command idempotent, and no error will be thrown if you attempt to create the same constraint twice.

Given the version of Neo4J we have, the only constraints we are interested in inserting are those of uniqueness on nodes (type 1). The properties on which we would be interested in placing this constraint are the same as those on which we would be interested in placing an index of type range. Given what we have just stated (referring to the contents of the official documentation), there is no need for nodes to explicitly define indexes. The insertion of constraints implicitly provides for the definition of an index on these fields. The case of the relation of type 'REPORT' is different, for which the uniqueness constraint is not applicable (an upgrade of Neo4J to a version >= 5.7 is suggested in order to make use of it as a possible future development). However, in order to be able to make use of an index to facilitate the search for reports, a special index is specified. Below is a table of the constraints and indexes (default or not) in the database (obtained by “SHOW INDEXES” command).

"id"	"name"	"type"	"entityType"	"labelsOrTypes"	"properties"	"owningConstraint"
1	"index_343aff4e"	"LOOKUP"	"NODE"	null	null	null
2	"index_f7700477"	"LOOKUP"	"RELATIONSHIP"	null	null	null
5	"post_id_uniqueness_constraints"	"RANGE"	"NODE"	["Post"]	["postId"]	"post_id_uniqueness_constraints"
6	"reader_id_uniqueness_constraints"	"RANGE"	"NODE"	["Reader"]	["readerId"]	"reader_id_uniqueness_constraints"
9	"report_id_index"	"RANGE"	"RELATIONSHIP"	["REPORT"]	["reportId"]	null
8	"reporter_id_uniqueness_constraints"	"RANGE"	"NODE"	["Reporter"]	["reporterId"]	"reporter_id_uniqueness_constraints"

Figure 35 - Table of indexes and constraints implemented in Neo4J database

In order to demonstrate the effectiveness of these indices, an example is given below in the case of a simple search query by reporter id. What emerges from this query in terms of performance improvement mirrors what happens with similar queries concerning the other entities.

```
PROFILE MATCH (reporter:Reporter {reporterId:"de505595-527c-42f3-b398-0a443e689fb1"})
RETURN reporter
```

Figure 36 - Query used in order to prove the performance improvement due to indexes

(a)

(b)

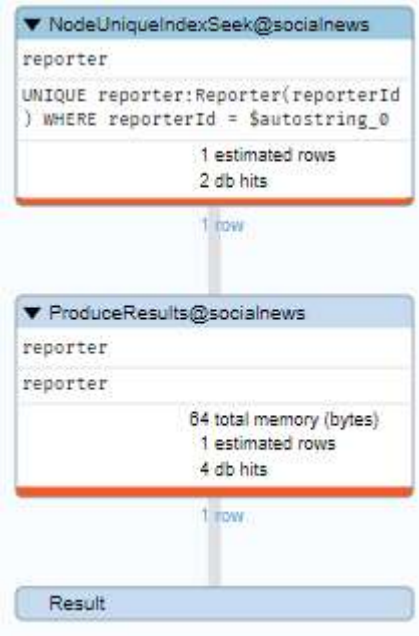

Figure 37 - Performance of the query without (a) and with (b) indexes

17 CONSISTENCY, AVAILABILITY AND PARTITION TOLERANCE

A set of MongoDB replicas with a default configuration, the one with write majority as write concern and read from primary as read preference, may be classified as CP system. The classification comes from the fact that all reads and writes are directed to the single primary replica so the consistency is guaranteed, but in case of disconnection of the primary the cluster need some time to elect a new leader and in the meanwhile the service won't be available.

The configuration that is set for "SocialNews" slightly changes the side in which the system resides because readings are no longer directed exclusively to the primary but can also be directed to the secondary. So, in order to enhance the availability of the system a little of consistency must be sacrificed, allowing to read data that may not reflect the last version, still guaranteeing an eventual consistency for the system. Moreover, the consistency is strengthened by the write concern established, write majority, that minimize the possibility of rollbacks and thus the reading of data that won't be effectively stored permanently in the system. Note that the availability for the write operations can't be guaranteed till the adopted system is based on a single primary that takes care of such operations, because the downtime to elect the new leader in case of failure of the previous one can't be avoided.

Finally, is worth to notice that the cluster is able to respond to read operations regardless the number of active replicas but can serve write operations only if the majority of the replicas are up because is necessary to get that amount of acks before confirming the operation.

18 USER MANUAL

18.1 Login page

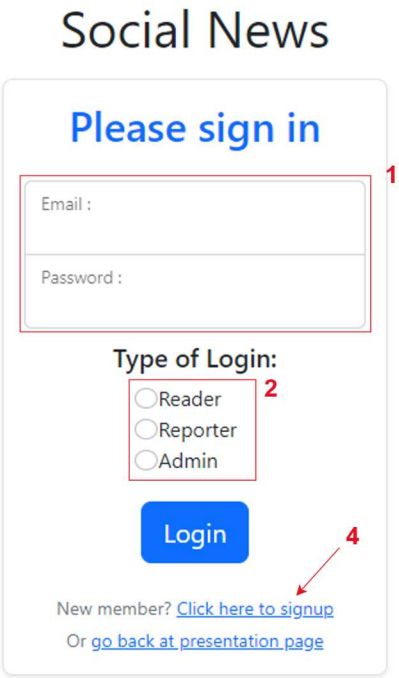

Figure 38 - Login page.

Legend:

1) Email and password fields required from the user for authentication
2) Boxes to tick to select the type of user is logging in
3) Link to the page by which a user of type “Reader” can register on the site

18.2 Signup page

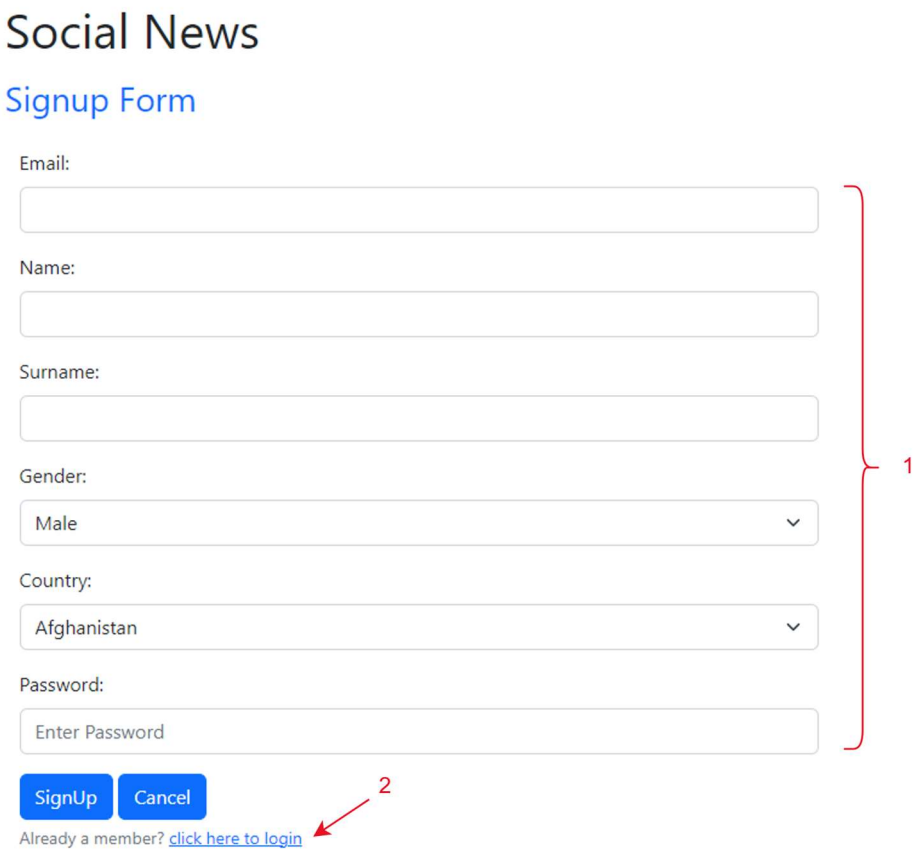

Figure 39 - Signup page.

