UNIVERSITÀ DEGLI STUDI DI PISA

Msc. in Artificial Intelligence and Data Engineering

Large Scale and Multi Structured Databases Project

WeMoveTogether

Student:

Lorenzo Tonelli

Academic Year 2022/2023

Contents

1 – Introduction

WeMoveTogether is a social network designed for both athletes and non-athletes who want to prioritize outdoor sports with new people and in new places. The athletic performances of each user are solely for tracking their physical activities. The social platform aims to encourage registered users, referred to as **movers**, to engage in the most common outdoor sports: running, cycling, and trekking.

The application is designed to handle large amounts of data that may have a non-rigid representation and to establish social relationships among users. For this reason, it was necessary to move away from the rigid paradigms of SQL databases and adopt new paradigms more suitable for this context. For this type of application, two different types of NoSQL databases were utilized, namely GraphDB and DocumentDB.

The project is available on GitHub at the following site:

https://github.com/lorytony/WeMoveTogether-LSMD

2- Dataset

Datasets containing detailed records of users engaging in specific sports activities along defined paths were not readily available. Addressing this challenge required a separate search for both path data and users' sports activities.

In the initial phase of dataset creation, genuine paths where users engage in sports activities were sought. One source was the municipality of Bologna which provided JSON datasets of cycling paths and trails for the province of Bologna. For this application, having paths with coordinates was crucial for visualization within the application. Another source of paths was Kaggle, from another dataset that listed sports routes in America.

Sports activities were identified separately from the paths. The objective was to obtain physiological data from users collected through phones or fitness trackers.

User data was randomly generated through a script and using some online tools.

Comments/reviews for the paths were generated by ChatGPT, tasked with creating credible and generic reviews for trails and natural paths, each associated with a rating from 1 to 5 indicating user appreciation for the review

The sources:
1) ***https://opendata.comune.bologna.it/***
2) ***https://dati.toscana.it/dataset/ce575c4a-b673-427a-94ce-ee560f99683a***
3) ***https://www.kaggle.com/datasets/fitness***
4) ***https://www.kaggle.com/datasets/arashnic/fitbit***
5) ***https://www.kaggle.com/datasets/purpleyupi/strava-data***
6) ***https://www.kaggle.com/datasets/kentvejrupmadsen/garmin-dataset-general***

Infine il Dataset ottenuto ha le seguenti dimensioni:
Users: 12.1 MB
Activities: 221.6 MB
Paths: 1.1 GB
Reviews: 15.1 MB

2.1 Data Cleaning

Final dataset characteristcs

3- Design

3.1 Main Actor

The main actors of the application are:
unregistered user: User that access to the application without apply login. They have to sign up to become a registered user.
registered user (*mover*): User that is registered to the application. They are considered as *mover* by the application after the login.
administrator: They are special users with unique privileges. Their role is to ensure and enforce the rules of the community by deleting offensive reviews or removing users.

3.2 Application Requirements

3.2.1 Functional requirements

Vediamo quali features deve offrire agli utenti categorizzati per il ruolo:

For hosts, who are unregistered individuals, the following features have been defined:
- **Sign in**, which is the ability to take on another role by registering
- **view stat user**, where the user can see some of their information
- **Find user**, allowing the host to browse users within the community
- **Find path**, enabling the host to see which paths have been shared by the community

For registered users (movers), we aim to provide the following functionalities:
- **login/logout**, After the sign up, a user can login *WeMoveTogether* using their credentials to start using the application. After logging in, they can always log out from the application.
- **add activity**: Users can upload a recently practiced activity.
- **modify activity**, Users have the ability to modify certain attributes of an activity
- **delete activity**, Users can delete their own activities.
- **add path**, Users can share a path with the entire community.
- **search path**, Users can search paths by name, distance, and indicative duration
- **search activities**, the user can search activities by name, date, type of activity
- **search users**, the user can search other user of the community by name, surname, nickname, location
- **browse trend paths**: Movers can explore the currently popular paths
- **browse most appreciated paths**: Movers can discover the most appreciated paths
- **browse most appreciated discovers**: Movers can explore the most appreciated discoveries
- **make friend**:Users can become friends with other movers.
- **comment a path**: Users who have practiced a path at least once can leave a comment.
- **browse friends:** Users can see their mover friends.

