



UNIVERSITY OF PISA

Master's Degree in Artificial Intelligence and Data Engineering

Large Scale and Multi Structured Databases

LoveMining

Professors:

Prof. Pietro Ducange
Ing. Alessio Schiavo

Students:

Dario Falaschi
Leonardo Lentini
Andrea Doni

ACADEMIC YEAR 2025/2026

1. Introduction.....	4
2. Datasets	5
2.1. User Dataset	5
2.1.1. Data Source	5
2.1.2. Data Manipulation	5
2.2. Interaction and Review Dataset	6
2.2.1. Interaction Generation Logic.....	6
2.2.2. Review Generation Logic.....	7
2.3. Recap General Data Manipulation	7
2.4. Final Datasets Volume	7
3. Design	9
3.1. Actors.....	9
3.2. Mock-ups.....	9
3.2.1. Login and Registration Mock-up.....	9
3.2.2. User Homepage Mock-up.....	10
3.2.3. Admin Homepage Mock-up	11
3.3. Requirement.....	11
3.3.1. Functional requirements	11
3.3.2. Non-Functional requirements	12
3.4. UML Class Diagram	13
3.5. Data Models	13
3.5.1. Document DB	13
3.5.2. Graph DB	15
3.6. Distributed database design	16
3.6.1. Replica set	17
3.6.2. Sharding	17
3.6.3. Inter Database Consistency	18
4. Implementation	20
4.1. Spring	20
4.2. Models	20
4.2.1. MongoDB Models	20
4.2.2. Neo4j Models	21
4.3. Repository	22
4.4. Service.....	23
4.5. Controller	23
4.5.1. Authentication Controller	24
4.5.2. User Controller	24
4.5.3. Admin Controller	24
4.6. RESTful API Endpoints	24
5. Most relevant queries	26
5.1. MongoDB Queries.....	26
5.1.1. Unhappy Cities	26
5.1.2. Glow Up.....	27
5.1.3. Status by Age Group.....	29
5.1.4. Orientation by Age Group	30

5.2.	Neo4j Queries	31
5.2.1.	Recommendation mechanism.....	31
5.2.2.	Love Points	32
6.	<i>Indexes</i>	34
6.1.	MongoDB Indexes	34
6.2.	Neo4j Indexes	35
7.	<i>AI Tools Usage</i>	36
7.1.	Purpose	36
7.2.	How.....	36
7.3.	Critical evaluation	36

Repository GitHub:
<https://github.com/Andryd22/LoveMining>

1. Introduction

LoveMining is a platform designed to connect people based on compatibility. The main idea behind the project is to combine geographical proximity with shared interests to help users find the best possible match. The application focuses on creating a system where people can easily interact based on where they live and what they like to do.

From the user's perspective, the application allows for detailed profiling. Users can edit their personal and demographics information to present themselves to the community. The system considers the age and the sexuality of the user, then it uses a geo-spatial search engine that filters users to show only those living in the same city or in a nearby area. In addition to location, the platform uses interest-based matching: this means the system suggests users who share common passions by analysing the possible connections they may have. To ensure the quality of interactions, the platform also includes a review system, where users can leave a rating and a comment after a match.

The project also provides specific tools for administrators to oversee the application. Through an analytics dashboard, admins can view demographic statistics to analyse the distribution of users and understand trends among the communities. Finally, administrators are responsible for managing the platform, with the possibility to search for specific users and delete their profiles, if necessary.

2. Datasets

To build a realistic and scalable environment for LoveMining, the project relies on two distinct types of data.

The first part consists of real user profiles sourced from an external repository, ensuring that the demographic and descriptive information is authentic.

The second part is a synthetically generated layer of interactions (such as likes, matches, and reviews) designed to simulate the activity of a social network.

2.1. User Dataset

The core of our application is populated using real-world data to ensure the profiles look and feel authentic.

2.1.1. Data Source

For the user profiles, we used the OkCupid Profiles dataset available on Kaggle.

This dataset consists of approximately 60000 records stored in CSV format, providing a significant volume of data to test the system's capacity. It contains a comprehensive collection of anonymized real user information, ranging from standard demographic details (such as age, sex, status) to specific physical attributes (such as height, body type) and information about their life (such as religion, diet, income).

The choice of this specific dataset was driven by its variety and the presence of unstructured text fields (essays), which are essential for testing the document-oriented features of our database and serve as the raw material for the interest extraction process described in the next section.

2.1.2. Data Manipulation

The raw data from Kaggle required several processing steps to be suitable for our system. We performed a data cleaning and transformation process to refine the information:

- **Column Cleaning:** we removed columns that were considered unnecessary or inappropriate for the scope of our project (for example, field related to "drugs" was dropped). This helped us focus only on the most useful attributes.
- **Credential Enrichment:** dataset was anonymized, it missed the necessary information for user authentication. To resolve this, we synthetically generated an email address and a password for every profile. These attributes are essential for simulating the login process and uniquely identifying users within the database.
- **Location Splitting:** the original dataset contained a single string for location (e.g., "San Francisco, California"). We processed this field to split it into two separate attributes: City and State. This separation is crucial for enabling the geo-spatial search features in our application.

- **Interest Extraction (NLP):** This was the most critical step. The original dataset did not have a clear list of "interests" but contained a long textual essay where users described themselves. We analysed these essays to extract key terms and hobbies. After extracting the keywords, we performed a general cleaning to remove stop-words and irrelevant terms. Finally, we verified the results to assign a specific list of "interests" (e.g., "sport", "wine", "music") to each user, transforming unstructured text into structured tags usable by our matching algorithm.

2.2. Interaction and Review Dataset

While the user profiles are real, the connections between them cannot be static. To simulate a living social graph, this second dataset focuses on the interactions between users.

Since we cannot retrieve real-time private interactions from public datasets, this layer is synthetically generated.

This dataset includes the topology of the network (who likes or dislikes whom, matches) and the reviews made by users, created according to specific logic to stress-test the system's performance.

2.2.1. Interaction Generation Logic

To avoid creating chaotic or unrealistic data, which would behave like random noise, we established specific rules to mimic authentic human behaviour on a dating platform.

First, we defined the volume of interactions.

We applied a Normal (Gaussian) Distribution to define two key metrics: an "Activity Level" (mean=20, deviation=10), which dictates the volume of swipes, and a "Popularity Score" (mean=0.5, deviation=0.2), which simulates the user's attractiveness.

This statistical approach ensures a realistic variance where most users perform an average number of interactions, while a minority of "power users" represents the tail of the curve.

Second, and most importantly, we applied strict compatibility rules to determine who interacts with whom. A generic random function would fail to reflect reality, so we introduced three key constraints:

1. **Sexual Orientation & Gender:** the generator strictly respects user preferences (e.g., a straight male will only "Like" female profiles). This acts as a hard filter to ensure logical consistency.
2. **Geo-Spatial Proximity:** given the uneven distribution of our dataset (heavily skewed towards California/San Francisco), random global matching would be inexact. We implemented a dynamic pooling strategy: users in populated urban centres (≥ 100 profiles) are matched locally, while users in smaller towns interact with a State-wide pool. To further simulate local preference, the algorithm applies a 0.1 probability boost if the candidate resides in the exact same city.
3. **Interest Affinity:** to add depth to the graph, the probability of a "Like" is dynamically increased by 0.05 (5%) for every specific interest shared between the two users, simulating psychological compatibility.

Matches, consequently, are not random events but the result of this structured process, generated only when the system detects a reciprocal interest (bidirectional Like).

The implementation of the logic described above was supported by AI tools, specifically for the creation and review of the Python generation script.

2.2.2. Review Generation Logic

The generation of reviews is strictly linked to the matches created in the previous step. To ensure data consistency, a review is created only if a confirmed match exists between two users in the matches.csv file.

To guarantee that the data makes sense (semantic coherence), we linked the text comments to the numerical ratings using pre-defined dictionaries (Excellent, Good, Average, Poor, and Horrible). For example, if the system assigns a 5-star rating, it automatically selects a comment from the "Excellent" generated list. This prevents errors, such as having a high score accompanied by a negative comment.

The process follows a simple logic using a Python script.

Firstly, the algorithm reads the entire matches file.

For each match, it decides whether a review should be written or not. If the outcome is positive, the system determines who writes the review: it can be just one user or both users reciprocally.

Finally, the script assigns a rating and selects a consistent text comment from the corresponding pool.

The final number of reviews will be approximately equal to the number of matches.

This result comes from our generation logic: for every match, there is a 30% chance that both users write a review, balanced by a 30% chance that neither user writes one. The remaining 40% represents cases where only one user leaves feedback. This distribution ensures realistic user behaviour.