Legend:

1) Form to be filled in for the reader user to register to the service
2) Link to the login page (see figure 36)

18.3 Reader homepage (after reader login)

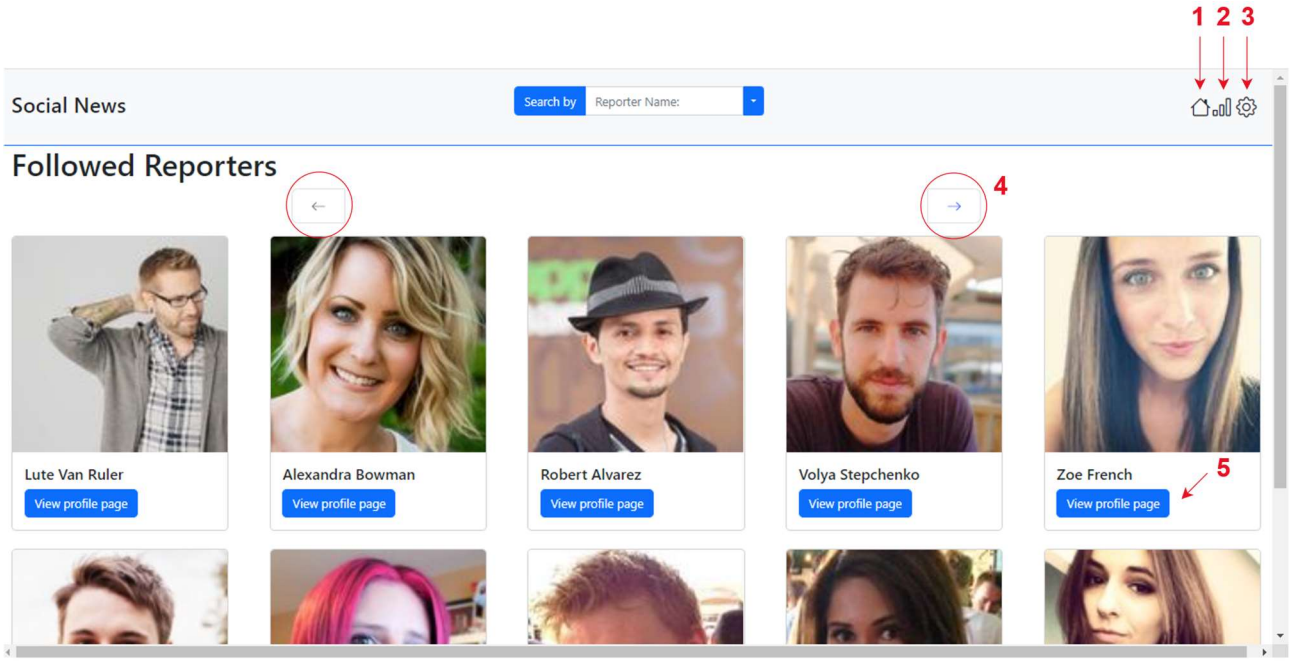

Figure 40 - Reader home with followed reporters.

Legend:

1) Home button to return to homepage when the reader navigate the site
2) Statistic to access suggested reporters page
3) Settings button to logout or view user profile
4) Buttons to navigate pages with followed reporters
5) Buttons on every reporter card to access their profile

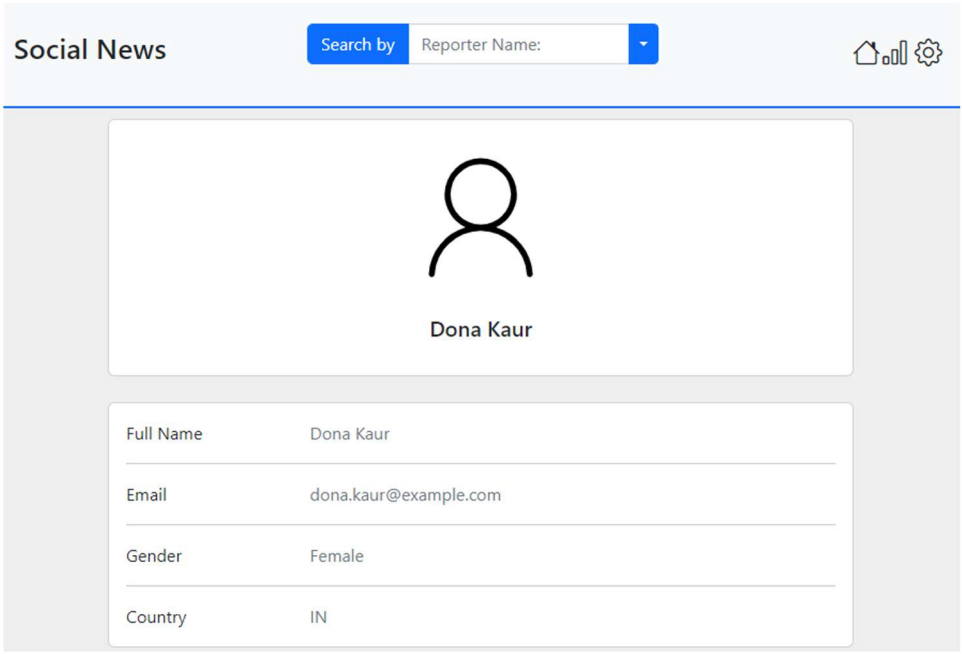

Figure 41 - Profile view of a reader after clicking on Settings -> My Profile.

18.4 Search as reader user by reporter name

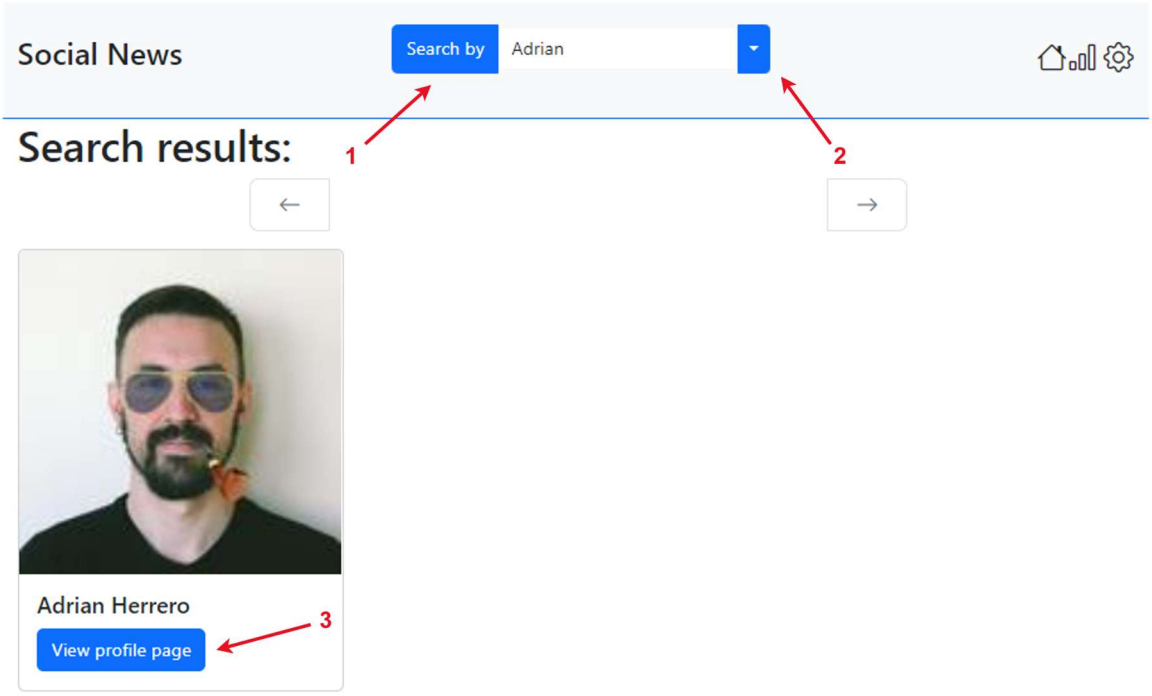

Figure 42 - Reader homepage after a search by reporter name.