- **browse mutual friends on a path:** The user can see mutual friends with a specific person
- **browe suggested friends**: The user can see which friends are suggested to them by the application based on certain criteria
- **browse suggested paths**: The user can see which paths are suggested to them by the application based on certain criteria

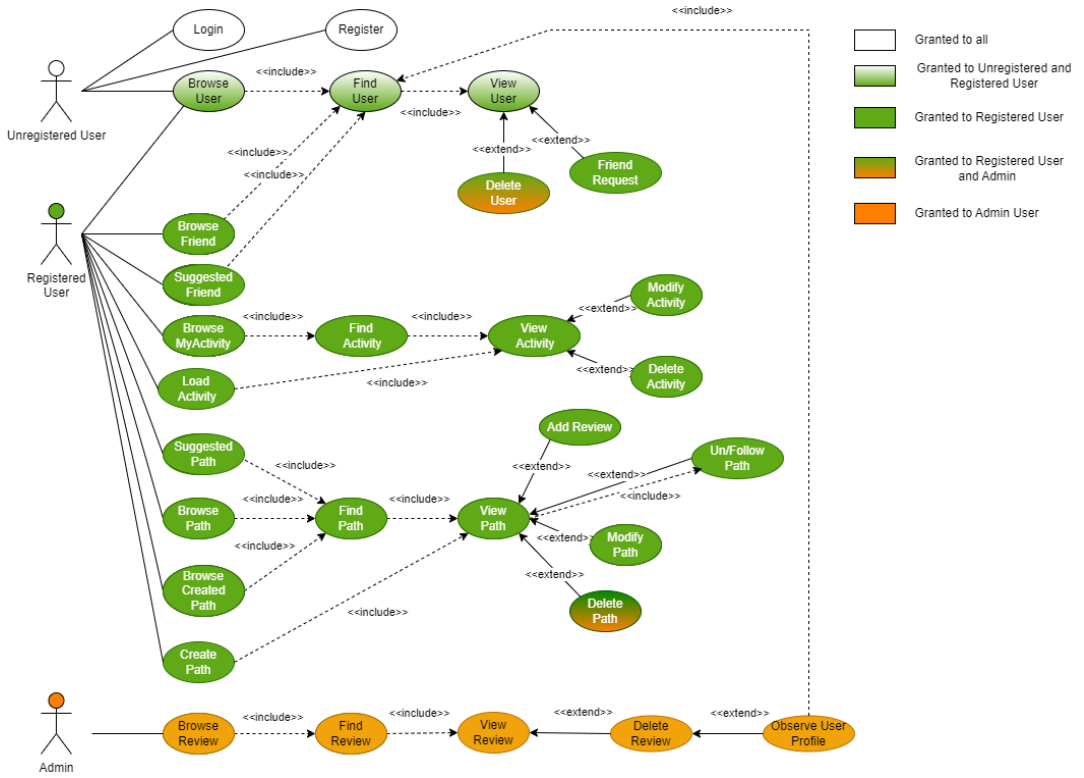
Features offered to the administrator user:
delete users, An administrator can delete a user possibly disrespecting other users
delete paths, An administrator can remove a irrealistic o ridondant paths.

3.2.2 Non-functional requirements

The non-functional requirements of *WeMoveTogether* are:

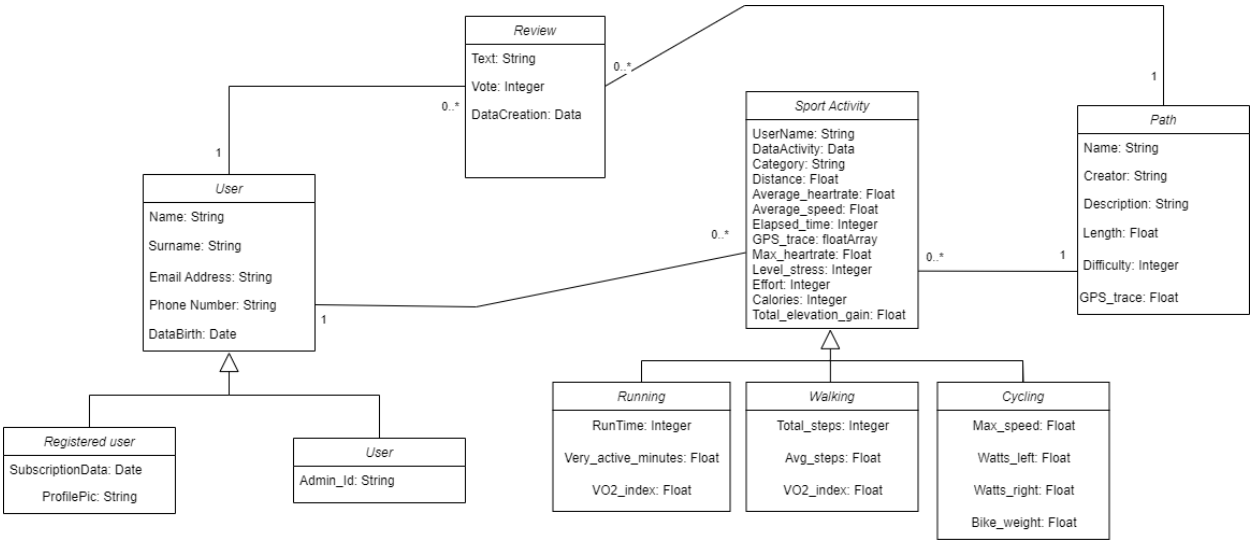
Usability: The application must be user-friendly and must ensure a low responsetime.
Availability: The service must be always available for users in order to guarantee the best possible user experience, providing a response to any query.
Low Latency: The response to the requests should be fast.
Partition Protection:The service must be tolerant to partitioning caused by failures, at the cost of being only eventually consistent.
Reliability: The system must be stable, must return reproducible results and handle exceptions if needed. flexibility The user that makes a course available may choose only the fields they find appropriate to describe their course. The data should be handled in a flexible way.
Portability: The system can be ported to different operating systems with no visible difference in its behavior.
Maintainability: The code should be maintainable and readable

