****

*Large Scale and Multi-structured Databases*

*PokeMongo: Project Documentation*

Edoardo Fazzari, Mirco Ramo, Olgerti Xhanej

Summary

[1. Introduction 3](#_Toc59093170)

[2. Analysis stage 4](#_Toc59093171)

[2.1 Functional requirements and use cases 4](#_Toc59093172)

[2.1.1 Use Cases List 4](#_Toc59093173)

[2.1.2 UML Use Case Diagram 7](#_Toc59093174)

[2.2 Non-functional requirements 8](#_Toc59093175)

[2.3 Sources, velocity properties and volume of data 8](#_Toc59093176)

[2.5 Main application queries 10](#_Toc59093177)

[3 Project Stage 11](#_Toc59093178)

[3.1 Adopted Databases 11](#_Toc59093179)

[3.2 Document Database 12](#_Toc59093180)

[3.2.1 Queries handled 12](#_Toc59093181)

[3.2.2 Entities handled 12](#_Toc59093182)

[3.2.3 Collections structure 13](#_Toc59093183)

[3.2.4 Indexes 14](#_Toc59093184)

[3.3 Graph Database 17](#_Toc59093185)

[3.3.1 Queries handled 17](#_Toc59093186)

[4 Implementation Stage 17](#_Toc59093187)

[5 Test stage 17](#_Toc59093188)

# Introduction

PokeMongo is a gaming application in which users compete each other to build up the best Team choosing between the set of Pokémon available in the environment; they can also follow other users in order to make new friends basing on common friends or common interests. Moreover users can express sentiments on Pokémon, choosing their favorite ones and posting/commenting on them.

Every trainer (normal user) can build up his own team. Every Team is composed by up to 6 distinct Pokémon and is assigned to a numerical value (points) based on features and properties of the chosen Pokémon, for ranking purposes.

Users can also navigate through the ranking in order to visualize the best teams (according to the values cited before) and the most used/caught Pokémon, both among their friends, grouped by country and among worldwide players.

User can browse for a specific Pokémon using the Pokédex tool, in which he/she can lookup for Pokémon according to search filters like Pokémon name, Type or Points.

Moreover, as a “real” Pokémon Trainer, the user is invited to *Catch ‘em‘ all*, i.e. to try to get a new Pokémon in order to create/update his/her own team. Thus, it is provided to the user a prefix number of daily Pokéball to be used to try to capture them. At each Pokémon is associated a probability to catch it, the higher the Pokémon’s value, the lower the probability.

Furthermore, the user can exploit the social network structure of the application to make new friends and discover new Pokémon. Indeed, he/she can search for new friends by username or choosing them among the provided *recommended friends* list.

The user can choose his/her favorite Pokémon, obtaining in this way a shortcut to catch it faster, and can *post* or *answer* to *posts* in order to express his/her opinion on that Pokémon.

In addition, to extend the dynamic behavior of the application, the *catch rate* (i.e. the probability to get a Pokémon using a Pokéball) changes in time depending on the number of users who have that Pokémon: the more it is popular, the harder will be to catch it. Since the rankings’ points are computed based on the *catch rate*, the winning strategy could be on predicting which Pokémon will become popular in the near future and try to get it early! Every user has access to the visualization of the temporal drift of the *catch rate*.

The safeguard and the improvement of the application is in charge of admin users. They are able to ban mischievous users, delete inappropriate posts or comments, add/remove Pokémon to the collection, consult geo-temporal usage statistics which are useful to make new business plans.