Legend:

1) Button to click to run the search
2) Menu to select search type (“Reporter name” or “Hashtag”)
3) Button to click to open profile page of a specific resulted reporter

18.5 Reader view of a post in a reporter page

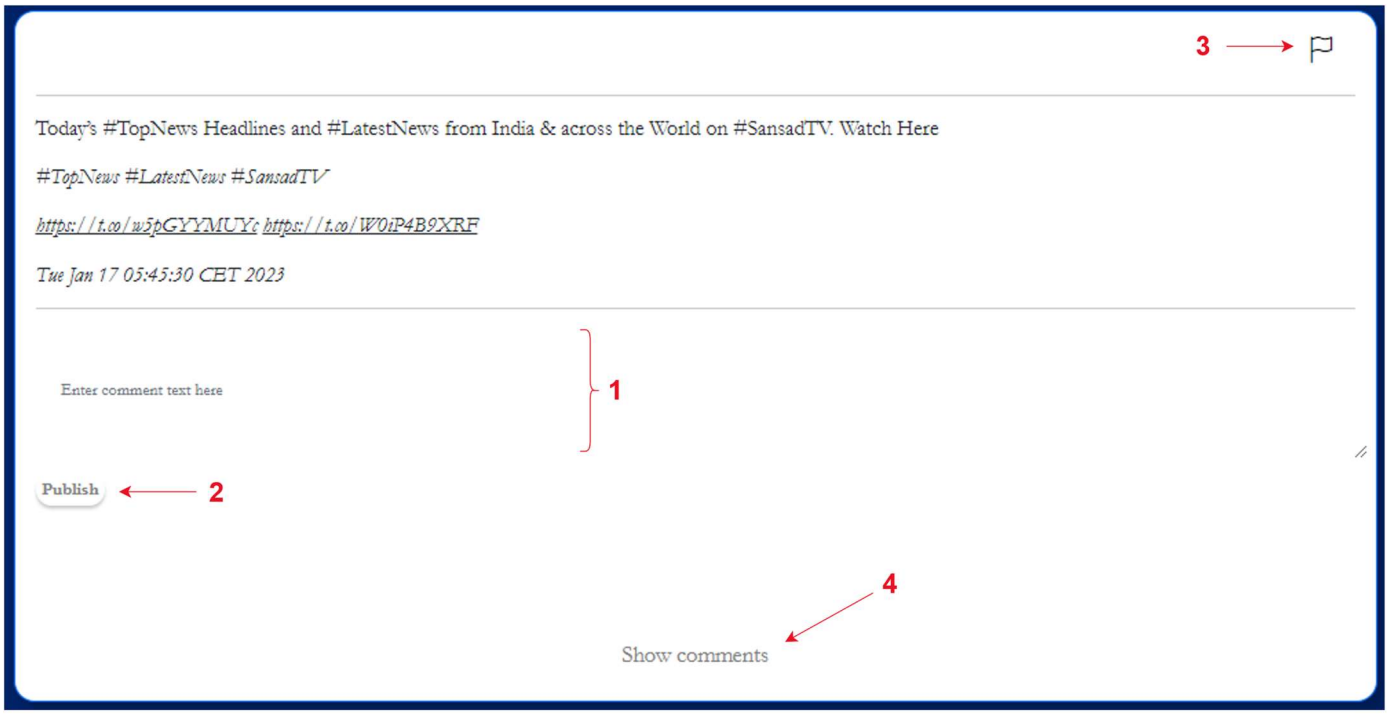

Figure 43 - Post view of a reader.

Legend:

1) Text area where the reader can write a comment
2) Button to click to publish the new comment
3) Button to click to report the post to the admin
4) Button to click to show the first page of comments left for this post

18.6 To report a post as reader

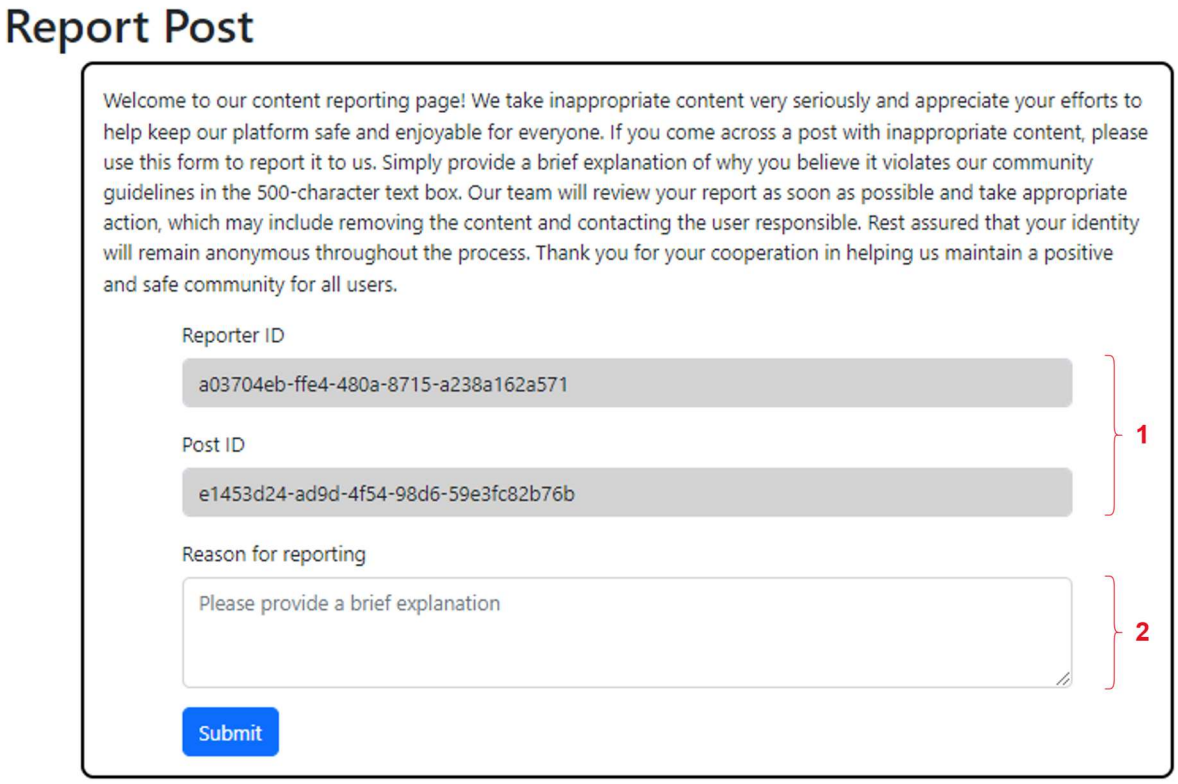

Figure 44 - Report form shown when a reader click on Report button.

Legend:

1) Reporter ID and Post ID fields automatically filled
2) Text area where reader can specify the explanation for the report

18.7 To show more comments

Figure 45 - "Show more" button in a post.

Legend:

1) Button to click to show more comments.