The implementation of the logic described above was supported by AI tools, specifically for the creation and review of the Python generation script.

2.3. Recap General Data Manipulation

The final data manipulation process began with the User dataset, where after inserting an administrator user, we used a unique Object_id for every profile to serve as the primary key across the system.

With these IDs, we proceeded to generate the primary interactions (likes, dislikes, and matches) based on the generation logic explained in the previous sections.

Subsequently, we moved to the creation of the reviews based on the confirmed matches. This step involved linking every review to the specific reviewer and implementing the "reviews_made" array structure directly within the users' file.

Finally, to optimize the dataset for importation, we cleaned the files by removing redundant fields, such as the specific reviewer ID, from every element in the reviews collection.

Moreover, for the User dataset, having a lot of sparse data, we removed every null field and realized a json file much cleaner.

2.4. Final Datasets Volume

After completing the generation process, the final dataset consists of five primary CSV files.

The core dataset.csv contains approximately 60,000 profiles, occupying roughly [~55MB].

The interaction layer is split into likes.csv (~400,000 records, [20MB]) and dislikes.csv (~730,000 records, [35MB]), providing over 1.1 million graph edges.

The reciprocity logic resulted in matches.csv, a file containing about 3,500 mutual connections and the reviews.csv file containing about the same number of reviews.

3. Design

3.1. Actors

In the system, we identified two main actors:

- The Registered User
- The Administrator

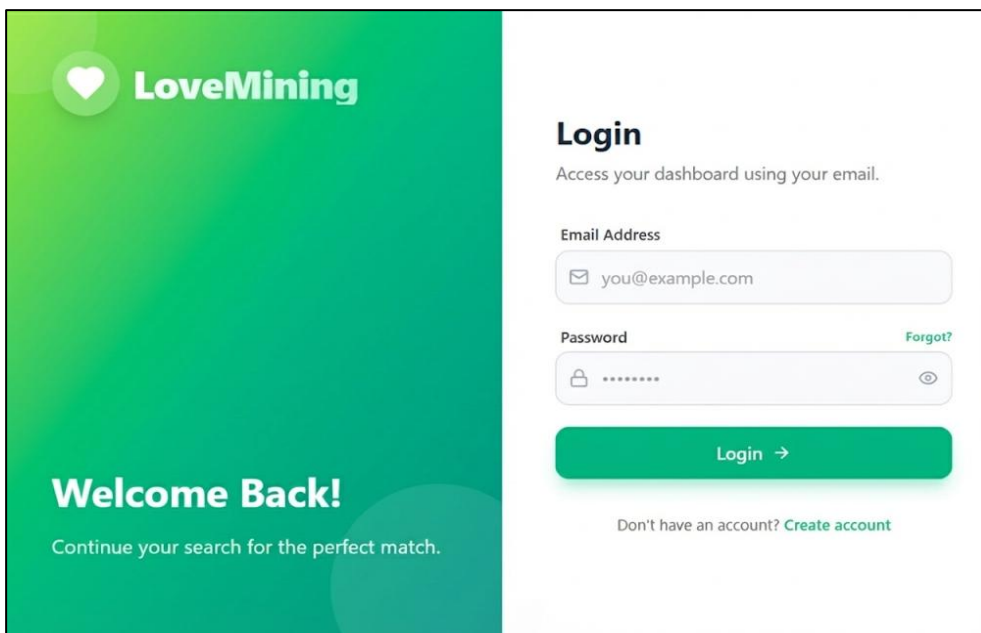
The Registered User represents the end-user of the application. Once the registration is complete, these actors have full access to the platform's dating features. They can manage their personal profile, search for potential partners using filters, and generate interactions (Likes/Dislikes). Moreover, upon a successful match, they can write reviews to rate their experience.

The Administrator has the role of supervising and maintaining the platform. This actor does not participate in the matching process but holds high-level privileges to manage the data. His responsibilities include monitoring user activity, banning fake or malicious profiles, and moderating the content of reviews. Additionally, the Administrator has the authority to run specific analytics to extract meaningful insights from the platform's usage data.

3.2. Mock-ups

3.2.1. Login and Registration Mock-up

First page represents the entry point of the application; it allows existing users to authenticate by entering their credentials. Additionally, it provides a direct link for new visitors to navigate to the registration page if they do not possess an account yet.



The mock-up shows a login and registration interface for 'LoveMining'. The left side features a green gradient background with the 'LoveMining' logo (a heart icon) and the text 'Welcome Back! Continue your search for the perfect match.' The right side is white and contains the 'Login' section. It includes a sub-header 'Access your dashboard using your email.', an 'Email Address' input field with the placeholder 'you@example.com', a 'Password' input field with a 'Forgot?' link, and a green 'Login →' button. At the bottom, there is a link 'Don't have an account? Create account'.

This page allows new users to create a profile. The interface presents a series of input fields to collect personal data; some fields are mandatory to ensure data quality, while others are optional. At the bottom of the form, there is a dedicated text area where users can write a short personal essay to describe themselves.

Registration

ACCOUNT DETAILS

EMAIL: you@example.com

PASSWORD: *****

CONFIRM PASSWORD: *****

THE BASICS

AGE: 25

SEX: Select...

ORIENTATION: Select...

STATUS: Select...

HEIGHT (INCHES): 68

CITY: San Francisco

STATE: California

LIFESTYLE

BODY TYPE: Select...

DIET: Select...

SMOKES: Select...

DRINKS: Select...

PETS: Select...

BACKGROUND

EDUCATION: Select...

JOB: Software Engineer

INCOME: \$ Select...

ETHNICITY: Select...

RELIGION: Select...

OFFSPRING: Select...

SPEAKS: Languages...

PERSONAL ESSAY

TELL US ABOUT YOUR INTERESTS & YOURSELF

I love hiking on weekends and trying new Italian restaurants. I'm looking for someone who shares my passion for...

Create Account & Profile →

3.2.2. User Homepage Mock-up

This is the main hub for the registered user.

On the left sidebar, the user can access their profile settings to modify personal information. Below this, there is a list of current matches, with a feature that allows writing a review for users they haven't rated yet. On the right side, a menu allows the user to filter the search by "City Scope" and set a specific "Age Range".

In the centre, the application displays a candidate profile card with key information. From here, the user can view full details and perform the main interaction by clicking the Like or Dislike buttons.

LoveMining

Alex Miner

YOUR GEMS (MATCHES)

- Jessica Write Review
- Emily Write Review
- Ashley Write Review
- Anna 4 Rated
- Julia 5 Rated

Mining in your area...

Sarah 26
Palo Alto, California

View All Information

HEIGHT: 170 cm

EDUCATION: Master's Degree

JOB: Interior Designer

STATUS: Single

DRINKS: Socially

SMOKES: No

Mining Filters RESET

CITY SCOPE

☒ Same City

☐ Nearby Cities

"Nearby" includes surrounding cities within standard commuting distance.

AGE RANGE 24 - 32

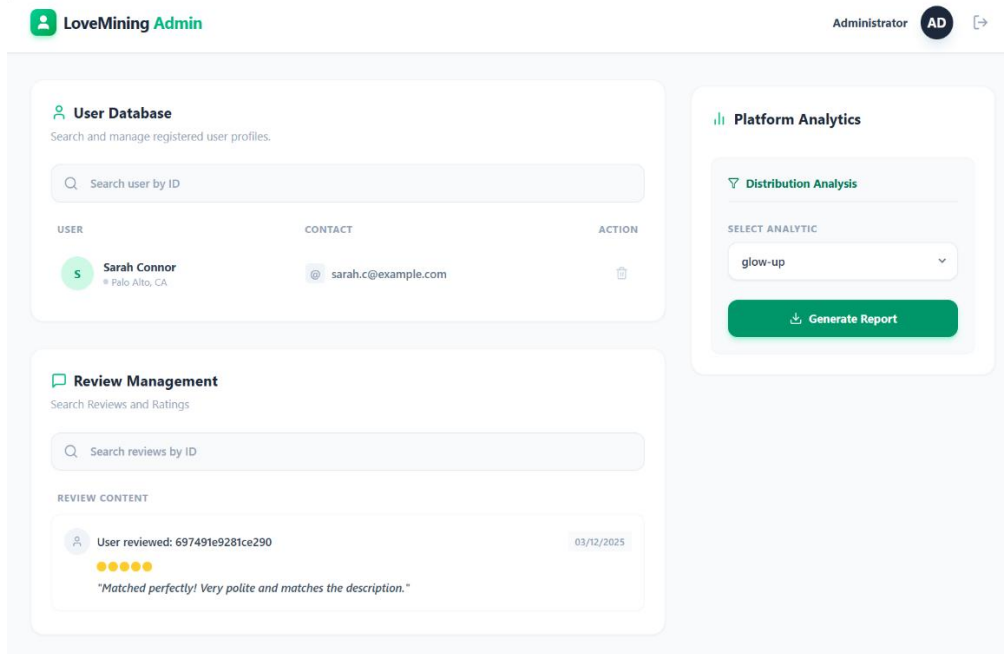
18 50+

Apply Filters

Log Out

3.2.3. Admin Homepage Mock-up

This interface is designed for the Administrator. On the left side, there is a search tool used to find specific user profiles and, if necessary, delete them from the platform. At the bottom, the admin can analyse the feedbacks of the users by searching for reviews. Finally, on the right side, the dashboard provides a set of controls to run pre-configured analytics on the dataset.



