|  |
| --- |
| UNIVERSITÀ DEGLI STUDI DI PISA |
| Immagine che contiene schizzo, clipart, Line art, illustrazione  Descrizione generata automaticamente |
| 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 4](#_Toc151651573)

[2 - Dataset 4](#_Toc151651574)

[3 Design 5](#_Toc151651575)

# 1 - Introduction

***WeMoveTogether***, è un social-network pensato per sportivi e non, che vogliono dare più importanza a praticare sport all’aperto con persone nuove e posti nuovi. Le prestazioni sportive di ogni utente servono solo per tenere traccia della propria attività fisica. Il social vuole incentivare gli utenti iscritti, chiamati ***mover***, a praticare i tipi di sport più comuni all’aperto: corsa, ciclismo e trekking.  
  
L'applicazione è stata pensata per gestire grandi quantità di dati che possono avere una rappresentazione non rigida e per mettere in relazione di tipo social fra gli utenti. Per tale motivo è stato necessario uscire dai parigmi rigidi dei database SQL e adottare nuovi paradigmi più adatti per questo contesto. Per questo tipo di applicazione è stato necessario usare due differenti tipi di NoSQL database, chiamati GraphDB ed DocumentDB.

È possibile consultare il progetto su GitHub al seguente sito: [TO DO – LINK]

# 2 - Dataset

Non sono stati trovati dataset che riportassero le attività sportive di utenti svolte in percorsi dettagliati. È stato necessario affrontare il problema ricercando separatamente la ricerca di percorsi e attività sportive degli utenti.  
Una prima fase per creare il dataset è stato ricercare dei veri percorsi dove utenti praticano attività sportiva in generale. Una prima fonte è stato il comune di bologna che ha messo a disposizione dataset in formato json dei percorsi ciclabili e sentieri per la provincia di bologna. Per questa applicazione era importante avere percorsi che avessere delle coordinate del percorso in modo da essere visualizzato dalla applicazione. Una seconda fonte dei percorsi è stata kaggle da questo dataset [TO DO – AGGIUNGERE IL LINK] che riporta percorsi sportivi in america.

La attività sportive sono state individuate separatamente dai percorsi. L’obiettivo era ottenere dati fisiologici di utenti raccolti da telefono o braccialetti.

Gli utenti sono stati generati casualmente attraverso uno script e usando alcuni tool online.  
I commenti/recensioni dei percorsi sono stati generati da ChatGPT chiedendo di generare recensioni generiche credibili a sentieri e percorsi naturali con associato un voto da 1 a 5 correlato al apprezzamento della recensione da parte dell’utente.

Infine il Dataset ottenuto ha le seguenti dimensioni: [TO DO – inserire dimensioni]  
Users: 12.1 MB  
Activities: 122.6 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.

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

Immagine che contiene testo, diagramma, schermata

Descrizione generata automaticamente

Mediante l’utilizzo di 3 colori viene indicato quale sia l’attore che possa effettuare quel use case. Bisogna osservare che ci sono use case che possono essere effettuati da due attori, in questo caso viene indicato da due colori. Ad esempio, il use case “Delete User” può essere effettuato dal utente che vuole cancellare il proprio account oppure dal admin che vuole cancellare un utente tossico per la community di WeMoveTogether.

3.4 UML class Diagram  
Immagine che contiene testo, diagramma, Parallelo, Piano

Descrizione generata automaticamente

[TO DO – ATTENZIONE questo diagramma uml non è definitivo, il prof ducange mi ha indicato degli errorini da correggere, alcuni erano: introdurre la specializzazione degli utenti e togliere i nomi agli archi. Io inoltre, per fare scena introdurrei delle specializzazioni in activity partendo da una activity con attributi generici e poi nel dettaglio in base al tipo di attività tipo bike, mountain bike, trekking, corsa ect. . Notare che nella documentazione di LearnIt è stata fatta la specializzazione degli utenti, prendere spunto da lì come si fa]

Questo è il diagramma UML che descrive le entità e le relazioni che dipingono lo scenario della applicazione WeMoveTogether. Questo diagramma non è influenzato dai database che andremo a considerare per gestire i dati della applicazione.

È possibile effettuare qualche modifica a questo diagramma in modo da ridurne la complessità e aumentarne la interpretabilità.