3.3 Use Case Diagram

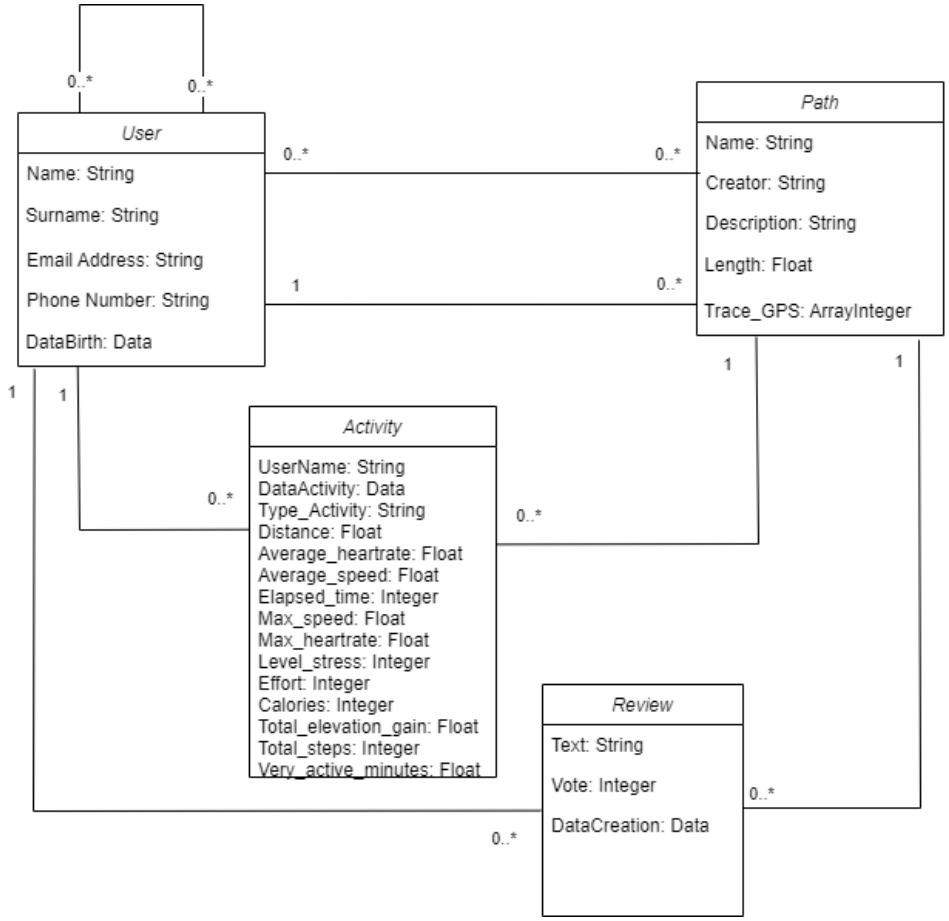


By using three colors, it is indicated which actor can perform a given use case. It is important to note that there are use cases that can be performed by two actors; in this case, it is indicated by two colors. For example, the use case 'Delete User' can be performed either by the user who wants to delete their own account or by the admin who wants to remove a user toxic to the WeMoveTogether community

3.4 UML class Diagram



This is the UML diagram that describes the entities and relationships depicting the WeMoveTogether application scenario. This diagram is not influenced by the databases we will consider for managing the application data. It is possible to make some modifications to this diagram in order to reduce its complexity and enhance interpretability.



In this revised UML diagram, specialized users become a single entity by introducing the attribute 'type_activity,' and sports activities become a single entity by introducing the attribute 'type_activity' and making some attributes optionalDesign Of mongoDB

There are four main entities: User, Review, Activity, and Path. We have User, which is a generalization of the user types Registered User and Administrator, and Activity as a generalization of the activities bike, walk, and trekking

….

3.5 Design of the classes
In questa sezione vogliamo dare una descrizione agli attributi delle entità che alcune sono opzionali.