3.3. Requirement

3.3.1. Functional requirements

Unregistered User

- The system must allow an Unregistered User to create a new account by providing credentials (email, password), some mandatory personal data (age, gender, status, sexual orientation, city and state), some optional personal data (like income, religion, pets, etc.) and a personal essay describing his interests.

Registered User

- The system must allow the User to modify their personal attributes, including optional fields and the descriptive essay.
- The system must suggest candidates to the User with high compatibility, calculated on the overlap of extracted interests, considering that the two Users must have sexual orientation compatibility, the desired age range, and must reside in the same city or state chosen.
- The system must allow the User to view the detailed profile of a candidate.
- The system must allow the User to express a positive (Like) or negative (Dislike) preference towards another profile. Upon a reciprocal Like, the system must atomically create a MATCHED relationship in the Graph Database and remove the preliminary unidirectional relationship.
- The system must allow the User to retrieve and view a list of their current confirmed matches.

- The system must allow the User to write a review for a confirmed match, providing a numerical rating and a textual comment.
- The system must prevent the suggestion of users who have already been interacted with or who do not meet the defined sexual orientation constraints.

Administrator

- The system must allow the Administrator to search for specific users and permanently delete their profiles from the database.
- The system must allow the Administrator to search for specific reviews.
- The system must allow the Administrator to execute and view aggregated statistics on the user population.
- The system must allow the Administrator to execute and view aggregated statistics on the reviews.

System

- Upon saving (creation or modification) a personal essay, the system must automatically process the text to extract relevant keywords for interest compatibility.

3.3.2. Non-Functional requirements

Product requirements

- The system must ensure eventual consistency between the Document Database (MongoDB) and the Graph Database (Neo4j). A temporal misalignment in the order of seconds is tolerated, provided that the data ultimately converges.
- The system must ensure high availability, even if the latest updates (e.g., new likes) are not yet fully propagated.
- The system must implement fault tolerance mechanisms to prevent data loss in the event of a single node failure within the database cluster.
- The system must execute core queries with a fast response time.
- The system must be capable of handling an increasing amount of work and data without compromising performance.
- The system must use a basic role-based authorization protocol.

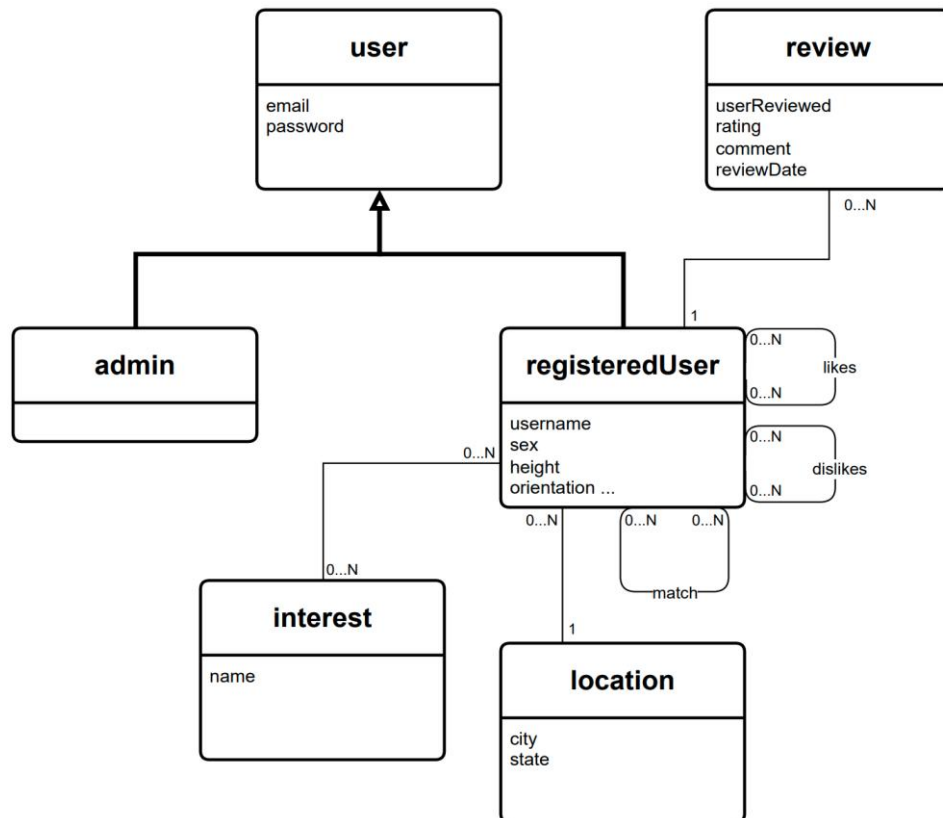
Organizational requirements

- The system must be developed using Object-Oriented Programming (OOP) languages.
- The system must use both a Document-Oriented Database and a Graph Database.
- The system must expose its functionalities via RESTful APIs, documented according to the OpenAPI standard.

Privacy requirements

- The system must store user and administrator passwords in the database using strong encryption hashing algorithms.

3.4. UML Class Diagram



3.5. Data Models

The system architecture integrates two distinct databases paradigms; each selected for its specific strengths.

This hybrid model allows us to optimize both data storage and relationship analysis simultaneously.

3.5.1. Document DB

The implementation of the Document-Oriented database focuses on managing semi-structured data, specifically user profiles and their associated feedback.

We selected this paradigm primarily for its schema flexibility. As highlighted during the dataset analysis, user profiles show significant structural variability due to the presence of numerous optional fields. A document model allows us to store these heterogeneous profiles as single, self-contained JSON objects, optimizing read performance by reducing the need for expensive JOIN operations during data retrieval.

3.5.1.1. Users Collection

The Users Collection serves as the central aggregate of the system.

Each document represents a single user and encapsulates authentication credentials (email, password) and all high-priority personal information, such as orientation, location, and the essay descriptions used for text analysis.

To further enhance query efficiency, we implemented the Partial Embedding for the review section. Instead of storing the full content of every review made, the user document embeds a lightweight array containing only the essential metadata of the reviews (such as the review ID, the target user ID, and the numerical rating).

This design choice allows the application to display a user's profile and their rating summary in a single database fetch, keeping the document size manageable while maximizing access speed for real-time analytics.

```
_id: "697491e9281ce290af403853"
Email: "user224@largescale.it"
Password: "$2b$12$sXxvaQq9aKaH0e621vnfB0dJ1oLLxnssPkWsdgtgBKTbhekWPLtVUUi"
is_admin: false
age: 42
status: "single"
sex: "f"
orientation: "straight"
body_type: "athletic"
diet: "anything"
drinks: "socially"
education: "working on space camp"
ethnicity: "white"
height: 63
income: -1
job: "medicine / health"
city: "lafayette"
state: "california"
offspring: "has kids"
pets: "has dogs and has cats"
religion: "christianity and somewhat serious about it"
smokes: "no"
speaks: "english (fluently)"
essay0: "hi, thanks for checking out my profile. i'm a diverse multi- faceted p..."
interests: Array (5)
  0: "food"
  1: "garden"
  2: "fishing"
  3: "wine"
  4: "bike"
reviews_made: Array (2)
  0: Object
    review_id: "6975e95d135e8063fcbafc6a"
    target_id: "697491ea281ce290af406521"
    rating: 5
  1: Object
    review_id: "6975e95d135e8063fcbafc6f"
    target_id: "697491ea281ce290af406dd6"
    rating: 5
```

3.5.1.2. Reviews Collection

While the Users collection holds a summary, the Reviews Collection stores the complete historical record of users' reviews.

Each document in this collection reflects the full match feedback, containing also the textual comment and the precise timestamp. This separation ensures that the main user documents remain lightweight, while the full review data remains available for detailed inspection.