18.8 To remove a comment

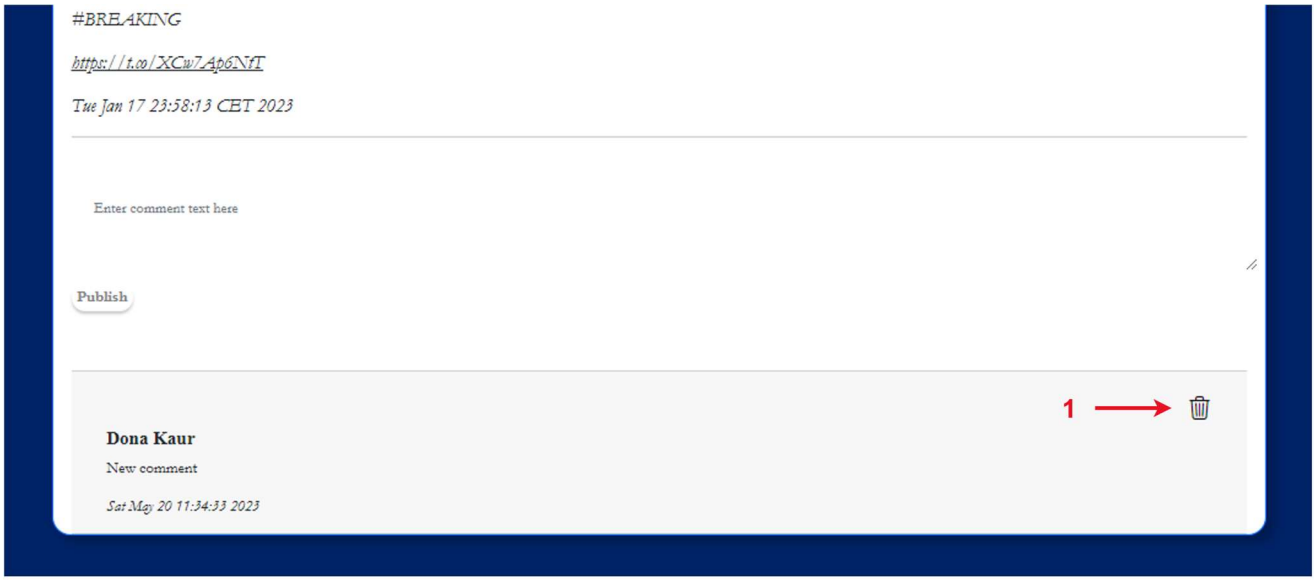

Figure 46 - Button to remove a comment.

Legend:

1) Button to click to remove a comment left by the reader user itself.

18.9 To navigate post pages

Figure 47 - Buttons to navigate post pages.

Legend:

1) Back and forward buttons to click to navigate post pages

18.10 Search as reader user by hashtag

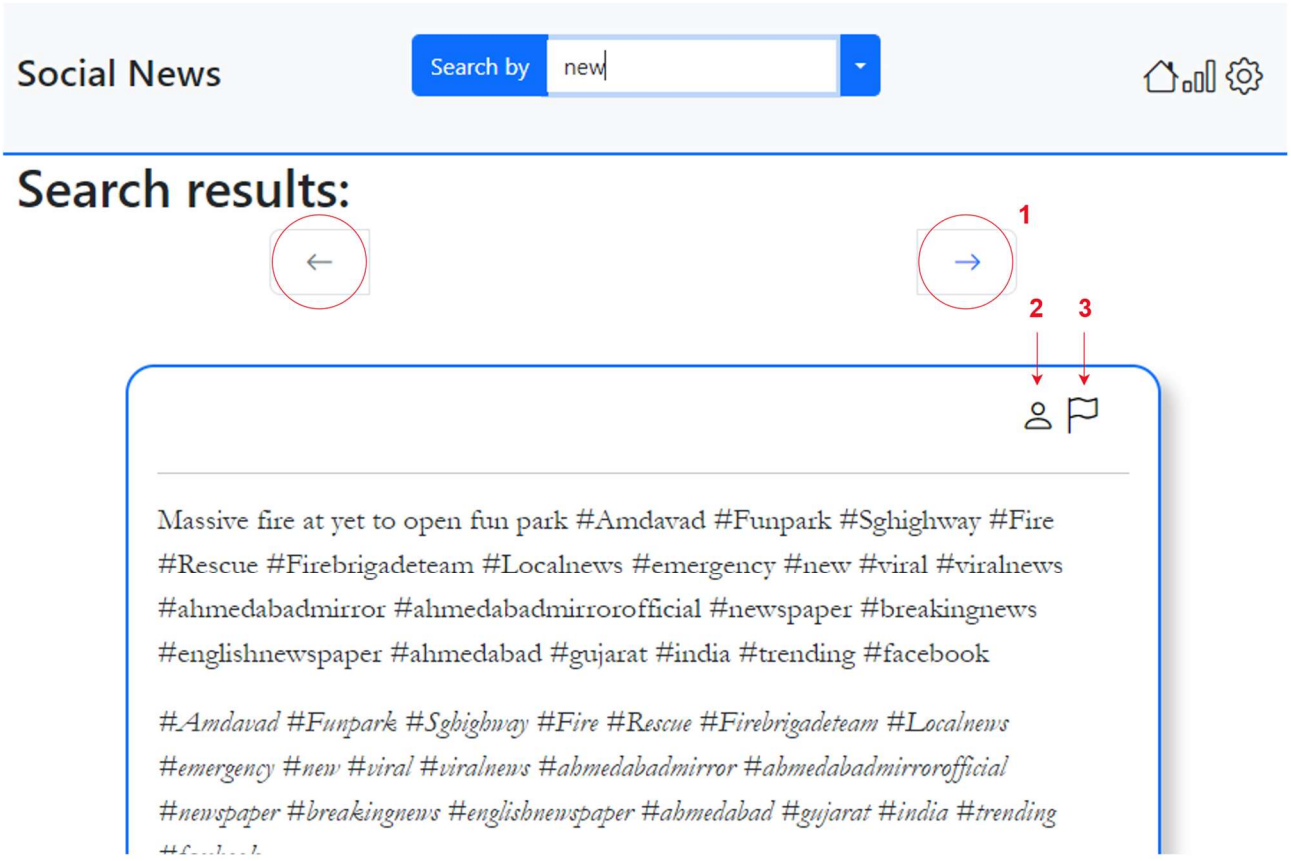

Figure 48 - Result page after a search with hashtag.

Legend:

1) Buttons to click to navigate pages of resulted posts
2) Button to click to open post author profile