3.5.1 Attribute della entità User

3.5.2 Attribute della entità Review

3.5.3 Attributi della entità Activity

3.5.4 Attributi della entità Path

4- Document DB

4-1 Implementation

MongoDB has been chosen as the DocumentDB because WeMoveTogether needs to manage large amounts of data in a flexible and efficient manner. The fact that some attributes are optional in the entity is an indication that the document database is suitable to meet the requirement for flexibility.

Let's examine how the entities Activities, Paths, Reviews, and Users have been designed in the DocumentDB. We'll start with the relationships between Reviews and Paths.

Reviews are used only to leave a comment on the path by a user who has practiced it and to provide a score. To efficiently meet these requirements, it was decided to integrate document reviews into paths so that by interacting only with the path collection, one can obtain the latest reviews for the specific path and its average rating. However, reviews increase over time, and there may be documents associated with some paths that exceed the default size limit in MongoDB, which is 16MB. Since it is not deemed necessary to have a complete history of all comments for a path, a decision was made to set a maximum number of reviews for each path. In the event of reaching this limit, the oldest review is deleted.

Let's take a look at an example of a Path Document:


```
{
  "_id": {
    "$oid": "655d35135403deaf4d8ff19e"
  },
  "name": "Via Francigena - 05 Liguria",
  "distance": 64,
  "data_enrolled": "1973-06-16",
  "country": "Italy",
  "city": null,
  "location": "Liguria",
  "website":"https://www.viefrancigene.org",
  "note_it":"relazione contenente highway regionali, per favore non
modificare nulla",
  "geo_shape":{
    "type":"Feature",
    "geometry":{
        "coordinates":[(too many elements to preview)],
        "type":"LineString"
                },
    "properties":{}
            },
```

```
  "comments": [
    {
      "title": "Avventura nel Cuore della Natura",
      "content": "La riserva naturale è un'oasi di pace lontano dalla
città.",
      "score": 4
    },

    {
      "title": "Oasi di Tranquillità",
      "content": "Il sentiero del vulcano è affascinante e educativo.",
      "score": 5
    },
……    …..    ….    ….. ….    ….    ..    ..    .    ..    .

    {
      "title": "Da Non Perdere",
      "content": "Il bosco autunnale trasuda calore con le foglie colorate.",
      "score": 3
    }
  ],
  "num_comments": 22,
  "avg_score": 4.2,
  "id_user": {
    "$oid": "655d33fd5403deaf4d8ff16d"
  },
  "name_creator": "melissacalderon",
  "id_creator": {
    "$oid": "655f2982cb563fca79f8424c"
  }
}
```


To soddisfy the query of obtaining the most practiced activity type by a user, one option was to integrate activities for each User Document. However, there is always the risk that the number of activities becomes high for some very sporty users, thereby increasing the maximum document size. It would not be acceptable to impose a limit on the number of activities per user to be recorded in the document because we want all activities to be saved with all the information. Therefore, a redundancy of User Document attributes in their respective activities was decided upon. The `Id_user` in the activities collection corresponds to a linking to the document in the users' collection. This way, the query to calculate the most practiced activity by a user simply involves using the activities collection.

It should be emphasized that this document linking, although the mechanism is similar to the Foreign Key (FK) in an SQL database, has a different usage. It will never be used to perform joins between documents in the user collection and those in the activity collection.

Implementing some attributes of the users' collection in the activity collection also allows us to easily soddisfy the following query: calculate the calories consumed by the user in the last period. This query returns the sum of calories consumed in the activities practiced in the last period. This query can be satisfied by working solely in the activities collection.

Another query that the application satisfies is as follows: Obtain the user's preferred path, preferred in the sense that they have engaged in more activities in the last period. It was deemed more practical to create redundancy for certain path attributes (including the key) in the activities collection, similar to what was done between users and activities. In this case, the query is fulfilled by acting solely within the activities collection.

Calculate the most appreciated explorers by the community. This query is directly resolved by intervening in the paths collection, which contains embedded documents of reviews with associated scores.

Calculate the trendiest paths in the community in the last period by users, i.e., retrieve 'k' paths that have the highest average score from users and have had a minimum number of reviews. It was deemed necessary to distribute the unique id_user attribute in the paths collection to group paths for different users.