[TO DO – Aggiungere diagramma UML in cui gli utenti specializzati diventano una entità sola, in cui le attività sportive diventano una attività sola, vedi come fanno quelli di LearnIt]

Design Of mongoDB  
Ci sono 4 principali entità: User, Review, Activity and Path. Abbiamo User che è una generalizzazione dei tipi di utenti Registered User ed Administrator e Activity come generalizzazione delle attività bike, walk, 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

Document DB

[MAH, io qua parlerei delle query che mi aspetto di implementare nella applicazione, altrimenti come faccio a motivare le scelte di progetto nelle zezione DocumentDB e GraphDB?] <- [guardando il powerpoint di progetto, a quanto pare, le query sono già separate per DocumentDB e graphDB]  
  
MongoDB è stato scelto come DocumentDB perché WeMoveTogether ha bisogno di gestire grandi quantità di dati in modo flessibile ed efficiente. Il fatto che alcuni attributi siano opzionali nella entità è una indicazione che il document database è adatto per soddisfare il requirement fexibility.

Le review servono solo per lasciare un commento al path da parte di un utente che l’ha praticato e per fornire un punteggio. Per soddisfare efficientemente questi requisiti, si è deciso di integrare i document review nei path in modo che interagendo solo alla collection path, di ottenere le ultime recensioni del path in questione e una sua stima media. Tuttavia, le review, aumentano del tempo. Quindi ci possono essere documenti associati ad alcuni path che possono superare il limite di size predefinito in MongoDB, 16MB. Poiché non è ritenuto necessario avere lo storico di tutti i commenti di un path, è scelto di dare un numero limite di review per ogni path e in caso di raggiungimento del limite, si procede ad eliminare la recensione più vecchia.

[TO DO – Aggiungere Esempio di path con recensioni embedded]  
Per soddisfare la query di ottenere il tipo di attività più praticato da un utente una opzione era quella di integrare le attività per ogni User Document. Tuttavia, c’è sempre il rischio che le attività diventino un numero alto per alcuni utenti molto sportivi andando ad aumentare la dimensione massima per documento. Non sarebbe accettabile mettere un numero limite di attività per ogni utente da registrare nel documento, perché vogliamo tutte le attività vengano salvate con tutte le informazioni. Allora è stato deciso di effettuare una ridondanza, di attributi del document User nelle relative attività. Dove il Id\_user nella collezione delle attività corrisponde a un linking al document nella collezione degli utenti. In questo modo la query di calcolare la attività più praticata da un utente basta usare la collezione delle attività.  
Bisogna sottolineare che questo document linking, sebbene il meccanismo sia uguale alla FK di un database SQL, l’utilizzo è diverso. Non servirà mai per effettuare join tra i documenti della collezione user e quelle della attività.  
  
Implementando alcuni attributi della collezione degli utenti in quella della attività ci permette anche di soddisfare agilmente la seguente query calcolare le kcal consumate dall’utente nell’ultimo periodo, cioè che restituisca la somma delle kcal consumate nelle attività praticate nell’ultimo periodo. Questa query può essere soddisfatta lavorando soltanto nella collezione delle attività.

[TO DO – mostrare esempi di documenti activty e user che implementano questa ridondanza. Magari attraverso un diagramminno uml circorscritta alla relazione entità User ed Activity che metta in risalto questa cosa della ridondanza.]

Un’altra query che l’applicazione soddisfare è la seguente Ottenere il path preferito dall’utente, preferito nel senso che ha effettuato più attività nell’ultimo periodo. È stato ritenuto più pratico fare come è stato fatto tra user e activities, quello di fare una ridondanza di alcuni attributi di path (tra cui la chiave) nella collezione activties. In questo caso la query si soddisfa agendo soltanto nella collezione delle attività.

[TO DO – Mostrare esempio documenti e uml]  
  
Calcolare gli esploratori più apprezzati dalla community. Questa query si risolve direttamente intervendendo nella collezione dei path che contengono in modo embedded i documenti review con il punteggio associato.

Calcolare i path più in voga nella community nell’ultimo periodo da parte degli utenti, cioè riportare k path che hanno avuto un punteggio medio da parte degli utenti più alto e che abbiano avuto un numero di recensioni minimo. È stato ritenuto necessario distribuire l’attributo univoco id\_user nella collezione dei path in modo da raggruppare i path per utenti diversi.

[TO DO – mostrare esempio document uml]

[Mostrare diagramma UML finale influenzato dal DocumentDB]  
  
Desing of Neo4j  
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.