3) Button to click to send a report of a specific post

18.11 Reporter homepage (after reporter login)

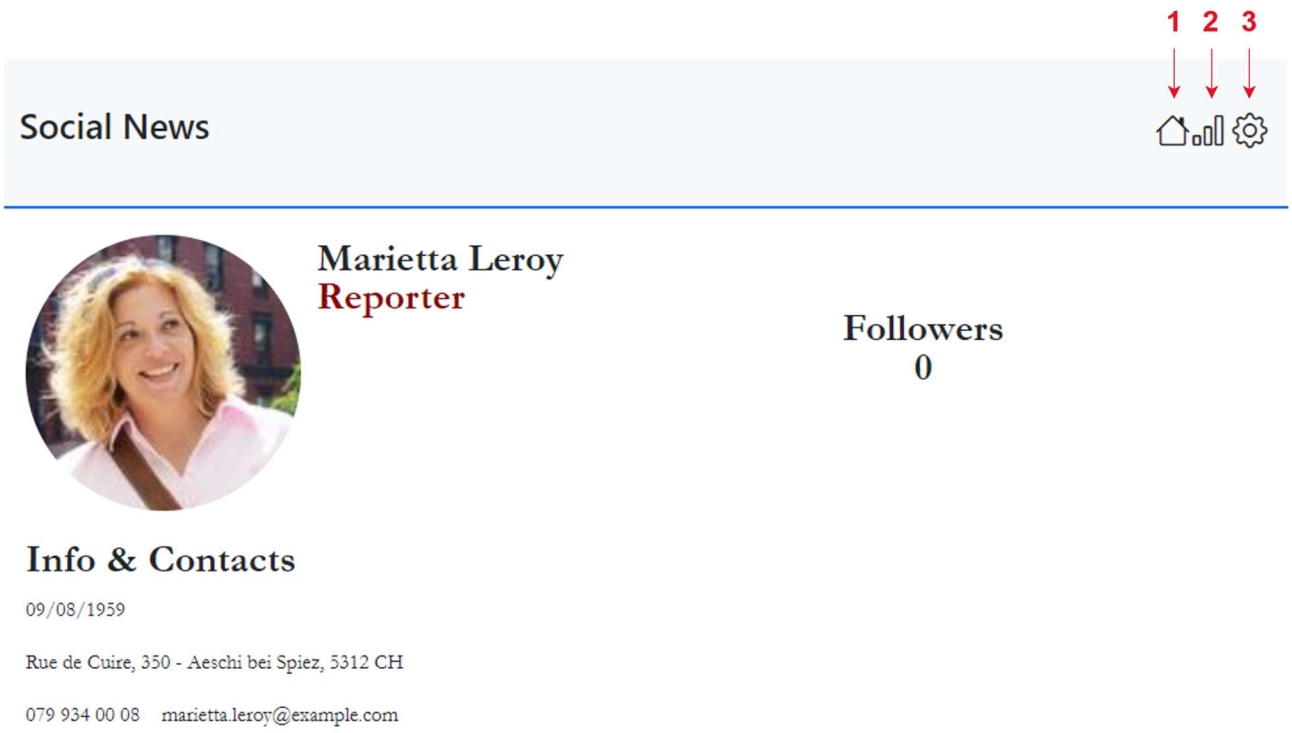

Figure 498 - Header part of a reporter homepage.

Legend:

1) Home button to return to homepage when the user navigate the site
2) Button to access statistics page
3) Settings button to log out

18.12 To write a new post as a reporter

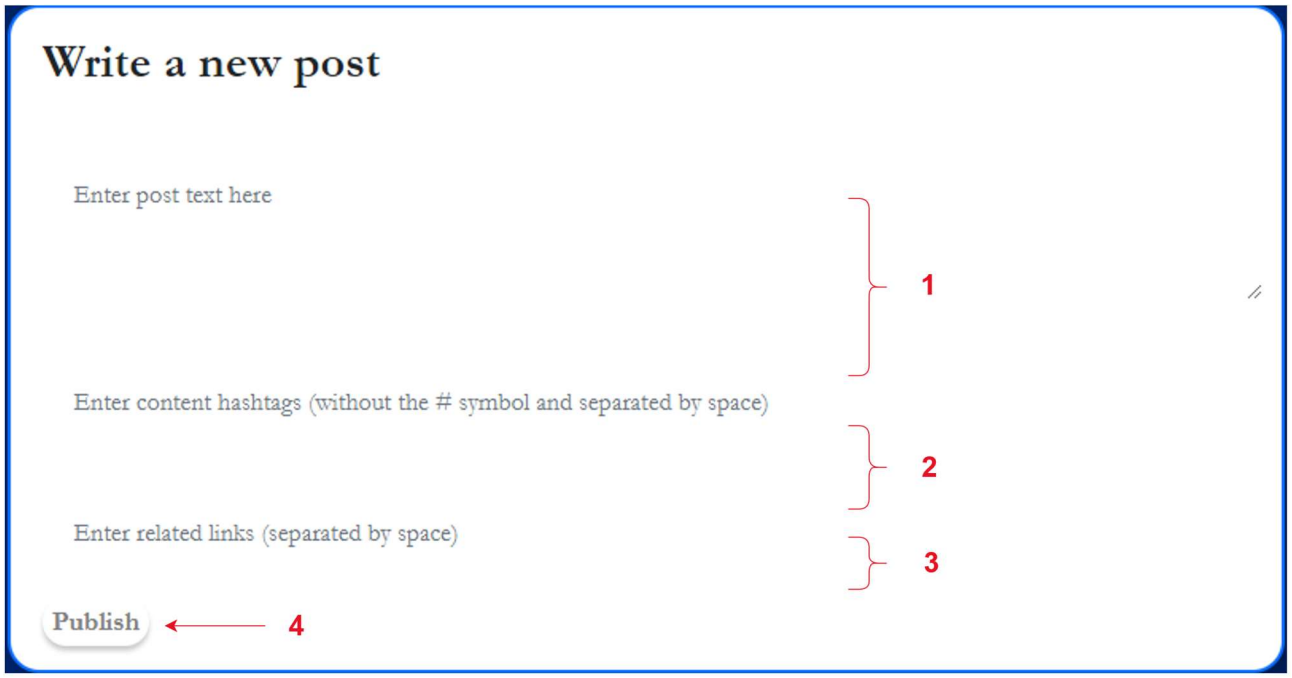

Figure 50 - Form to write a new post as a reporter.

Legend:

1) Area to write the text of the new post
2) Area to write one or more hashtags related to the new post
3) Area to write one or more links related to the new post
4) Button to click to publish the new post

18.13 Post view from the reporter author

Figure 51 - Post view from the author.

Legend:

1) Button to click to show the first page of comments
2) Button to click to remove a specific post

18.14 Reporter statistics

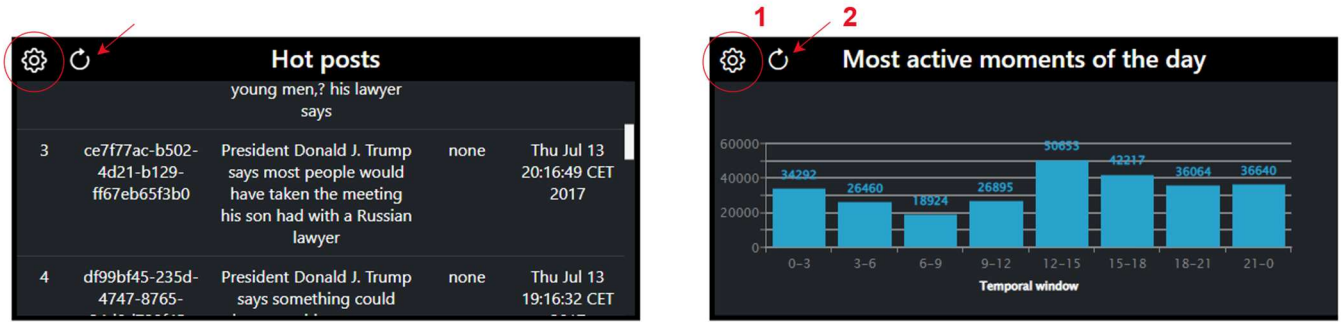

Figure 52 - The two types of reporter statistics.

Legend:

1) Statistics settings button.
2) Button to click to reload the statistic.

18.15 To change statistics settings

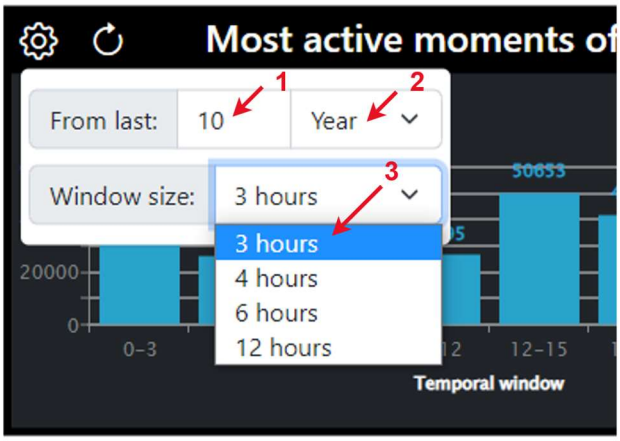

Figure 53 - Focus on the settings of one of the statistics: "Most active moments of the day".