The reviewer_id was intentionally omitted in this collection to avoid ending up in a relational model.

```
_id: "6975e95d135e8063fcbafc62"  
target_id : "697491e9281ce290af403805"  
rating : 5  
comment : "Match of the year! The algorithm did its magic."  
review_date : 2025-07-28T05:12:06.000+00:00
```

3.5.2. Graph DB

A Graph DB was adopted to manage the system's relationships. This choice was driven by the need to treat complex user interactions (Likes, Dislikes, Matches) as main features. Compared to traditional databases, the graph structure ensures high efficiency for the traversal of the connections, making it ideal for modelling the natural topology of a dating app based on geographical proximity and shared interests.

3.5.2.1. Graph Nodes

User: The central nodes of the graph. It maintains a lightweight structure, containing only the ID and essential demographic attributes for mandatory constraints during matching queries (gender, age, sexual orientation).

Interest: Represents a specific passion or activity. The connection of multiple users to the same Interest node is considered a critical element for calculating compatibility and identifying shared affinities between users.

Location (City & State): User location is modelled in a hierarchical way across two distinct nodes. The user is connected to the City node, which is related to the State node. This structure optimizes geospatial search by allowing the user's search radius to be gradually expanded from a single city to the entire state.

3.5.2.2. Graph Relationships

(User) - [DISLIKES] -> (User): Unidirectional relationship that expresses a user's rejection towards another user and excludes them from their future suggestions.

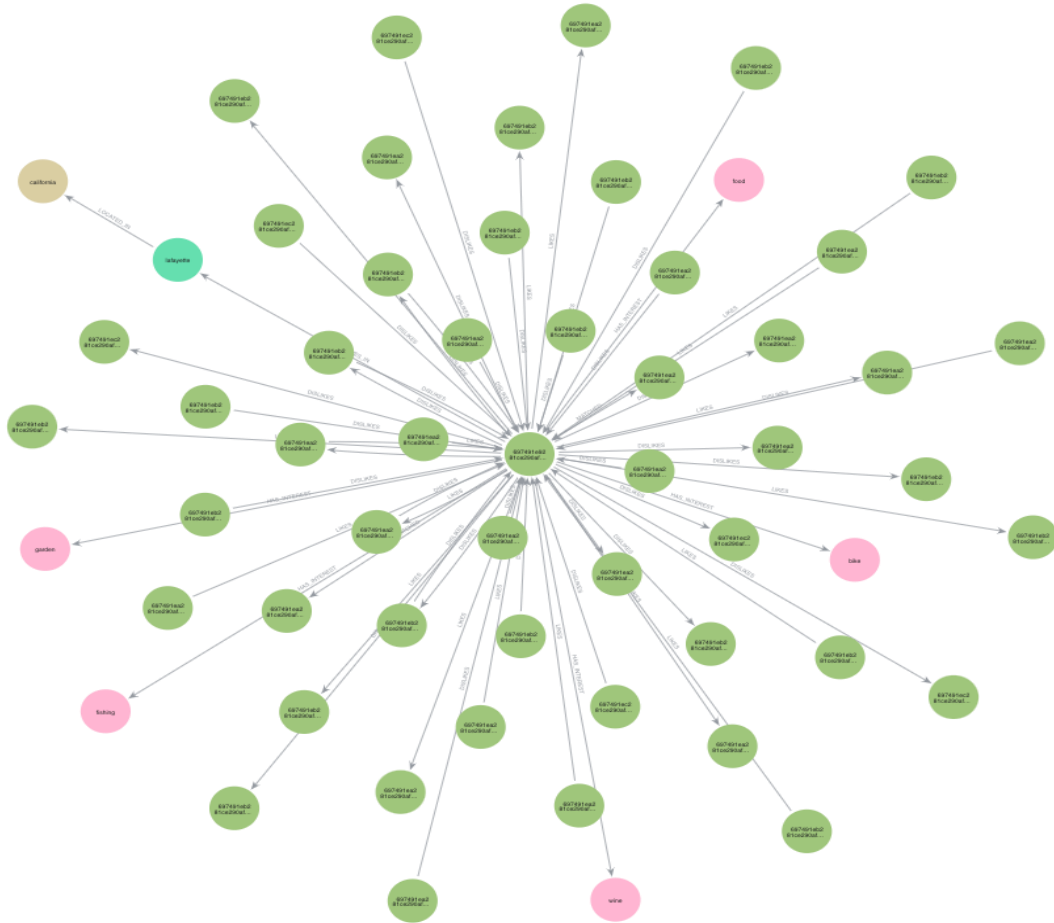
(User) - [LIKES] -> (User): Unidirectional relationship expressing the interest of one user towards another.

(User) - [MATCHED] - (User): Bidirectional relationship generated when two users exchange a like. This change of state represents the evolution from a unilateral preference to a confirmed reciprocal interaction. Changing two opposite edges into a single MATCH relationship optimizes queries by avoiding the search for pairs of reciprocal edges and enables the possibility for the two users to insert reviews.

(User) - [HAS_INTEREST] -> (Interest): Connects users to their passions, creating the prerequisites for calculating compatibility between users.

(User) - [LIVES_IN] -> (City) and (City) - [LOCATED_IN] -> (State): Chain of relationships that establishes the geographical position and allows users to be searched based on their location (city and state) following a hierarchical structure.

3.5.2.3. Example



3.6. Distributed database design

According to the non-functional requirements the system is designed to be always online and ensuring a quick response.

According to the CAP theorem, it is impossible to have Consistency, Availability, and Partition Tolerance all at the same time.

For LoveMining, we chose the AP approach (Availability and Partition Tolerance). This means:

- **Availability:** the system must always respond to user requests, even if some servers are down.
- **Partition Tolerance:** the system continues to work even if there are network problems between servers.

To achieve this, we accept Eventual Consistency. We decided to have an always available database, capable of surviving network failures, even if it takes time for all nodes to be perfectly synchronized.

3.6.1. Replica set

The MongoDB database is deployed on a Replica Set spanning a cluster of 3 nodes.

To control the election mechanism effectively, we assigned different priority values (5, 2, and 1) to the members. The node with Priority 5 (10.1.1.16) is the preferred Primary and handles all write operations, while the node with Priority 2 (10.1.1.15) serves as the immediate failover. The third node, assigned Priority 1 (10.1.1.14), is designed to remain a permanent Secondary.

To optimize resource usage, we installed the Neo4j instance specifically on the node with Priority 1. Since graph traversals require significant power, placing Neo4j on a node that only handles light read replication (and is unlikely to become Primary) prevents resource contention and ensures system stability.

3.6.1.1. Read Operations

To maximize the system's responsiveness and minimize network latency, we configured the Read Preference to nearest.

This mode instructs the driver to route read operations to the replica set member with the lowest network latency, regardless of whether it is a Primary or a Secondary node. This choice follows our non-functional requirements.

3.6.1.2. Write Operations

For write operations, our priority is throughput rather than strict durability. Therefore, we configured the Write Concern with the following parameters:

- `w: 1`: The application requires acknowledgment only from the Primary node before considering the write successful. We do not wait for the data to be replicated to the secondaries.
- `j: false`: We request acknowledgment as soon as the operation is applied in memory, without waiting for the on-disk journal flush.

This configuration provides the lowest possible write latency, by avoiding the wait times associated with network replication (`w: majority`) and disk I/O (`j: true`).

3.6.2. Sharding

Given the current infrastructure and the limited dimension of our total dataset (~110MB), which fits in the memory of any single node, we decided not to implement sharding at this stage. Partitioning such a small volume of data would introduce unnecessary overhead without providing any performance gain.

In case of a significant growth of the number of users in the future, causing also an increment on of the interactions among them, the current strategy will become unsustainable. So, we analysed potential sharding keys to adopt for the users' collection:

- Sharding on the hashed User ID: this approach would guarantee a perfectly even distribution of data, but it would force our queries - which need to match within a city - to broadcast the request to every shard and merge the results, causing high latency.

- Sharding only on location: we first evaluated the possibility of using a single location field (State or City) as the shard key since our service is concentrated on locality. However, this approach brings to a problem of low cardinality: our dataset doesn't have a uniform distribution, most of our users live in a big city or a populous state (such as San Francisco or California) and this choice would create too big shards that are not divisible.

To solve the distribution issues while maintaining the benefits of our locality logic, we identified adding the hashed IDs of the users to the shard key as the necessary solution.

We compared the two specific implementations: sharding on State and user ID and sharding on City and user ID. Although both options could be a good solution, since the majority of our system's queries involve grouping by city, the best choice would be sharding on city and hashed user ID.

The growth of the number of users would increment also the reviews' database. The best sharding key to adopt on this collection is driven by the query "Glow Up" which would only benefit the sharding on the hashed target_id.

3.6.3. Inter Database Consistency

Since MongoDB and Neo4j are not automatically connected, we use our Java application (Spring Boot) to act as a coordinator.