# Analysis stage

## 2.1 Functional requirements and use cases

### 2.1.1 Use Cases List

* An unregistered user can
  + - * + Register
* A registered user
  + - * + Sign in
        + Consult Pokédex:

Search by name

Search by type(s)

Search by Pokédex ID

Search by catch rate

Search by points

Search by Pokémon characteristics like height or weight

* + - * + Consult rankings:

Most popular Pokémon among all users

Most popular Pokémon in each country

Best world team

Best friends’ team

Best team by country

* + - * + Find users:

See recommended users based on common friends

See recommended users based on common Pokémon interests

Find users by username

Follow them

Unfollow them

* + - * + Interact with Pokémon network:

Insert a Pokémon in his/her own favorite Pokémon list

Remove a Pokémon from the favorite ones

Create a post on a Pokémon to share opinions

Add answers to posts

The post owner can also remove the post at his/her will

* + - * + Team handling:

View team

Remove Pokémon from the team

Change name to the team

Save modified team

View the value of the team

* + - * + Catching:

Browse a Pokémon you want to catch searching it by name

Select a Pokémon you want to catch from the list of favorites

Try to catch the Pokemon to add to your team

* + - * + Settings:

Change email

Change password

Change country

* + - * + Logout:

Exit from the account

Return to the sign in window

* + - * + At each time can:

See the remaining daily Pokéballs

Start/Mute music

By clicking on a Pokémon name, visualize all the information about it

* An admin can

Sign in

Add Pokémon to the Pokédex

Remove Pokémon from the Pokédex

See number of registered users in time

See number of logins per day

See number of logins per day in every country

Remove a user from the system

Remove posts from the network

Remove answers from the network

Consult rankings

Logout

* The system should

Daily update Pokéball number of each user

Periodically update Pokémon’ catch rates based on the number of users that own that Pokémon

Update team points if the user has 6 Pokémon of different types

Periodically compute usage statistics to be consulted by the administrators

### 2.1.2 UML Use Case Diagram

(\*) Only for the user who created the post

Browse-find-view comments and browse-find-view answers had not been reported

## 2.2 Non-functional requirements

* The application should guarantee a high availability
* It should be easy to use, especially for children and youngsters, and enjoyable
* It should have a read-your-own-writes consistency on each user’s own team, so he/she can always be sure that Pokémon have been correctly caught/freed up
* The application should always provide to each user the most recent version of the rankings in order to permit him/her to immediately verify his/her progresses
* The statistics regarding usage and catch rate evolution are not needed to be real-time, they can be updated periodically and be eventually consistent
* Posts, comments and answers must follow a causal-consistency
* Response time is an important issue: redundancies and larger memory consumptions are preferred over high latencies
* Passwords are crypted for security reasons
* A graphical interface and the usage of multimedia are crucial for an involving game experience

## 2.3 Sources, velocity properties and volume of data

Data stored in the application backend has been downloaded and imported from the following sources:

[*https://pokeapi.co*](https://pokeapi.co/), [*https://bulbapedia.bulbagarden.net/wiki*](https://bulbapedia.bulbagarden.net/wiki/Main_Page) 🡪 Pokémon data

[*https://gist.github.com/kalinchernev/486393efcca01623b18d*](https://gist.github.com/kalinchernev/486393efcca01623b18d) 🡪 Countries data

[*https://github.com/smashew/NameDatabases/blob/master/NamesDatabases/surnames/all.txt*](https://github.com/smashew/NameDatabases/blob/master/NamesDatabases/surnames/all.txt) 🡪 Data for the generation of realistic users

All the imported data has been modified, updated and preprocessed in order to satisfy the application needs.

Users added have the only purpose of showing the application functionalities, privacy issues they are not real people; anyway they have been created using realistic criteria.

Velocity is guaranteed by the dynamic *catch rate* mechanism: the popularity of a Pokémon influences both its *catch rate* and the amount of points that it will provide. As a consequence, users are continuously stimulated by catching new Pokémon, in order to try to raise their amount of points: in this way old teams’ data becomes quickly out-of-date.

Volume of data, considering 250K users, almost 1K Pokémon and about 500K posts is no lower than 100Mb

2.4 UML Entities Diagram

A User can build up only 1 Team: of course, each Team has just one owner.

A Team is composed of a maximum of 6 Pokémon, every Pokémon can be caught by anyone, so can belong to many Teams.

A User can follow many Users, in the meanwhile he/she can have many followers.