4.2- DocumentDB query

Let's look at the queries that have been implemented in the DocumentDB on MongoDB.

getLastActivities: We want all the activities of a user performed in the last period of time. The query is fulfilled by working on the Activities collection, identifying the activities of the user of interest, and sorting their activities in descending order of time.

```
//------------------------------------------------------------------------
MongoCollection<Document> collection = database.getCollection("Activities");
MongoCursor<Document> cursor = collection.aggregate(Arrays.asList(
        new Document("$match",new Document("id_user",new ObjectId(idUser))),
        new Document("$sort", new Document("date", -1)),
        new Document("$limit", limit),
        new Document("$skip",skip)
)).iterator();
//------------------------------------------------------------------------
```

getLastComment: We want the latest 'k' comments that have been left by users for a given path. Since the reviews have been integrated into the paths, it is necessary to identify the path of interest, extract the relevant paths using the appropriate command in MongoDB, and sort them in descending order of time.

```
//------------------------------------------------------------------------
MongoCollection<Document> collection = database.getCollection("Paths");
MongoCursor<Document> cursor = collection.aggregate(Arrays.asList(
        new Document("$match", new Document("_id", new ObjectId(pathId))),
        new Document("$project", new Document("comments", 1)),
        new Document("$unwind", "$comments"),
        new Document("$project",new Document("name","$comments.name").
```

```
        append("surname","$comments.surname").
                append("country","$comments.country")
                .append("comment_timestamp","$comments.comment_timestamp")
                .append("content","$comments.content")
                .append("title","$comments.title")
                .append("score","$comments.score")),
        new Document("$sort", new Document("comment_timestamp",-1))
)).iterator();
//-------------------------------------------------------------------------
```

getBestRatingPath: We want to display the top 'k' paths that have been most appreciated by the community of movers. Paths with an average score greater than 3 are considered as most appreciated. Additionally, only paths with a minimum number of reviews, for example, 100, are selected. The query exclusively operates on the Paths collection and will return the top 'k' paths with a sufficient number of comments, sorted in descending order of the average score.

```
//-------------------------------------------------------------------------
MongoCollection<Document> collection = database.getCollection("Paths");
MongoCursor<Document> cursor = collection.aggregate(Arrays.asList(
        new Document("$match", new Document("num_comments",
        new Document("$gte", threshold))),
        new Document("$sort", new Document("avg_score", -1)),
        new Document("$limit", limit)
    )).iterator();
//-------------------------------------------------------------------------
```

getTrendingPath: We want the 'k' paths that have been most practiced by the community in the last period. Initially, we identify the activities from the last period, group them by path, counting the total number of activities for each path. Finally, we return the top 'k' paths that have been most practiced by users in the last period

```
//-------------------------------------------------------------------------

MongoCollection<Document> collection = database.getCollection("Activities");

MongoCursor<Document> cursor = collection.aggregate(Arrays.asList(
        new Document("$match", new Document("date",
                                new Document("$gte",startDay))),
        new Document("$group",
                new Document("_id",new Document("id_path","$id_path"))
                        .append("count_votes", new Document("$sum",1))),
        new Document("$sort", new Document("count_votes",-1)),
        new Document("$set", new Document("id_path", "$_id.id_path")),
        new Document("$unset","_id")
)).iterator();
//-------------------------------------------------------------------------
```

getMostFrequentActivityTypeByUser: We want to understand which type of sports activity is most practiced by the user. We need to work on the Activities collection, identifying all the user's activities,

and for each type of activity, count the total time spent, ordered in descending order. Finally, select only the top result.

```
//-------------------------------------------------------------------------
MongoCollection<Document> collection = database.getCollection("Activities");
MongoCursor<Document> cursor = collection.aggregate(Arrays.asList(
        new Document("$match", new Document("id_user", new
ObjectId(moverId))),
        new Document("$group",
                new Document("_id",new
Document("type_activity","$type_activity"))
                        .append("count_type", new Document("$sum",1))),
        new Document("$sort", new Document("count_type",-1)),
        new Document("$set", new Document("type_activity",
"$_id.type_activity")),
        new Document("$unset","_id"),
        new Document("$limit",1)
)).iterator();
//-------------------------------------------------------------------------
```

getNumberKcalsBurnedLastPeriod: The goal is to calculate the total quantity of calories burned by the user in activities over the last period.

getMostPracticedPathByUser: We want to understand which path is the most practiced by the user. This is achieved by working on the Activities collection, grouping by id_path, and counting the activities associated with each path.

getMostAppreciatedDiscovers: We want to identify 'k' users who have discovered the most appreciated paths by the community.

```
//-------------------------------------------------------------------------
MongoCollection<Document> collection = database.getCollection("Paths");
MongoCursor<Document> cursor = collection.aggregate(Arrays.asList(
        new Document("$match", new Document("num_comments",
                                new Document("$gte", threshold))),
        new Document("$group", new Document("_id",
                                new Document("id_creator", "$id_creator"))
                .append("avg_score", new Document("$avg", "$avg_score"))),
        new Document("$sort", new Document("avg_score", -1)),
        new Document("$set", new Document("id_creator", "$_id.id_creator")),
        new Document("$unset","_id"),
        new Document("$limit", limit)
)).iterator();
//-------------------------------------------------------------------------
```

getUserStatPerformance. We want to know, in percentage terms, how a user's best performance for a given path compares to the entire community.