We identified three specific scenarios where the code manually executes operations on both databases to ensure they stay in sync.

1. User Registration

When a new user registers, the system follows this strict order:

Validation: Checks input data.

MongoDB: Saves the user in MongoDB first to generate the unique _id.

Neo4j sync: Immediately tries to create a User Node in Neo4j using the same _id. If this step fails, the system automatically deletes the user from MongoDB.

2. Profile Update

When a user updates their profile:

Validation: Checks input data.

Neo4j: If graph-related data changes, the system attempts to update Neo4j first. If this update fails, the operation stops immediately.

MongoDB: Only if the Neo4j update is successful, the system saves the changes to the MongoDB document.

3. User Deletion

To remove a user completely, the system performs a "Safe Deletion":

Neo4j: The system attempts to remove them from the graph first. If the graph database is down, the operation stops.

MongoDB: Only after the node is successfully removed, the system deletes the User document and all associated reviews from MongoDB.

We installed Neo4j on a server that also acts as a MongoDB Secondary Node.

This configuration does not affect consistency because the connections are managed separately:

- MongoDB: The application always sends Write operations to the Primary Node of the cluster.
- Neo4j: The application connects directly to the Neo4j instance on its specific IP.

Even though they share the same physical machine, the two databases are logically separate. The Java application ensures that data is written to both correctly.

4. Implementation

4.1. Spring

We built the backend using Spring Boot 3. This framework allows us to manage two different databases (MongoDB and Neo4j) in the same project.

We customized the MongoDB configuration to remove the `_class` field from the database documents, this keeps the database clean and saves storage space.

For security, we used Spring Security. We implemented BCrypt to encrypt user passwords, so they are never stored as plain text.

The system also separates normal "Users" from "Admins" to control access to the data and giving authorizations to the specific RESTful API Endpoints.

4.2. Models

We divided the application's model into two separate folders: "mongo" and "neo4j".

4.2.1. MongoDB Models

User Model

```
public class UserDocument {
    @MongoId(FieldType.STRING)
    private String id;
    @Field("Email")
    private String email;
    @Field("Password")
    private String password;
    @Field("is_admin")
    private Boolean isAdmin;
    private Integer age;
    private String status;
    private String sex;
    private String orientation;
    @Field("body_type")
    private String bodyType;
    private String diet;
    private String drinks;
    private String education;
    private String ethnicity;
    private Integer height;
    private Integer income;
    private String job;
    private String offspring;
    private String pets;
    private String religion;
    private String smokes;
    private String speaks;
    private String city;
    private String state;
    private String essay0;
    private java.util.List<String> interests;
    @Field("reviews_made")
    private java.util.List<ReviewSummary> reviewsMade;
    @Data 7 usages
    @NoArgsConstructor
    @AllArgsConstructor
    public static class ReviewSummary {
        @Field("review_id")
        private String reviewId;

        @Field("target_id")
        private String targetId;

        private Integer rating;
    }
}
```

Review Model

```
public class ReviewDocument {  
  
    @MongoId(FieldType.STRING)  
    private String id;  
  
    @Field("target_id")  
    private String targetId;  
    private Integer rating;  
    private String comment;  
  
    @Field("review_date")  
    private LocalDateTime date;  
}
```

4.2.2. Neo4j Models

User Model

```
public class UserNode {  
  
    @Id  
    @Property("_id")  
    private String id;  
  
    private Integer age;  
    private String sex;  
    private String orientation;  
  
    @lombok.EqualsAndHashCode.Exclude  
    @lombok.ToString.Exclude  
    @Relationship(type = "LIVES_IN", direction = Relationship.Direction.OUTGOING)  
    private CityNode cityNode;  
  
    @lombok.EqualsAndHashCode.Exclude  
    @lombok.ToString.Exclude  
    @Relationship(type = "HAS_INTEREST", direction = Relationship.Direction.OUTGOING)  
    private java.util.Set<InterestNode> interests;  
  
    @com.fasterxml.jackson.annotation.JsonIgnore  
    @lombok.EqualsAndHashCode.Exclude  
    @lombok.ToString.Exclude  
    @Relationship(type = "LIKES", direction = Relationship.Direction.OUTGOING)  
    private java.util.Set<UserNode> likes;  
  
    @com.fasterxml.jackson.annotation.JsonIgnore  
    @lombok.EqualsAndHashCode.Exclude  
    @lombok.ToString.Exclude  
    @Relationship(type = "DISLIKES", direction = Relationship.Direction.OUTGOING)  
    private java.util.Set<UserNode> dislikes;  
}
```

Interest Model

```
public class InterestNode {  
    @Id  
    private String name;  
}
```

City Model

```
public class CityNode {  
  
    @Id  
    private String name;  
  
    @Relationship(type = "LOCATED_IN", direction = Relationship.Direction.OUTGOING)  
    private StateNode state;  
}
```

State Model

```
public class StateNode {  
    @Id  
    private String name;  
}
```

4.3. Repository

The repository layer provides an abstraction for database operations, allowing the application to interact with both MongoDB and Neo4j. By extending Spring Data interfaces, these repositories simplify data access, offering methods for common operations while also supporting custom queries for more complex requirements.

- **UserMongoRepository**

Manages user documents in MongoDB. It provides method for retrieving users by the email but its most advanced feature is the implementation of Analytics pipelines

- **ReviewMongoRepository**

Manages the storage of review documents in MongoDB. Although reviews are partially embedded within user profiles for performance, this repository allows for independent querying and analysis of the full review data.

- **UserNeo4jRepository**

Handles the graph-based representation of users in Neo4j. It is responsible for managing social interactions, such as "LIKES" and "MATCHED" relationships, and executing simple queries and Analytics queries for finding matches and recommendations.

- **CityNeo4jRepository & StateNeo4jRepository**

These repositories manage the geographical entities in the Neo4j graph, allowing users to be linked to specific locations.

- **InterestNeo4jRepository**

Manages the Interest nodes in the graph. It enables the system to link users to shared interests, facilitating the discovery of potential matches based on common hobbies or passions.

4.4. Service

The service layer serves as the bridge between the repository and the controller, encapsulating the core business logic of the LoveMining application. It orchestrates operations between MongoDB and Neo4j to ensure data consistency and integrity.

- **AuthenticationService**

Handles the critical processes of user registration. It manages the creation of new user accounts, ensuring that a consistent identity is established in both MongoDB and Neo4j. It checks all fields entered by the user, returning an error if any required fields are missing. It then creates the Document and Node for the new user in both databases.

- **UserService**

Implements the core functionalities related to user profile management and social interactions. It handles updating user details (ensuring changes propagate to both databases where necessary), extracting interests from user essays, and managing the "Like/Dislike" logic that drives the matching system. It also provides methods for retrieving user matches and their details.

- **AdminService**

Provides administrative capabilities for the system. It includes basic queries to retrieve or delete users and to retrieve reviews. Furthermore, it also includes analytics queries to calculate statistics.

- **InterestExtractorService**

A utility service dedicated to processing natural language text from user essays. It employs keyword extraction logic to automatically identify and normalize user interests, which are then used to create relationships in the Neo4j graph.

- **CustomUserDetailsService**

Integrates with Spring Security to load user's data during the authentication process, enabling a secure access control.

4.5. Controller

The Controller layer manages the REST API. It receives the HTTP requests and sends them to the Service layer. The controller relies on the Service to clean and validate the input data. It returns the correct status codes, such as "201 Created" for a new user or "400 Bad Request" if some data is missing. We divided this layer in three modules, one for each type of actor considering also unregistered users.

All modules are subject to basic authentication thanks to the security config module, which manages the necessary permissions to execute operations in each module depending on whether the actor is a user or an administrator.

4.5.1. Authentication Controller

This module is responsible for new users. In particular, it manages the registration of new users delegating the validation of the input data to the Authentication Service. If validation is successful, it sends back a positive response with the user's new profile, otherwise, it provides the error message. The registration operation is the only one allowed for unauthenticated users.

4.5.2. User Controller

This controller maps all functionalities for registered users. It is the central point for user profile management, enabling users to view and update their details, and to access others' profiles. It is also possible for users to ask the system to generate recommendations based on filters defined by them. Moreover, users can indicate their interest through like/dislike mechanisms and offer their opinions on the matches they made by submitting a review.

4.5.3. Admin Controller

The Admin Controller manages all the functionalities of the system administrator. Specifically, an administrator can search users' profiles by their ID and delete them. He can also see the reviews made by users and perform a series of analytics useful for generating statistics that can bring modifications to the system, improving its performance or the user experience.