Legend:

1) Field to add or change the number of hours/days/weeks/months/years considered for the statistic
2) Field to select the unit time measure for the statistics between Hour, Day, Week, Month and Year
3) Field to select the window size to calculate the most active moments of the day

18.16 Admin homepage (after admin login)

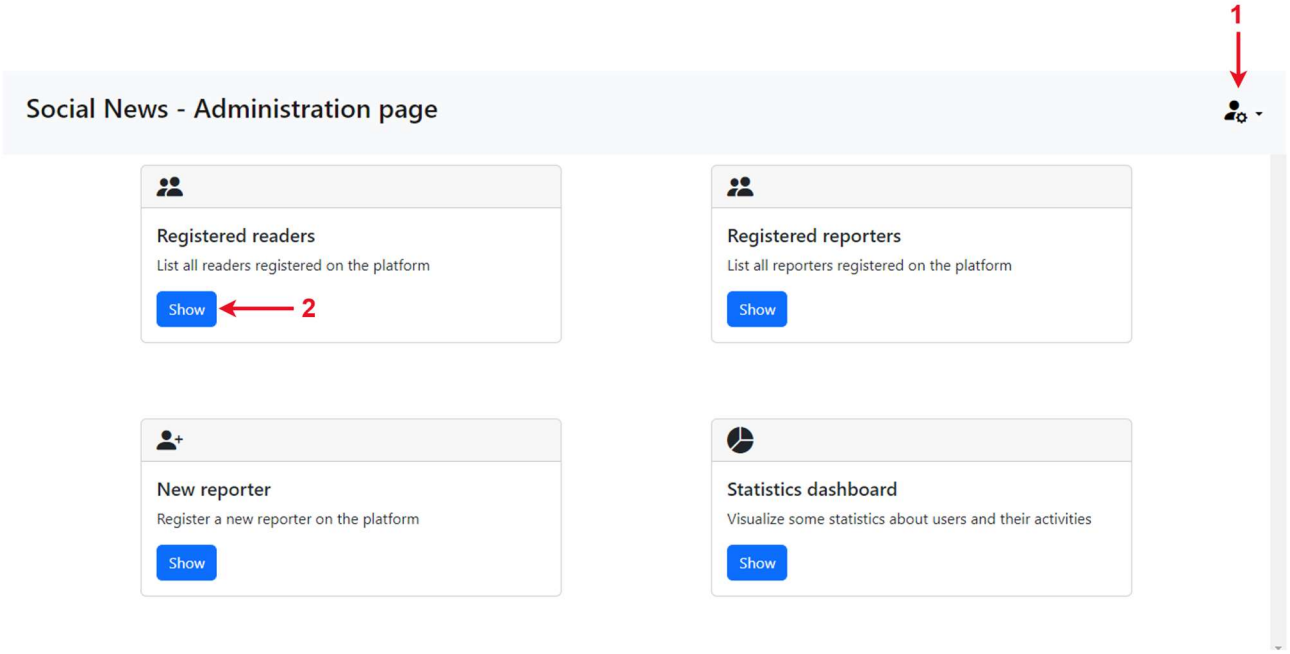

Figure 54 - Admin homepage.

Legend:

1) Button to click to log out
2) Button to access the page related to a specific operation

18.17 Registered readers admin view

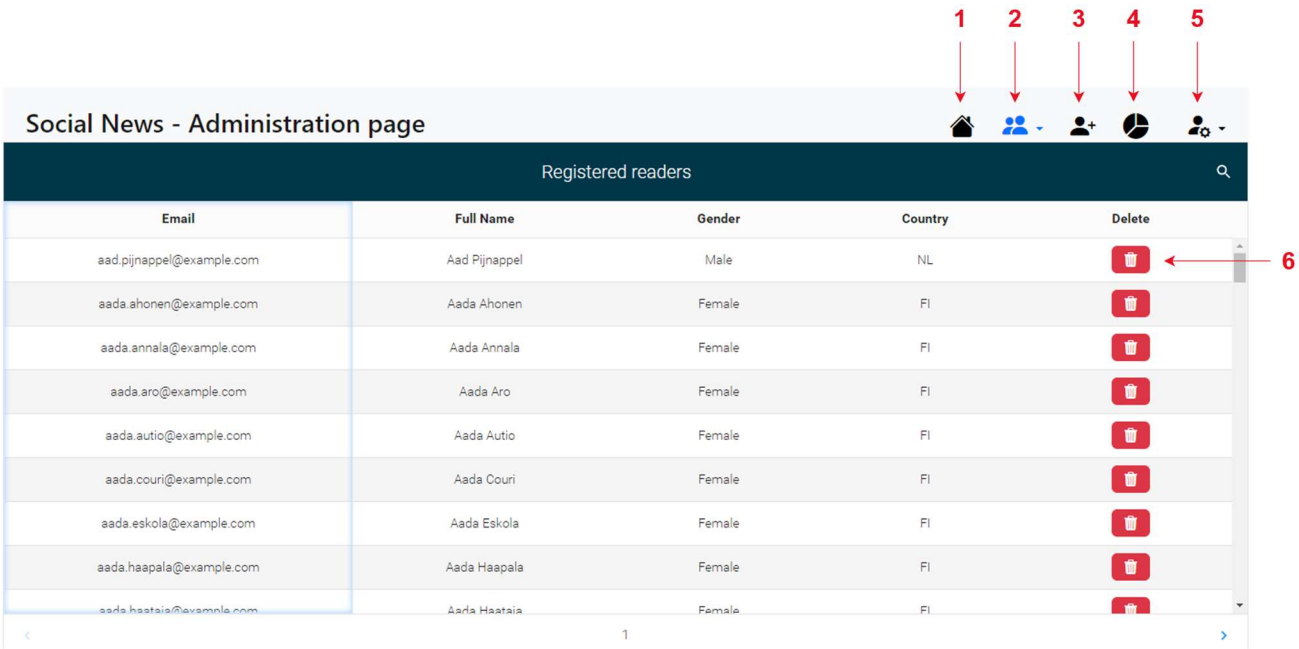

Figure 55 - List of readers admin table.

Legend:

1) Home button
2) Button related to the registered readers view current selected function
3) Button to click to access the “Add new reporter” function

4) Button to access admin statistics page
5) Button to click to log out
6) Button to remove a specific reader user