The query has been fulfilled by working solely within the Activities collection. Initially, the best performances of each user for that path are identified, sorted in descending order of performance.

Performance is defined as the shortest time taken to complete the trail. Once the best athletic performances (activities) of each user for that path are obtained, the total count is determined, and the position of the user of interest is identified relative to the entire community. Finally, with the following formula:

```
perc = (float) (numDocuments - userPosition) / numDocuments;
```

si calcola la prestazione in termini percentuali dell'utente rispetto a tutta la community.

```
//------------------------------------------------------------------------------
MongoCollection<Document> collection = database.getCollection("Activities");
MongoCursor<Document> cursor = collection.aggregate(Arrays.asList(
        new Document("$match", new Document("id_path", new
ObjectId(idPath))),
        new Document("$group", new Document("_id",new
Document("id_user","$id_user"))
                        .append("bestTime", new
Document("$min","time_elapsed"))),
        new Document("$sort", new Document("time_elapsed",1)),
        new Document("$set", new Document("id_user", "$_id.id_user")),
        new Document("$unset","_id")
)).iterator();
//------------------------------------------------------------------------------
```

4.3 Indexes

To improve the performances of the application we performed index analysis, in this subsection we will discuss the introduction of indexes for both Neo4J and MongoDB.

Indexes introduction on MongoDB

Query	Index	ms	keys	docs
getLastUserActivities	-	343	0	101.000
	date	220	40.326	40.326
Find activity by parameters	-	156	0	101.000
	name	212	101.000	702

Collection activity contains the followind indexes:

{Id_activity }
{id_user, timestamp }

Collection Users:
{id_user}

Collection Paths:
{id_path}
{id_creator}

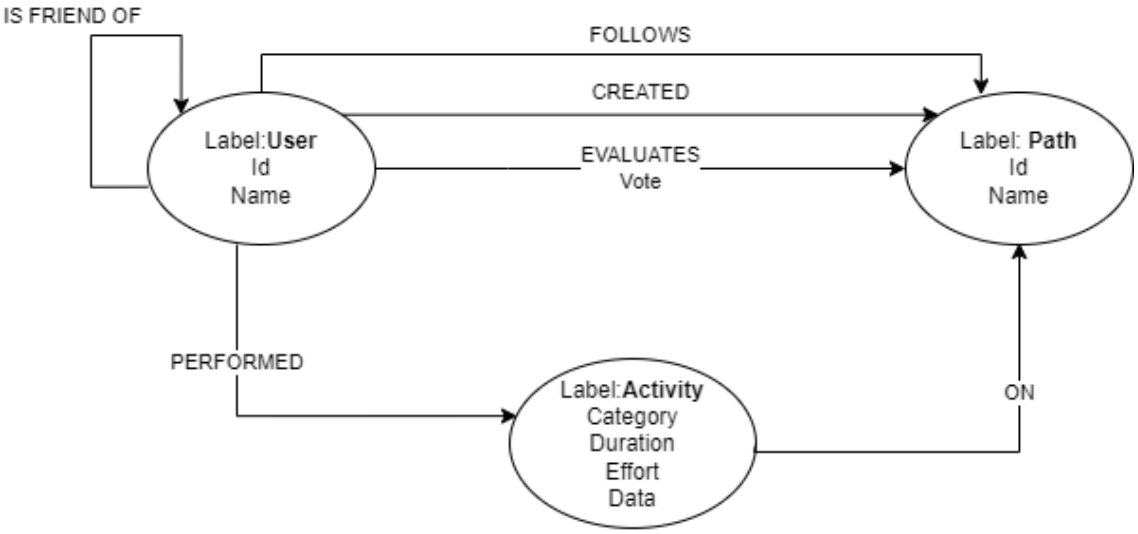
Indexes introduction on GraphDB

Query	Index	ms	hits
findSuggestedPaths	-	680	101.000
	name	509	12.932
Find suggested Friend	-	883	45.321
	name	331	7435
IsFriend	-	303	19202
	username	100	294

5 Design of GraphDB

5-1 Implementation

The graph database is used to perform fast networking, suggestion queries that need to scan several relationships between users or relationships between users,activities and paths and avoid multiple joins that may be too computationally intensive and would not guarantee the low latency requirement.



All the IDs mentioned correspond to the MongoDB-generated object IDs since the Graph dataset is a replica of the MongoDB dataset. The graph primarily consists of 2 nodes and 5 edges.

The first node is User, representing a user in the graph. Its attributes include id_user (corresponding to the MongoDB-generated object ID in the User documents), name, surname, and age.

The second node is Path, which has attributes id_path (derived from the MongoDB-generated Object ID), name, location, and distance.

The GraphDB dataset is a replica of the MongoDB dataset, and only the necessary attributes were selected to avoid queries involving both databases.