4.6. RESTful API Endpoints

To allow the frontend to interact with the databases and trigger business logic, the application exposes several RESTful APIs. These endpoints are documented and tested using Swagger UI. The routes are divided into three main categories based on the actor's role and privileges.

Authentication Endpoints

POST	/api/authentication/register	🔒 ▼
------	------------------------------	-----

- **“POST /api/authentication/register”**: Allows a new user to create an account by providing their personal details.

The “POST/api/authentication/login” endpoint was intentionally omitted to stay in the Stateless RESTful principles.

User Endpoints

POST	/api/users/{id}/review	🔒 ▼
POST	/api/users/{id}/like	🔒 ▼
POST	/api/users/{id}/dislike	🔒 ▼
PATCH	/api/users/me	🔒 ▼
GET	/api/users	🔒 ▼
GET	/api/users/{id}	🔒 ▼
GET	/api/users/reviews	🔒 ▼
GET	/api/users/recommendations/{filters}	🔒 ▼
GET	/api/users/matches	🔒 ▼

- **“POST /api/users/{id}/review”**: Allows a user to leave a text review and a rating for a matched profile with the specified {id}.
- **“POST /api/users/{id}/like”** and **“POST /api/users/{id}/dislike”**: Enables the user to express interest or disinterest in another profile specified by the {id}, updating the graph relationships.
- **“PATCH /api/users/me”**: Allows the logged-in user to update their personal information.
- **“GET /api/users”** and **GET /api/users/{id}**: Retrieves the personal information of the logged-in user or a specific profile.
- **“GET/api/users/reviews”**: Retrieves the list of reviews made by the logged-in user.
- **“GET/api/users/recommendations/{filters}”**: Fetches a personalized list of suggested profiles based on graph algorithms (shared interests), location, and user-defined filters.
- **“GET /api/users/matches”**: Retrieves the list of successful mutual likes (matches) for the current user.

Admin Endpoints

GET	/api/admin/userReviews/{id}	🔒 ▼
GET	/api/admin/user/{id}	🔒 ▼
GET	/api/admin/analytics/unhappy-cities	🔒 ▼
GET	/api/admin/analytics/status-by-age-group	🔒 ▼
GET	/api/admin/analytics/orientation-by-age-group	🔒 ▼
GET	/api/admin/analytics/love-points	🔒 ▼
GET	/api/admin/analytics/glow-up	🔒 ▼
DELETE	/api/admin/users/{id}	🔒 ▼

- **“GET /api/admin/userReviews/{id}”**: Retrieves a specific review by its associated {id}, for moderation purposes.
- **“GET /api/admin/user/{id}”**: Allows the admin to inspect a user's complete profile data.
- The APIs related to the **“analytics”** provide the administrator with advanced business intelligence tools.
- **“DELETE /api/admin/users/{id}”**: Completely removes the user specified by the {id} and their associated data from the platform.

Performed Tests

All the APIs described above has been successfully tested and has given positive results in line with those expected.

5. Most relevant queries

5.1. MongoDB Queries

This section details the analytics implemented using aggregation pipelines and shows some examples of the results.

5.1.1. Unhappy Cities

The analytic operates on the User collection, processing the “city” field and the embedded “reviews_made” array. Its primary goal is to identify the top five cities with unhappy users based on the negativity of the reviews they have made, dividing the users into two distinct categories (Extreme and Moderate) based on the severity of their ratings.

The aggregation does:

1. Match: Keep only users who have made reviews.
2. Project: Calculate the average rating for each user.
3. Match: Filter for "unhappy" users (average rating less than 3).
4. Group: Group by City. Count total unhappy users and split by severity.
5. Project: Format output and round the score to 2 decimal places.
6. Sort: Order by the highest number of unhappy users (descending).
7. Limit: Return only the top 5 cities.

```
@Aggregation(pipeline = { 1 usage
    "{ '$match': { 'reviews_made.rating': { '$gt': 0 } } }",
    "{ '$project': { 'city': 1, 'userPersonalAvg': { '$avg': '$reviews_made.rating' } } }",
    "{ '$match': { 'userPersonalAvg': { '$lt': 3 } } }",
    "{ '$group': { " +
        "'_id': '$city', " +
        "'totalUnhappyUsers': { '$sum': 1 }, " +
        "'extremeHatersCount': { '$sum': { '$cond': [ { '$lt': [ '$userPersonalAvg', 2 ] }, 1, 0 ] } }, " +
        "'moderateUnhappyCount': { '$sum': { '$cond': [ { '$gte': [ '$userPersonalAvg', 2 ] }, 1, 0 ] } }, " +
        "'avgUnhappinessScore': { '$avg': '$userPersonalAvg' } } }",
    "{ '$project': { " +
        "'totalUnhappyUsers': 1, " +
        "'extremeHatersCount': 1, " +
        "'moderateUnhappyCount': 1, " +
        "'avgUnhappinessScore': { '$round': [ '$avgUnhappinessScore', 2 ] } } }",
    "{ '$sort': { 'totalUnhappyUsers': -1 } }",
    "{ '$limit': 5 }"
})
List<Map<String, Object>> findTopUnhappyCities();
```

An example of the results is the list of documents in the image below, where each document presents the city name and its aggregate statistics:

```
[
  {
    "_id": "el cerrito",
    "totalUnhappyUsers": 16,
    "extremeHatersCount": 7,
    "moderateUnhappyCount": 9,
    "avgUnhappinessScore": 1.59
  },
  {
    "_id": "redwood city",
    "totalUnhappyUsers": 16,
    "extremeHatersCount": 6,
    "moderateUnhappyCount": 10,
    "avgUnhappinessScore": 1.69
  },
  {
    "_id": "stanford",
    "totalUnhappyUsers": 15,
    "extremeHatersCount": 2,
    "moderateUnhappyCount": 13,
    "avgUnhappinessScore": 1.97
  },
]
```

5.1.2. Glow Up

The second analytic operates on the Review collection. Its primary goal is to identify users who have significantly improved their reputation over time. It compares the average ratings received in the last six months against those received in the past, calculating a "Glow Up Index".

This allows the administrator to find users who have improved their profile over time.

The aggregation does:

1. Group: Sum ratings and count reviews for recent and past periods.
2. Match: Filter users having activity in both periods and at least 3 reviews total.
3. Project: Calculate averages.
4. Sort: Sort users in a descent order.
5. Limit: Limit by top 3 users.

```

@Aggregation(pipeline = { 1 usage
    "{ $group: { _id: '$target_id', " +
        "    recentRatingSum: { $sum: { $cond: [{ $gte: ['$review_date', ?0] }, '$rating', 0] } }, " +
        "    recentCount: { $sum: { $cond: [{ $gte: ['$review_date', ?0] }, 1, 0] } }, " +
        "    pastRatingSum: { $sum: { $cond: [{ $lt: ['$review_date', ?0] }, '$rating', 0] } }, " +
        "    pastCount: { $sum: { $cond: [{ $lt: ['$review_date', ?0] }, 1, 0] } }, " +
        "    totalCount: { $sum: 1 } } }",

    "{ $match: { recentCount: { $gt: 0 }, pastCount: { $gt: 0 }, totalCount: { $gte: 3 } } }",

    "{ $project: { " +
        "    recentAvg: { $divide: ['$recentRatingSum', '$recentCount'] }, " +
        "    pastAvg: { $divide: ['$pastRatingSum', '$pastCount'] }, " +
        "    glowUpIndex: { $subtract: [ " +
        "        { $divide: ['$recentRatingSum', '$recentCount'] }, " +
        "        { $divide: ['$pastRatingSum', '$pastCount'] } " +
        "    ] } } }",

    "{ $sort: { glowUpIndex: -1 } }",

    "{ $limit: 3 }"
})
List<Map<String, Object>> findGlowUpRaw(Date cutoffDate);

default List<Map<String, Object>> findBestGlowUpUsers() { 1 usage
    Date sixMonthsAgo = Date.from(LocalDate.now().atZone(ZoneId.systemDefault()).toInstant().minusMonths(6).atZone(ZoneId.systemDefault()).toInstant());

    return findGlowUpRaw(sixMonthsAgo);
}

```

An example of the query output is provided below, displaying the top users with their historical rating averages and the resulting Glow Up Index:

```

[
  {
    "_id": "697491ec281ce290af411a90",
    "recentAvg": 4,
    "pastAvg": 2.5,
    "glowUpIndex": 1.5
  },
  {
    "_id": "697491eb281ce290af40d831",
    "recentAvg": 4,
    "pastAvg": 3,
    "glowUpIndex": 1
  },
  {
    "_id": "697491ea281ce290af4048c8",
    "recentAvg": 4,
    "pastAvg": 3,
    "glowUpIndex": 1
  }
]

```

5.1.3. Status by Age Group

The third analytic focuses on demographic distribution inside the User collection. By analysing the “age” and the “status” fields, it groups users into three age categories and calculates the percentage of singles for each group.