18.18 Admin view to register a reporter

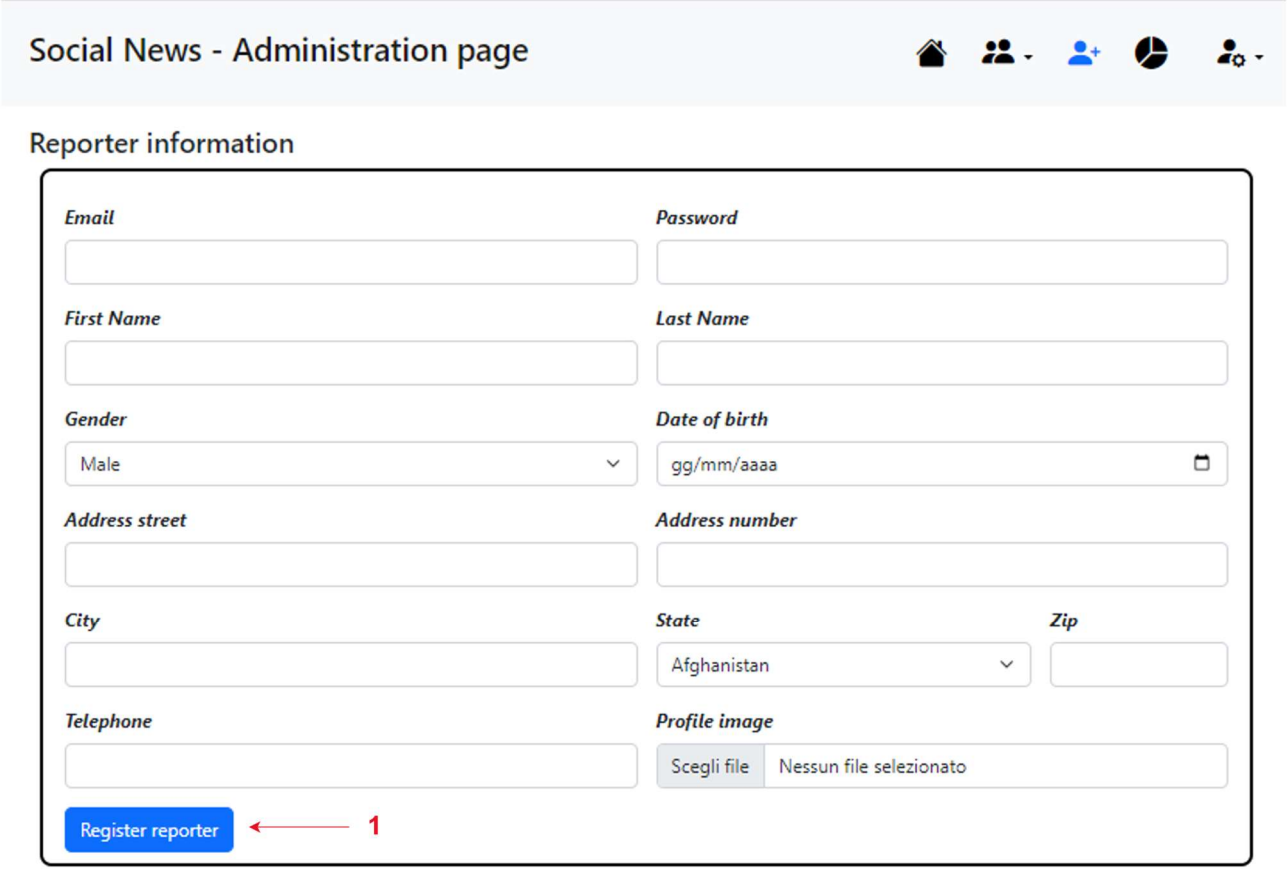

Figure 56 - Form to register a reporter.

Legend:

1) Button to register a new reporter.

18.19 Admin statistics

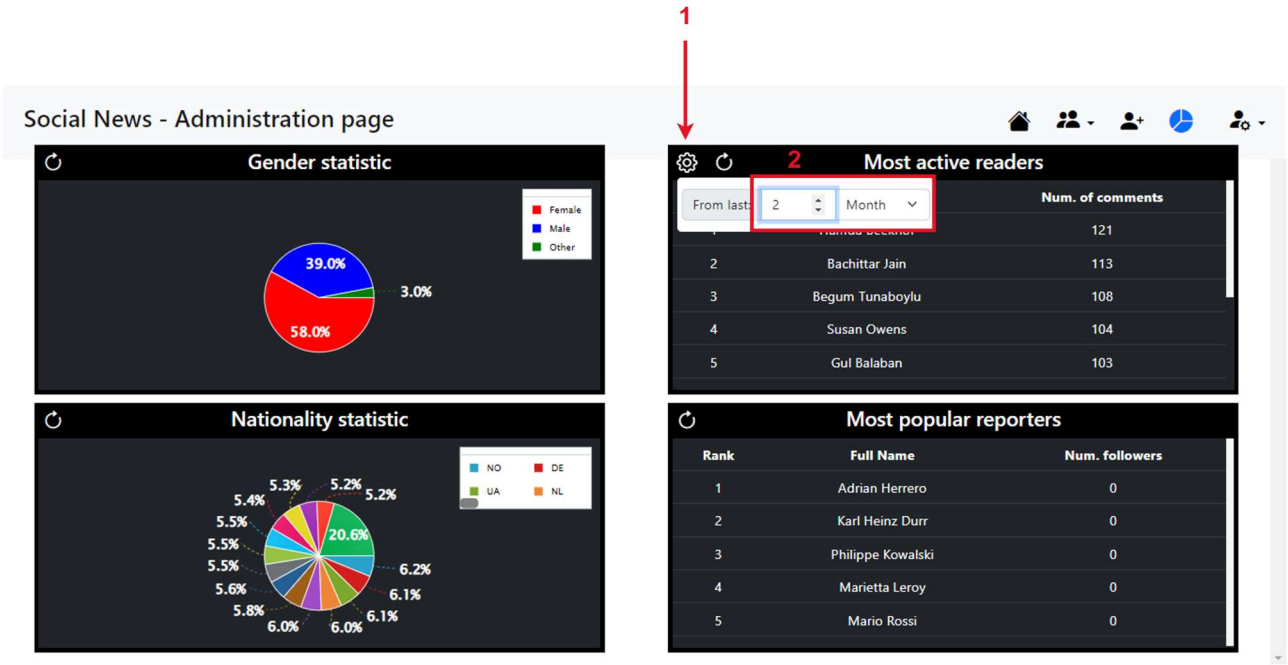

Figure 57 – Statistic page of admins.

Legend:

1) Setting button to change features of a specific statistic (see point 2 for an example of available options)
2) Fields to select number and type of a specific unit of time to calculate the statistic

18.20 To handle reports as admin

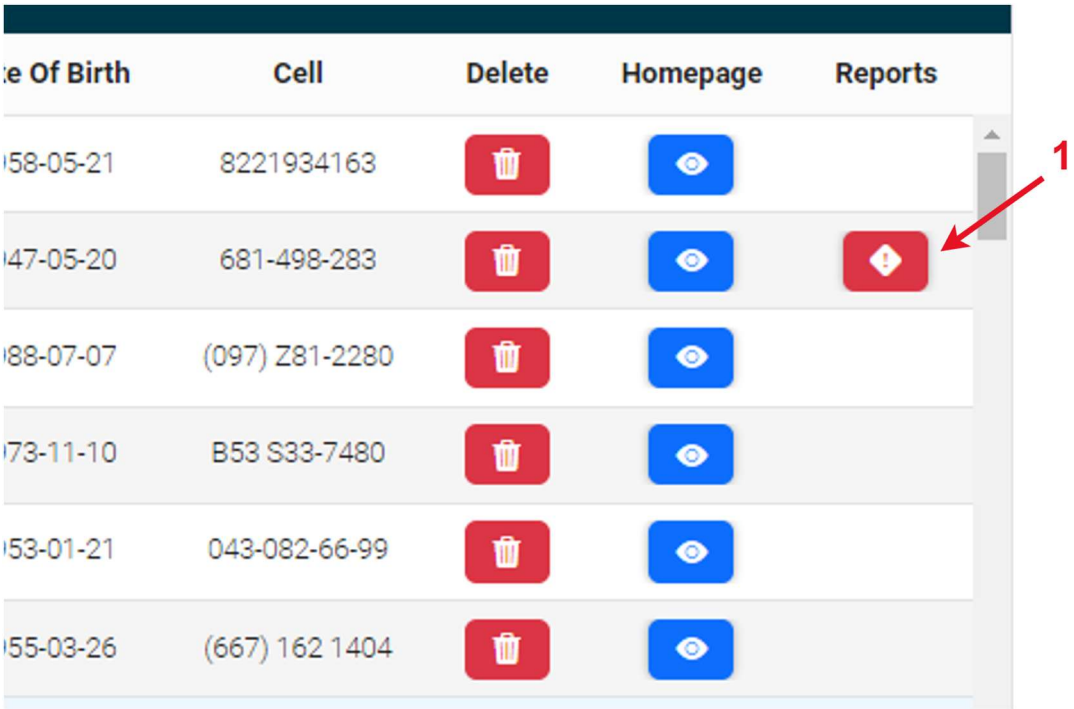

Figure 58 - Focus on registered reporters admin list buttons.

Legend:

1) Buttons to click to view reports text and details

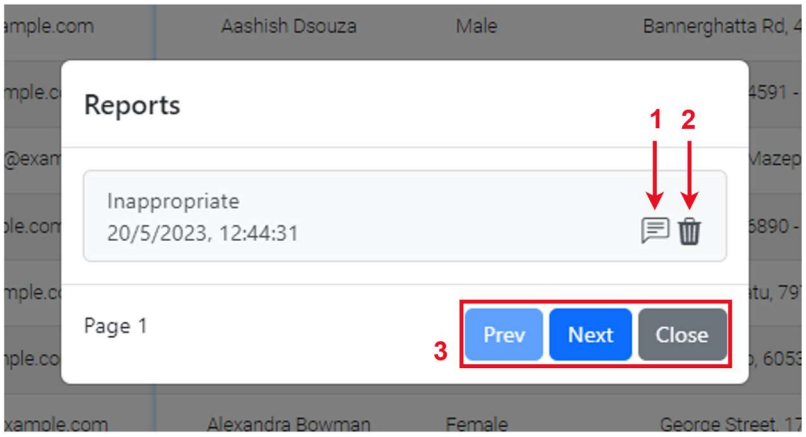

Figure 59: Window to handle reports associated to a reporter.