Let's explore the relationships present:

Friendship: This relationship indicates a bidirectional edge and signifies the friendship between two users.

Review: This relationship has the attribute 'score,' indicating the user's appreciation of a path.

Discover: This relationship indicates the user who discovered a particular path.

Activities: This relationship indicates the activity performed by the user for a specific path.

5-2 Query GraphDB

Let's take a look at the queries that have been implemented in the GraphDB database deployed in Neo4j.

getFriends. We want friends of a given user. In this application, the concept of friendship is viewed as a mutual follower and following, similar to Facebook.

getMutualFriends. Given two users, we want to know which friends they have in common..

```
MATCH (me:User{id:$idUser})-[:FRIEND]->(u:User)<-[:FRIEND]-
(u2:User{id:$idUser2})
return distinct(u.id) as id,u.name as name
limit $limit;
```

getMutualLikedPaths. We want to know the paths that are liked by two users.

```
MATCH (me:User{id:$loggedUserId})-[ev1:EVALUATION]->(p:Path)<-
[ev2:EVALUATION]-(u2:User{id:$moverId})
where ev1.score >= 1 and ev2.score >= 1
return distinct(p.id) as id,p.name as name
limit $limit;
```

getSuggestedFriends. We want to suggest to the logged-in user 'k' users as friends. Two levels of suggestions are used: a strong one and a regular one.

For the strong suggestion, we aim to recommend friends of the user's friends who have given positive reviews to paths that the user has positively reviewed. Naturally, these users should not already be friends with the user.

The weaker suggestion involves identifying users who have given positive reviews to paths that the user has also positively reviewed, while ensuring that they are not already friends with the user.

```
MATCH (p1:Path)<-[ev1:EVALUATION]-(me:User{id:$idUser})-[:FRIEND*2..2]-
(u:User)-[ev2:EVALUATION]->(p1)
where not exists ((me)-[:FRIEND]->(u)) and ev1.score >=3 and ev2.score >= 3
```

```
return distinct(u.id) as id,u.name as name,u.surname as surname,u.age as age,
u.img_profile as img_profile
UNION
MATCH (me:User{id:$idUser})-[v1:EVALUATION]->(p:Path)<-[v2:EVALUATION]-
(u:User)
where v1.score >= 3 and v2.score >= 3
    and not exists((me)-[:FRIEND]->(u))
return distinct(u.id) as id,u.name as name,u.surname as surname,u.age as age,
u.img_profile as img_profile
limit $limit;
```

getSuggestedPaths, We want to suggest 'k' paths to the user. The suggestion is based on three types: highly suggested, strongly suggested, and suggested.

In the first case, we aim to suggest paths to the user that they haven't practiced, created by friends, and have received a positive average review from the community. In the second case, we suggest paths not practiced by the user but appreciated by friends. Finally, we suggest paths that are highly appreciated by the community.

```
MATCH (p:Path)<-[ev:EVALUATION]-(u:User)
with avg(ev.score) as AverageScore,p
where AverageScore >= 3.0 and
not exists((:User{id:$idUser})-[:PERFORMED]->(:Activity)-[:on]->(p)) and
not exists((:User{id:$idUser})-[:EVALUATION]->(p))  and
exists((:User{id:$idUser})-[:FRIEND]->(:User)-[:CREATED]->(:Path))
RETURN distinct(p.id) as IdPath
       UNION
MATCH (me:User{id:$idUser})-[:FRIEND]->(f:User)-[ev:EVALUATION]->(p:Path)
where ev.score >= 3.0 and
not exists((me)-[:PERFORMED]->(:Activity)-[:on]->(p)) and
not exists((me)-[:EVALUATION]->(p))
RETURN distinct(p.id) as IdPath
       UNION
MATCH (p:Path)<-[ev:EVALUATION]-(u:User)
with avg(ev.score) as AverageScore,p
WHERE  not exists((:User{id:$idUser})-[:EVALUATION]->(p)) and
not exists(((:User{id:$idUser})-[:PERFORMED]->(:Activity)-[:on]->(p)))
and AverageScore >= 3.0
RETURN distinct(p.id) as IdPath
LIMIT $limit;
```

6 Distributed Database Design

According to the non-functional requirements of the LearnIt! application, we should guarantee high availability, low-latency and partition protection.

We oriented our application to the AP edge of the CAP triangle ensuring the eventual consistency.

In order to ensure that the partition protection and availability requirements were satisfied, we've designed a distributed system. Thanks to the three different replicas we decided to use, the system

guarantees avoiding the single point of failure and guarantee also the low latency requirement thanks to the balancing of the load (different requests made by clients are handled by different servers in order to guarantee a fast response) and thanks to the fact that after a write operation is received only one replica's data will be immediately updated while the data on the other replicas will be updated in a second moment in order to don't keep the server busy for too much time during a write operation.