A User can have many favorites Pokémon. A Pokémon can be favorite of many Users.

A Post is created just by one User on one Pokémon. A User can create many posts and a Pokémon can have many Posts talking about it.

An answer is written by one User and it refers to one Post. Users can submit many Answers and there can be many Answers behind a Post.

## 2.5 Main application queries

* Insert a user into the system at registration time
* Create a new Pokémon (admin only)
* Insert a Pokémon into a team
* Create a new Post
* Create a new Answer
* Create a follow relationship
* Add a Pokémon to the favorites
* Retrieve user information at login time
* Retrieve a user by username when looking for a new friend
* Retrieve team information based on user
* Retrieve Pokémon information using several filters
* Retrieve recommended users
* Retrieve list of a user’s friends
* Retrieve a Pokémon by name when trying to catch it
* Retrieve all the posts relative to a Pokémon
* Retrieve all the answers to a post
* Retrieve user’s favorite Pokémon
* Modify user settings (email, password, country)
* Update team’s name
* Update team’s points
* Update Pokémon’ *catch rates* 🡪Analytics: find % of users that own that Pokémon
* Remove a user (admin only)
* Remove a Pokémon (admin only)
* Remove a post (only admin and post’s owner)
* Remove a follow relationship
* Remove a Pokémon from the favorite ones
* Analytics: ranking of most popular Pokémon in world/each country
* Analytics: ranking of best teams in the world/each country/among friends
* Analytics: evolution on time of a Pokémon *catch rate*
* Analytics: evolution on time of number of logins per day/total users/logins per day by country (admin only)

# 3 Project Stage

## 3.1 Adopted Databases

According to concept presented int the previous chapter we can make the following considerations:

1. Because of the performance constraint, a fast backend is required. Moreover, since the aim is to spread the application worldwide, the database infrastructure should be easy to distribute.
2. Pokémon must store heterogeneous data like URLs, different kinds of bios, float arrays and so on.
3. Users are divided into normal users and admins. Although the second ones are few, a denormalized approach could be better to handle the fact that these 2 categories have very different attributes.
4. Rankings are real-time OLAP queries: they need fast aggregation strategies.
5. Favorite Pokémon, friends, posts and answers together form a real Social Network.
6. A Team, in a normalized relational model, could be seen as a relationship table between Users and Pokémon. Anyway a huge table with a lot of duplicated PokémonID is not scalable for the requirements of this application. We need to find the best way to perform quickly both the retrieving of a user’s team and the ranking of the most used Pokémon, optimizing if possible memory consumption.

The points 1 to 4 guided the choice of a Document Database for handling User and Pokémon data. The flexibility, denormalization and performance of this kind of the database make it the most appropriate one.

The point 5 is best handled by a Graph DB, optimized for networks and different kinds of relationships. Moreover, we realized that the best way to handle a team is to decompose it in a set of Graph Relationships (USER – OWNS 🡪 POKEMON). Indeed, in this way queries mentioned at point 6 are very fast (just counting incoming/outcoming edges, see paragraph 3.3.1), and there are no useless, waste-memory repetitions of User IDs/Pokémon IDs.

Since each user can have only a team, team name and points are stored in the user collection.

## 3.2 Document Database

### 3.2.1 Queries handled

* Insert a user into the system at registration time
* Create a new Pokémon (admin only)
* Retrieve user information at login time
* Retrieve a user by username
* Retrieve Pokémon information using several filters
* Retrieve a Pokémon by name when trying to catch it
* Modify user settings (email, password, country)
* Update team’s name
* Update team’s points
* Update Pokémon’ *catch rates*
* Remove a user (admin only)
* Remove a Pokémon (admin only)
* Analytics: ranking of best teams in the world/each country/among friends
* Analytics: evolution on time of a Pokémon *catch rate*
* Analytics: evolution on time of number of logins per day/total users/logins per day by country (admin only)

### 3.2.2 Entities handled

Document Database stores information about USERS and POKEMON.