Legend:

1) Button to view the entire post related to the report
2) Button to remove the report
3) Buttons to navigate between all the reports left for the reporter

18.21 Admin post view

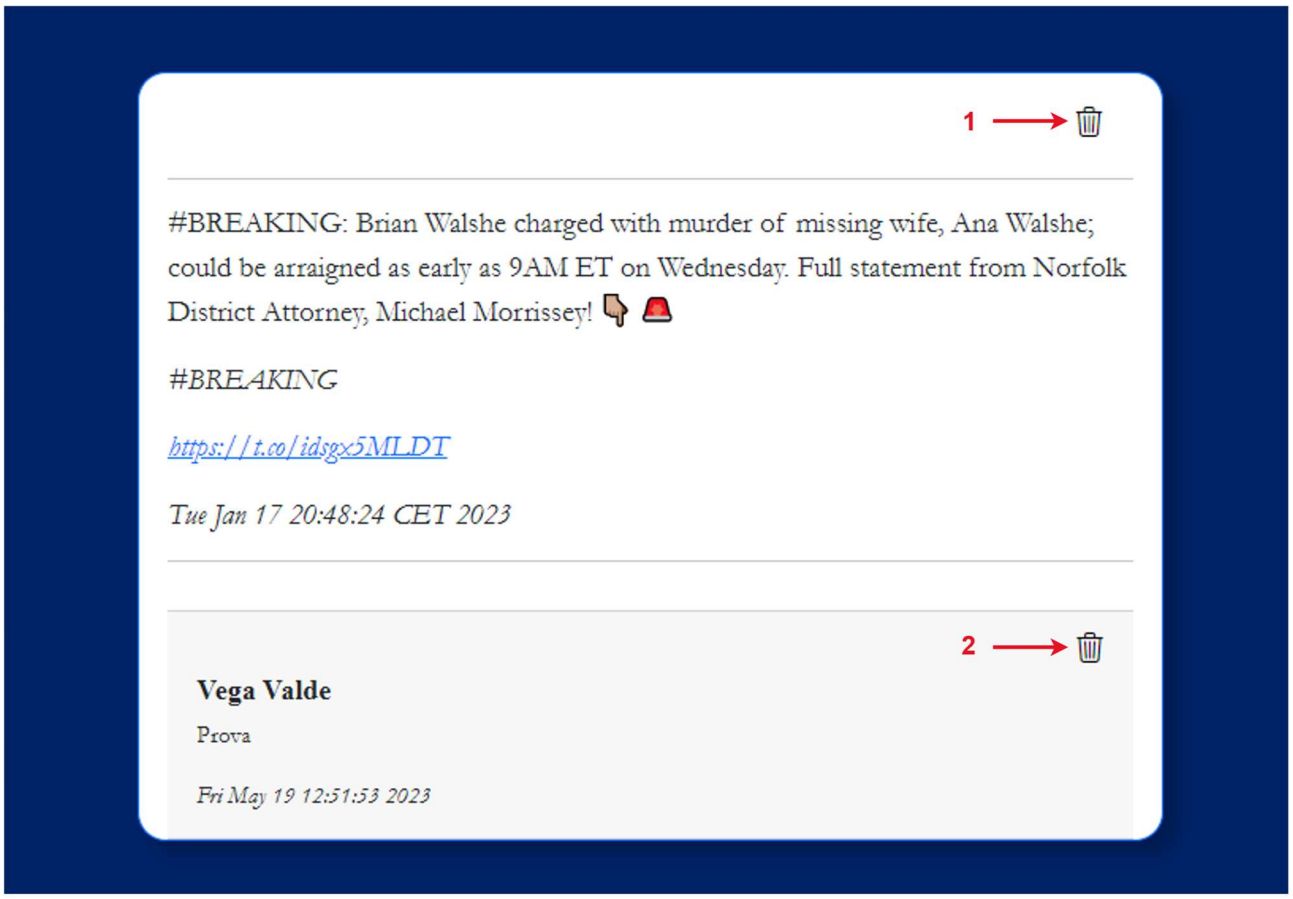

Figure 60 - Admin post view.

Legend:

1) Remove button to delete a specific post

2) Remove button to delete a specific comment

19 FUTURE WORKS

Although many techniques have been analysed to improve system performance, probably to achieve an application capable of supporting millions of users scattered around the world it would be necessary to adopt the sharding technique offered by mongoDB. The sharding technique allows to distribute data across multiple machines, realizing a horizontal scaling approach. The distribution of data is performed at collection level and it is based on a selected shard key, composed of one or more field included in the documents of the target collection. The choice of the shard key determines the performance of the entire sharded cluster because the queries must be routed to the shard(s) that contain the target data. If the routing process is not able to identify a single shard then the query must be broadcast to all shards, resulting in poor performance. The selection of the shard key should be carried out taking into account the queries that the system is expected to execute, in order to minimize the possibility of broadcasting of operations. Moreover, the selected attribute should have some properties to ensure a balanced distribution of data:

- **Cardinality**, determines the maximum number of chunks that can be created, if the field can assume few values then also the possible chunks will be a limited number
- **Frequency**, if the selected field assume very often the same value then the load will be skewed, resulting in a high load on the same shard
- **Monotonic**, if the value assumed by the selected field increases/decreases monotonically then the load will be directed to the same shard, the one dedicated to the max/min value assumed by field

Regarding the operations implemented in the system a good sharding strategy should guarantee:

- All the reporter's documents containing personal information and the embedded array of posts should be stored on the same shard
- All the comments associated to a post should be stored on the same shard
- The data processed by complex aggregation, such as the statistic, can spread across shards to distribute the computation in the cluster

For the collections on MongoDB the desired properties can be achieved using the right field as sharding key:

Collection	Field(s)	Brief explanation
Reporters	reporterId	Ensure that all the documents about the same reporter will be stored on the same shard. The field assume randomly unique values and thus satisfy the properties for a good shard key
Comments	post._id	Ensure that all the documents about the comments on the same post will be store on the same shard. The desired properties for a shard key are again satisfied due to the randomicity of the field
Users	_id	"Users" collection doesn't have particular requirements given by the queries executed by the system, choosing the "_id" field as shard key allow to obtain a balanced distribution of the data

Table 17 - Sharding key

The typology of shard keys may be both hashed and range based because the randomicity of the selected field should guarantee a good balancing. However, as stated by the documentation the configuration of a hashed shard key may lead to more likely broadcast operations, thus the range based should be preferred.

20 BIBLIOGRAPHY

1. **Facebook posts and comments**, https://github.com/jbencina/facebook-news
2. **Breaking news tweets**, https://www.kaggle.com/datasets/bwandowando/breaking-news-twitter-dataset
3. **Random user generation**, https://randomuser.me
4. **JSON style guidelines from Google,** https://google.github.io/styleguide/jsoncstyleguide.xml
5. **MongoDB read concerns**, https://www.mongodb.com/docs/manual/reference/read-concern
6. **MongoDB index intersection,** https://www.mongodb.com/docs/manual/core/index-intersection
7. **Neo4J indexes,**
    - https://neo4j.com/docs/operations-manual/5/performance/index-configuration/
    - https://neo4j.com/docs/cypher-manual/current/query-tuning/
    - https://neo4j.com/docs/cypher-manual/current/indexes-for-search-performance/
8. **Neo4J constraints,** https://neo4j.com/docs/cypher-manual/current/constraints/
9. **MongoDB hashed-sharding**, https://www.mongodb.com/docs/manual/core/hashed-sharding/
10. **GitHub code repository**, https://github.com/MatteBiondi/LSMSD_SocialNews