6-1 Replicas

Three replicas are present in our system for the document database, one for each

machine of the cluster provided by the University of Pisa. For graph database

we have only one replica, but theoretically there should have been three of them

(it's a premium feature). Summing up everything, the system has:

- 3 replicas for the document database
- 1 instance for the graph database
- only one replica out of three must be updated in order to commit a write operation
- we choose nearest read preference

Write concern

For what concerns the write operations, we choose that they can be considered completed as soon as the primary replica set member has successfully completed the write. Our decision was led by the need to ensure the lowlatency requirement.

Read preference

We decided to set the read preference as nearest because we want to guarantee as fast as possible responses to the clients' requests. Since we can consider LearnIt! as a read-heavy application, it is fundamental to quickly respond to the client, this is why we choose to set the read preference to nearest replica.

6-2 Sharding Idea

In a plausible scenario where we have servers per country, such as Italy, Germany, and France, and in line with the context in which the WeMoveTogether application operates, we can make an intuitively valid assumption. We can suppose that users from a certain country are more likely to engage in paths or trails within their own country of residence.

Furthermore, the populations of these states are of the same order of magnitude (around 65 million people). So, if a percentage of these users were to use the application (0.005%), there would be a similar order of magnitude of clients per country.

In this scenario, we can use the geographical location as the sharding key for all our collections. Locations belonging to Germany, for example, would be directed to a specific server named 'Germany,' and so on.

However, in the specific case of the dataset, it wouldn't be feasible to use this sharding key because:

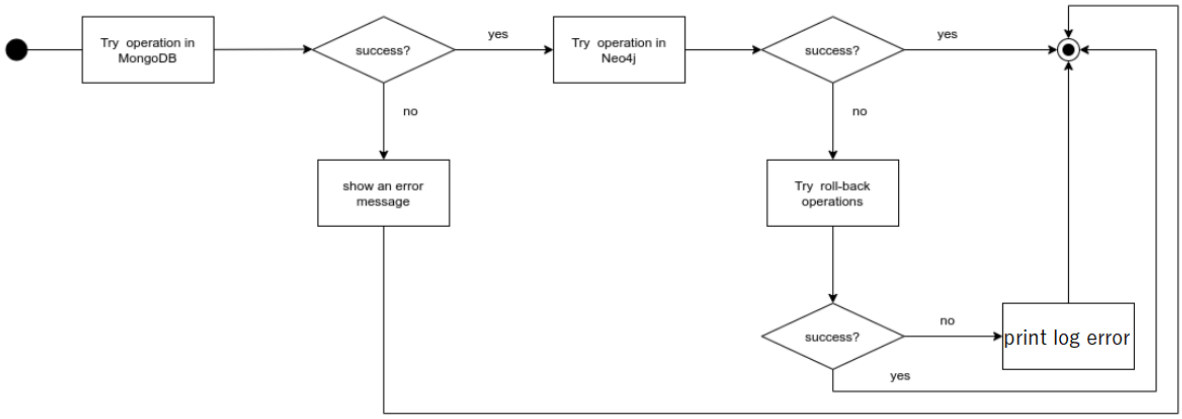
- Users are randomly generated and, therefore, can belong to any part of the world.
- The discovered paths are predominantly located in Italy.

There are no attributes that can somehow balance the read/write workload of the Users collection. Therefore, the idUser is chosen as the sharding key, using it as input in a hashing function for partitioning the collection.

The same sharding key strategy would be applied to the activities performed by users. As for the paths, a similar approach is taken.

6-3 Handling inter-database consistency

To prevent possible inter-database inconsistencies we had to perform some queries on both the database and to ensure that those queries are successfully completed only if the operation on both the databases ends with a success and if one of the two operations fails, we should guarantee a consistent state.



Let's explore how to handle CRUD operations in the application.

Delete User

1) Save temporary the user document to be deleted.
2) Delete the user in MongoDB.
3) Save user-related activities.
4) Delete activities.
5) Delete the user in Neo4j using the objectID from step 1.
6) Delete activities in Neo4j using the objectIDs from step 3.

A potential issue is whether to delete paths discovered by the user and used by the community. In this case, the paths wouldn't be deleted because they have become public domain. Instead, information related to the user who discovered the path should be updated to indicate that the user has been deleted.

Delete Activity

1) Save the user's activity.
2) Delete the activity in MongoDB.
3) Delete the activity in Neo4j.

Delete Path

1) Save temporary the path.
2) Delete the path in MongoDB.
3) Delete the path in Neo4j.

Activities related to the deleted path should not be deleted because user activities should always be visible. Instead, the 'Info path' button will be rendered unclickable with the label 'Deleted'. It would be a good idea to create a 'PATH DELETED' document with a specific objectID for use in MongoDB and Neo4j."