**DocumentDB query**

Vediamo le query che sono state implementate nel database DocumentDB esiguito su MongoDB.

getLastActivities: Si vogliono tutte le attività di un utente effettuate nell’ultimo periodo di tempo.   
La query si soddisfa lavorando sulla collezione Activities individuando le attività dell’utente di interesse e ordinando le sue attività in ordine decresente di tempo.

getLastComment: Si voglio gli ultimi k commenti che sono stati lasciati dagli utenti per un dato path.  
Poiché le recensioni sono state integrati nei path, bisogna individuare il path di interesse, estrarre i path con l’opportuno comando in MongoDB e ordinarli in odine decrescente di tempo.

getBestRatingPath: Si vuole mostrare i *k* percorsi che sono stati più apprezzati dalla community di movers.   
Per path più apprezzati si intendono quei path che hanno score media maggiore di 3. Inoltre si selezionano soltanto quei path che abbiamo avuto un numero minimo di recensioni, ad esempio 100.  
La query quindi lavora soltanto sulla collezione Paths. La query restituirà i k percorsi che hanno un numero sufficiente di commenti e in ordine decrescente di score medio.

getTrendingPath: Si vogliono i *k* percorsi che sono stati più praticati dalla Community nell’ultimo periodo.  
Inizialmente si individano le attività dell’ultimo periodo, si raggruppano per path conteggianod il totale di attività. Infine si restituiscono i *k* percorsi che sono stati più praticati dagli utenti nell’ultimo periodo.

getMostFrequentActivityTypeByUser: vogliamo capire quale tipo di attività sportiva è più praticata dall’utente.   
Bisogna lavorare sulla collezione Activities individuando tutte le attività dell’utente e per tipo di attività conteggiare il tempo complessivo ordinato in ordine descrescente. Infine, selezionare solo la prima risposta.

getNumberKcalsBurnedLastPeriod: Si vuole calcolare la quantità complessiva di kcal bruciate dall’utente nelle attività dell’ultimo periodo.  
  
getMostPracticedPathByUser: vogliamo capire quale percorso è il più praticato dall’utente.  
Si lavora sulla collezione Activities raggruppando per id\_path e conteggiando le attività impiegate in ciascun path.

getMostAppreciatedDiscovers: vogliamo individuare *k* utenti che hanno individuato i path più apprezzati dalla community.

getUserStatPerformance. Vogliamo sapere, in termini percentuali, come la migliore performance di un utente per un dato path si colloca rispetto a tutta la community.  
La query è stata soddisfatta lavorando soltanto nella collezione Activities. Inizialmente vengono individuate le migliori performance di ogni utente per quel path ordinati in ordine decresente di performance. Per performance si intende il minor tempo impiegato per percorrere il sentiero. Una volta ottenute le migliori prestazioni sportive (activity) di ogni utente per quel path si va a contare quanti sono in totale e si individua la posizione dell’utente di interesse rispetto a tutta la community. Infine con la seguente formula:

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

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

getTrendingPath

**Indexes**

[TO DO – riportare tabella]

Collection activity

{Id\_activity }  
{id\_user, timestamp }

Collection Users  
{id\_user}

Collection Paths

{id\_path}  
{id\_creator}

Immagine che contiene testo, diagramma, linea, cerchio

Descrizione generata automaticamente

[TO DO – da modificare il diagramma del grafo! Activity deve diventare un arco diretto tra User e Path. Magari a lezione far notare questa cosa durante la presentazione]

Tutti gli id riportati corrispondono agli object\_id generati da MongoDB poiché il dataset per il Graph corrisponde a una replica del dataset di MongoDB.

Il grafo è costitito principalemnte da 2 nodi e 5 archi.  
Il primo nodo è User, che rappresenta un utente nel grafo. I suoi attributi sono id\_user, che corrisponde al object\_id generato dai documenti in User di MongoDB; name, surname, age.  
Il secondo nodo è Path, che riporta id\_path, anche questo si basa dal Object\_id generato da MongoDB, name, location, distance.

Il dataset del GraphDB è una replica del dataset di MongoDB. Sono stati presi gli attributi necessari in modo da evitare di fare query che coinvolgessero entrambi i database.

Vediamo le relazioni che sono presenti:  
Friendship, questa relazione indica un arco di duplice direzione e indica l’amicizia che c’è tra due utenti.  
Review, questa relazione ha l’attributo score che indica l’apprezzamento di un Path da parte di un utente.  
Discovere, relazione che indica l’utente che ha scoperto quel path.  
Activities, questa relazione indica l’attività svolta dall’utente per un percorso.

**Query GraphDB**  
Vediamo le query che sono state implementate nel database GraphDB implementato in Neo4j.  
  
*getFriends*. Vogliamo amici di un dato user.  
In questa applicazione il concetto di friendship è visto come un follower e following reciproco stile Facebook.