This allows the administrator to understand the composition of the community and the most active age groups looking for a partner.

The aggregation does:

1. Match: Filter users with a valid status.
2. Project: Determine Age Group (1: Young, 2: Adult, 3: Senior).
3. Group: Group by the calculated “ageGroup”.
4. Project: Calculate Percentage.
5. Sort: Sort by group name (1 → 2 → 3).

```
@Aggregation(pipeline = { 1 usage
  "{ '$match': { 'status': { '$in': ['available', 'single', 'seeing someone', 'married'] } } }",
  "{ '$project': { ' +
    "    '_id': 0, " +
    "    'status': 1, " +
    "    'ageGroup': { '$cond': [ { '$lte': [ '$age', 25 ] }, '1. Young (18-25)', " +
    "                        { '$lte': [ '$age', 40 ] }, '2. Adult (26-40)', " +
    "                        { '$lte': [ '$age', 40+ ] }, '3. Senior (40+)' ] ] } } }",
  "{ '$group': { ' +
    "    '_id': '$ageGroup', " +
    "    'totalUsers': { '$sum': 1 }, " +
    "    'singleCount': { '$sum': { '$cond': [ { '$in': [ '$status', ['single', 'available'] ] }, 1, 0 ] } } }",
  "{ '$project': { ' +
    "    '_id': 0, " +
    "    'ageGroup': '$_id', " +
    "    'totalUsers': 1, " +
    "    'singlePercentage': { '$round': [ { '$multiply': [ { '$divide': [ '$singleCount', '$totalUsers' ] }, 100 ] }, 1 ] } } }",
  "{ '$sort': { 'ageGroup': 1 } }"
})
List<Map<String, Object>> findSinglesByAgeGroup();
```

The image below presents a sample of the aggregated results, illustrating the total number of users and the calculated percentage of singles for each defined age bucket:

```
[
  {
    "totalUsers": 14453,
    "_id": "0",
    "ageGroup": "1. Young (18-25)",
    "singlePercentage": 95.7
  },
  {
    "totalUsers": 35156,
    "_id": "0",
    "ageGroup": "2. Adult (26-40)",
    "singlePercentage": 95.7
  },
  {
    "totalUsers": 10327,
    "_id": "0",
    "ageGroup": "3. Senior (40+)",
    "singlePercentage": 97.7
  }
]
```

5.1.4. Orientation by Age Group

The fourth analytic is built upon the logic of the previous one but focuses on the orientation. The key difference here is the use of a double grouping. Instead of a simple flat count, this query first aggregates users by a combination of “orientation” and “age”, then restructures the data to produce a nested list.

This allows the administrator to visualize not only the total number of users for each sexual orientation, but also the specific age distribution of each category.

The aggregation does:

1. Match: Filter valid data.
2. Project: Calculate Age Group (Same logic as before).
3. Group #1: Group by combination of “orientation” and “ageGroup”.
4. Sort: Sort by group name (1 → 2 → 3).
5. Group #2: Group only by “orientation”.
6. Sort: Sort by total users in descending order.

```
@Aggregation(pipeline = { 1 usage
  "{ '$match': { 'orientation': { '$in': ['straight', 'gay', 'bisexual'] } } }",
  "{ '$project': { " +
    "  '_id': 0, " +
    "  'orientation': 1, " +
    "  'ageGroup': { '$cond': [ { '$lte': ['$age', 25] }, '1. Young (18-25)', " +
    "    '{ '$cond': [ { '$lte': ['$age', 40] }, '2. Adult (26-40)', " +
    "      '3. Senior (40+)' ] } ] } } }",
  "{ '$group': { " +
    "  '_id': { 'orientation': '$orientation', 'ageGroup': '$ageGroup' }, " +
    "  'count': { '$sum': 1 } } }",
  "{ '$sort': { '_id.ageGroup': 1 } }",
  "{ '$group': { " +
    "  '_id': '$_id.orientation', " +
    "  'totalUsers': { '$sum': '$count' }, " +
    "  'demographics': { '$push': { 'ageGroup': '$_id.ageGroup', 'usersCount': '$count' } } } }",
  "{ '$sort': { 'totalUsers': -1 } }"
})
List<Map<String, Object>> findOrientationDemographics();
```

A sample of the result set is shown in the two following images, with a hierarchical structure where each sexual orientation includes an array of nested documents along with demographic statistics divided by age group:

```
[
  {
    "_id": "straight",
    "totalUsers": 51606,
    "demographics": [
      {
        "ageGroup": "1. Young (18-25)",
        "usersCount": 11937
      },
      {
        "ageGroup": "2. Adult (26-40)",
        "usersCount": 30687
      },
      {
        "ageGroup": "3. Senior (40+)",
        "usersCount": 8982
      }
    ]
  },
  {
    "_id": "gay",
    "totalUsers": 5573,
    "demographics": [
      {
        "ageGroup": "1. Young (18-25)",
        "usersCount": 1445
      },
      {
        "ageGroup": "2. Adult (26-40)",
        "usersCount": 3059
      },
      {
        "ageGroup": "3. Senior (40+)",
        "usersCount": 1069
      }
    ]
  },
  {
    "_id": "bisexual",
    "totalUsers": 2767,
    "demographics": [
      {
        "ageGroup": "1. Young (18-25)",
        "usersCount": 1072
      },
      {
        "ageGroup": "2. Adult (26-40)",
        "usersCount": 1417
      },
      {
        "ageGroup": "3. Senior (40+)",
        "usersCount": 278
      }
    ]
  }
]
```

5.2. Neo4j Queries

This section covers the graph algorithms implemented in Neo4j using Cypher. For each query, the execution logic is detailed and an example of the possible results is provided.

5.2.1. Recommendation mechanism

This query operates on the graph structure connecting User, City, State, and Interest nodes.

Its primary goal is to generate a personalized list of ten compatible partners by traversing the relationship network. The recommendation engine applies a strict set of filters (including geographic proximity, age range, and sexual orientation compatibility), while prioritizing candidates who share the highest number of common interests.

The query does:

1. Match: Identifies the user and traverses the graph to locate their City and State
2. Filter: Excludes candidates with whom a relationship (LIKES, DISLIKES, or MATCHED) already exists
3. Filter: Applies a dynamic constraint to filter candidates by Age and Location
4. Filter: Enforces a complex logic check to ensure the candidate's Sex and Orientation align with the user's preferences
5. Scoring: Counts the number of shared interests between the user and the candidate
6. Return: Returns the ordered IDs of the top 10 candidates

```
@org.springframework.data.neo4j.repository.query.Query( 1 usage
    "MATCH (me:User {_id: $userId}) " +
    "MATCH (me)-[:LIVES_IN]->(myCity:City)-[:LOCATED_IN]->(myState:State) " +
    "MATCH (candidate:User)-[:LIVES_IN]->(candCity:City)-[:LOCATED_IN]->(candState:State) " +
    "WHERE me._id <> candidate._id " +

    "AND NOT (me)-[:LIKES|DISLIKES|MATCHED]-(candidate) " +
    "AND candidate.age >= $minAge AND candidate.age <= $maxAge " +
    "AND ( ($filter = 'City' AND myCity = candCity) OR " +
    "      ($filter = 'State' AND myState = candState) )" +

    "AND ( " +
    "(me.orientation = 'straight' AND candidate.orientation = 'straight' AND me.sex <> candidate.sex) OR" +
    "(me.orientation = 'straight' AND candidate.orientation = 'bisexual' AND me.sex <> candidate.sex) OR" +
    "(me.orientation = 'gay' AND candidate.orientation = 'gay' AND me.sex = candidate.sex) OR" +
    "(me.orientation = 'gay' AND candidate.orientation = 'bisexual' AND me.sex = candidate.sex) OR" +
    "(me.orientation = 'bisexual' AND candidate.orientation = 'gay' AND me.sex = candidate.sex) OR" +
    "(me.orientation = 'bisexual' AND candidate.orientation = 'straight' AND me.sex <> candidate.sex) OR" +
    "(me.orientation = 'bisexual' AND candidate.orientation = 'bisexual') " +
    ")" +

    "WITH me, candidate" +
    "OPTIONAL MATCH (me)-[:HAS_INTEREST]->(i:Interest)-[:HAS_INTEREST]-(candidate) " +
    "WITH candidate, count(i) AS commonInterests" +

    "RETURN candidate._id " +
    "ORDER BY commonInterests DESC " +
    "LIMIT 10")
List<String> findRecommendations(@Param("userId") String userId, @Param("filter") String filter,
                                @Param("minAge") int minAge, @Param("maxAge") int maxAge);
```

An example of the output is shown in the image below, which returns a simple array containing the unique IDs of the top 10 recommended candidates for the user of ID “697491e9281ce290af403774” using as filters: State as location scope, and an age range between 18 and 25 years.

```
[  
  "697491ea281ce290af406e9e",  
  "697491eb281ce290af40eb11",  
  "697491ea281ce290af404127",  
  "697491ea281ce290af4054a8",  
  "697491ea281ce290af40881d",  
  "697491ea281ce290af408a10",  
  "697491eb281ce290af40ce90",  
  "697491ea281ce290af4040bf",  
  "697491ea281ce290af404583",  
  "697491eb281ce290af40b103"  
]
```

5.2.2. Love Points

This analytic operates on the graph structure connecting State, City, and User nodes.