*getMutualFriends*. Dati due utenti vogliamo sapere quali amici ci sono in comune.

getMutualLikedPaths. Vogliamo conoscere i percorsi che piacciono a due utenti.

getSuggestedFriends. Vogliamo suggerire all’utente che ha fatto il login *k* utenti come amici.  
Sono usati due livelli di suggerimento. Uno forte e uno normale.  
Vogliamo suggerire gli amici degli amici dell’utente che hanno messo recensioni positive a percorsi che sono state recensite positivamente dall’utente. Naturalmente questi utenti non devono essere tra gli amici dell’utente.  
Il suggerimento più debole consiste nel individuare utenti che hanno recensito positivamente percorsi che sono stati recensiti positivamente anche dall’utente controllando però che non siano già amici.

getSuggestedPaths, vogliamo suggerire all’utente *k* percorsi.  
Il suggerimento è basato secondo tre tipi di suggerimenti: assolutamente suggerito, fortemente suggerito ed suggerito. Nel primo caso, vogliamo suggerire all’utente path che non ha praticato e creati da amici, che hanno avuto una recensione media positiva dalla community. Nel secondo caso si suggeriscono percorsi non praticati dall’utente che sono stati apprezzati da amici. Infine si suggeriscono i path che sono più apprezzati dalla community.

**Replication Configuration**

3.7 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 (exploiting 3 virtual machines provided by University of Pisa). 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.

3.7.1 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.

**Sharding Idea**

In uno scenario verosimile in cui abbiamo un server per nazione, tipo Italia, Germania e Francia e in accordo al contesto in cui l'applicazione WeMoveToghether si ritrova

Possiamo inoltre fare una ipotesi intuitiva ma valida per lo scenario in cui la nostra applicazione, WeMoveTogether, lavora. Possiamo supporre che gli utenti di una certa nazione è più probabile che svolgano percorsi o sentieri appartenenti alla propria nazione di residenza.

Inoltre, la popolazione di questi stati hanno lo stesso ordine di grandezza(intorno ai 65 milioni di persone), quindi se una percentuale di questi utenti dovesse usare l'applicazione(0.005%) ci sarebbe lo stesso ordine di grandezza di client per nazione.

In questo scenario, possiamo usare come shardingKey la posizione geografica per tutte le nostre collezioni. Le località appartenenti alla germania saranno indirizzati a un server specifico chiamato "Germania", ect.

Tuttavia, nel caso specifico del dataset non sarebbe possibile usare questo sharding key, in quanto:

- gli utenti sono generati casualmente e quindi appartenenti in ogni parte del mondo

- i percorsi trovati sono massicciamente situati in italia

Non ci sono attributi che possono in qualche modo bilanciare i read/write della collezione Users, allora come sharding key si sceglie idUser come input in una funzione hashing per il partizionamento della collezione.

Anche le attività svolte dagli utenti userei la chiave sharding key.

Mentre per i path uguale. [?? non sicuro]

Consistency

Manual

Insieme di note (da sistemare opportunamente nella documentazione!!!)  
  
pillow1   
Questo discorso si adatta bene per la mia applicazione.  
“Our application perform mainly read operations (read heavy), so we decided to introduce indexes to speed up frequently queries. We introduce index regarding user and courses fields that aren’t editable after the first time. Thanks to that, there won’t be an overhead on updates queries.”

Pillow2  
Vediamo come gestire le operazioni CRUD della applicazione.  
Delete User ->   
1)salvataggio document dell’utente da cancellare  
1)cancellazione utente lato MongoDB  
2)salvataggio activity relative all’utente  
2)cancellazione attivita  
3)cancellazione utente lato Neo4j usanto il objectID del punto1  
4)cancellazione attività lato Neo4j usando gli objectID del punto2

Un problema(che non ho tempo per affrontare) è se si deve cancellare anche i path scoperti dall’utente e usati dalla community. In questo caso i path non li cancelleri perché diventati di dominio pubblico. Piuttosto le informazioni del path relative all’utente che l’ha scoperto dovrà venire scritto che l’utente è stato eliminato.

Delete Activity -> [fatto]  
1)Salvataggio activity dell’utente   
1)cancellazione della activity da parte MongoDB  
2)cancellazione della activity da parte Neo4j

Delete Path ->  
1)Salvataggio del path   
2)cancellazione path lato MongoDB  
3)cancellazione path lato Neo4j

Le attività relative al path cancellato non devono essere cancellate perché le attività dell’utente devono essere sempre visibili. Piuttosto il relativo bottone “Info path” non sarà cliccabile con scritto “Deleted”.  
Quindi sarebbe carino creare un documento “PATH DELETED” con uno specifico objectId da usare in MongoDB e Neo4j.