Its primary goal is to measure the application "success" of the cities in a specific state. It calculates a custom "Love Points" score based on internal interactions (users interacting with others in the same city), assigning higher weight to mutual matches compared to simple likes.

This allows the administrator to identify the most active dating communities.

The query does:

1. Match: Locates the specific State node and traverses the graph to find all related Cities and the Users living in them.
2. Optional Match: Identifies internal LIKES relationships where both the user and the target reside in the same city
3. Optional Match: Identifies internal MATCHED couples within the same city, using “u._id < partner._id” to ensure that each couple is counted only once
4. Aggregation: Groups by City to sum the total distinct users, likes, and matches
5. Scoring: Applies a weighted formula to calculate the total "Interactions" score: 1 point for every Like and 3 points for every Match
6. Project: Calculates the "Love Ratio" (Total Interactions / number of Users) to normalize the score based on the city's population
7. Return: Outputs a JSON object containing the city name and statistics, sorted by the highest Love Ratio

```

@org.springframework.data.neo4j.repository.query.Query( 1 usage
    "MATCH (s:State {name: $stateName})<-[[:LOCATED_IN]]-(c:City)<-[[:LIVES_IN]]-(u:User) " +

        "OPTIONAL MATCH (u)-[[:LIKES]]->(target:User)-[:LIVES_IN]->(c) " +
        "WITH c, u, count(l) as userLikes " +

        "OPTIONAL MATCH (u)-[[:MATCHED]]-(partner:User)-[:LIVES_IN]->(c) " +
        "WHERE u._id < partner._id " +
        "WITH c, u, userLikes, count(m) as userMatches " +

        "WITH c.name AS city, " +
        "count(DISTINCT u) AS users, " +
        "sum(userLikes) AS totalLikes, " +
        "sum(userMatches) AS totalMatches " +

        "WITH city, users, (totalLikes * 1) + (totalMatches * 3) AS interactions " +

        "WITH city, users, interactions, " +
        "CASE WHEN users > 0 " +
        "THEN toFloat(interactions) / users ELSE 0.0 END AS loveRatio " +
        "RETURN { city: city, users: users, interactions: interactions, loveRatio: loveRatio } " +
        "ORDER BY loveRatio DESC")
List<java.util.Map<String, Object>> getLovePointsAnalytic(@Param("stateName") String stateName);

```

The image below displays a sample result set using the state of California, where each record represents a city with its calculated Love Ratio and aggregate interactions statistics:

```

[
  {
    "loveRatio": 8.090909090909092,
    "city": "fairfax",
    "users": 121,
    "interactions": 979
  },
  {
    "loveRatio": 7.849162011173185,
    "city": "sausalito",
    "users": 179,
    "interactions": 1405
  },
  {
    "loveRatio": 7.802721088435374,
    "city": "san anselmo",
    "users": 147,
    "interactions": 1147
  },
  {
    "loveRatio": 7.724279835390947,
    "city": "belmont",
    "users": 243,
    "interactions": 1877
  },
]

```

6. Indexes

6.1. MongoDB Indexes

To ensure fast response times for the analytics of the system we identified bottlenecks caused by full collection scans and decided to implement the following indexes on collection users and reviews to optimize the performance of the queries:

Index on rating and city

We created the following index:

```
db.users.createIndex({ "reviews_made.rating": 1, "city": 1 })
```

This index drastically improves the performance of the `findTopUnhappyCities` query. The query filters users who have written negative reviews and groups them by city. Without the index, MongoDB was forced to perform a full collection scan, examining all 60,000 documents. The index allows it to jump directly to the entries with low ratings and already provides the city field needed for the group aggregation.

	Performance without Index:	Performance with Index
Execution time in ms:	402	21
Total keys examined:	0	3300
Total docs examined:	59947	2973

Index on status and age

We defined the index:

```
db.users.createIndex({ "status": 1, "age": 1 })
```

This index was designed to optimize the `findSinglesByAgeGroup` query. We included both the filtering field (status) and the calculation field (age) in the index. By explicitly excluding the `_id` field in the projection of the aggregation pipeline, we allowed MongoDB to satisfy the query by reading data exclusively from the index, without ever accessing the documents.

	Performance without Index:	Performance with Index
Execution time in ms:	86	52
Total keys examined:	0	59936
Total docs examined:	59947	0

Index on orientation and age

Similarly to the previous case, we created the index:

```
db.users.createIndex({ "orientation": 1, "age": 1 })
```

This index supports the `findOrientationDemographics` query. Also in this case, the index entirely covers the fields required for filtering (orientation) and demographic grouping (age). Since the query does not require other fields from the original document, the operation occurs by requesting exclusively the index fields.

	Performance without Index:	Performance with Index
Execution time in ms:	72	52
Total keys examined:	0	59946
Total docs examined:	59947	0

Indexes on the "reviews" collection

Unlike the user's collection, we did not implement indexes for the findGlowUpRaw query. This aggregation starts directly with a \$group stage containing complex conditional logic to evaluate past and recent ratings for every document. Since there is no initial \$match stage to filter the data, MongoDB is forced to perform a full collection scan (COLLSCAN) regardless. Creating an index here would not improve read execution times.

6.2. Neo4j Indexes

In our graph database, we implemented a single index on the user _id:

```
CREATE INDEX user_id_index FOR (u:User) ON (u._id)
```

This choice is driven by the fact that the user “_id” acts as the entry point for our main queries, specifically for findRecommendations analytics.

By providing direct access to the user starting node, this index reduces execution times, allowing the engine to start from the exact node rather than scanning all User nodes.

	Performance without Index:	Performance with Index
Total database accesses:	672141	461442
Total allocated memory in bytes:	1.208.264	1.137.984
Total time in ms:	164	73

We did not implement additional indexes, such as those on location like city or state. Even though these are fields used in our queries, especially in getLovePointsAnalytics, after evaluating their impact, we noticed that these queries are processed much more efficiently by leveraging the nature of the graph itself (traversing the LIVES_IN and LOCATED_IN relationships).

7. AI Tools Usage

7.1. Purpose

We used Large Language Models, such as Gemini and ChatGPT, to speed up the development of the LoveMining project. The main reasons for using these tools were:

- Data Creation: To extract and create the main list of User Interests. Furthermore, we used the tools to generate Reviews and Interactions.
- Code Generation: To write standard code and configuration files faster.
- Documentation: To improve the grammar and clarity of this report.

7.2. How

We used specific prompts to help us with the following tasks:

- Python Scripts: We asked the AI to write Python scripts to generate particular type of data for the database and to import this data easily into MongoDB.
- Configuration: We used the AI to generate the base code for the MongoConfig file, specifically the role-based authorization part.
- Java Utilities: We requested the code for the InterestExtractionService, a utility class used to extract user interests from text.
- Code Help: We used the tool as a "pair programmer" to find errors in our Java Spring Boot code and to understand how to fix them.
- Documentation: We pasted our draft text into the chat and asked the AI to correct English mistakes and make the sentences simpler.

7.3. Critical evaluation

Even though the AI was very helpful, we did not simply copy and paste the results. We applied critical thinking and modified the code manually:

- Human-Driven Design: The logic behind the dataset generation, the database structure, and the complex queries was entirely designed by the team. The AI only acted as a "helper" to write the code based on our strict instructions.
- Rejection of AI Advice: The AI often suggested incorrect or too generic solutions for our specific architecture. We ignored these wrong suggestions and implemented our own logic to ensure the system worked correctly.
- Manual Fixes: The Python scripts initially created random IDs. We had to modify the code manually to ensure that MongoDB and Neo4j used the same User IDs, which is necessary for our system.
- Configuration Tuning: The AI suggested standard MongoDB settings. We manually changed the Write Concern to w: 1 and j: false to match our specific "Availability-First" (AP) decision.