In particular it remembers user’s anagraphics and login data, last login, remaining Pokéballs, team name and earned points; a Boolean field distinguish admin from normal users. Admins have no points nor team or Pokéballs.

In a separate collection are stored data about Pokémon: PokédexId (*src: PokeAPI*), characteristics, one or more types, bio, images URLs, current *capture\_rate* and its last 30 *catch\_rates* stored into an array of Floats.

The details of the collections are reported in the following paragraph.

### 3.2.3 Collections structure

Immagine che contiene testo

Descrizione generata automaticamente

User Collection

Relevant attributes:

* *Admin*: TRUE🡪admin, FALSE🡪normal user
* *Username:* unique mnemonic ID of the user
* *Email:* must respect typical e-mail format
* *Password:* encrypted version of the user-chosen password
* *Last Login:* timestamp of the last time the user logged into the application
* *dailyPokeball:* number of daily Pokéballs left. They are up to 10 per day
* *points:* worth of his/her team

Immagine che contiene testo

Descrizione generata automaticamente

Pokémon Collection

Relevant attributes:

* *Id*: Pokédex ID (unique)
* *Name:* unique mnemonic ID of the Pokémon
* *Capture\_Rate*: current index of probability to catch the Pokémon
* *Portrait/Sprite:* URLs of the graphical representations of this Pokémon
* *Capture\_Rates:* array of the last 30 values of the *capture\_rate*, one for each of the last 30 days.

### 3.2.4 Indexes

* USERNAME

The first field in which we study the possibility of indexing is the *username* one in the *user* collection. A username is a REQUIRED and UNIQUE field of each user, and it is his/her mnemonic id inside the application.

The field username is involved in the following queries:

W1-)Insert a new username at registration time of an arbitrary user

W2-)Remove a username when an admin delete’s a user from the system

R1-)Check uniqueness of a username at registration time

R2-)Check user’s credential at login time

R3-)Find a user by username when a new follow request is submitted

Assuming that a registered user will play the game for about 100 days before “getting bored”, we can state that the number of logins-per-day will be 100 times the number of registrations-per-day: this means that the queries R1+R2 are submitted 101 times more than query W1.

Moreover, we can assert that query W2 will be very rare, while R3 is a popular query among the network structure of the application, say 30 times the number of registered users: we find out that read operations on this field are about 130 times the number of write operations.

Now consider MongoDb performances with and without using an index on the *username* field, in a Database populated by 250k users.

db.user.find({username:”eee”}, {username:1}).explain(“executionStats”)

Immagine che contiene testo, monitor, screenshot, elettronico

Descrizione generata automaticamente

In the picture on the right is reported the output of the query when we do not use an index. Execution time is huge due to the very high number of docs examined.

Immagine che contiene testo, screenshot, elettronico, computer

Descrizione generata automaticamenteOn the contrary, with an index, the same query need an execution time almost 100 times lower, and of course thanks to the index, DBMS only need to examinate one document. Moreover the unique property permits to eliminate the need of submitting query R1 at each registration

Considering the very high speed-up ratio of the indexing and the high frequency of this kind of queries w.r.t. the write operations (as explained before), a UNIQUE INDEX on *username* has been created.

COUNTRY

As seen before, starting from the application queries we demonstrate the benefits of an index in the field *country.*

W1-)Insert the country data at registration time

W2-)Remove all the user’s data if a user is banned by an admin

W3-)Changing of settings after a user changes residence’s country

R1-)Rank all users by country

R2-)Rank countries with the highest logins-per-day ratios

Let x be the number of registrations-per-day (W1), w.r.t this number W2 and W3 are very rare operations. Indeed, even though we can expect mischievous behaviors from some user, the number of country changes will never be comparable with x.

On the other hand, in order to guarantee a read-your-own-write eventual consistency on ranking R1, this query is recomputed every time a user asks to see the ranking itself. Thus, since the gameplay is highly based on rankings, we can estimate that R1 frequency will be about 400x.

Furthermore we have to consider R2. Despite the fact that this query is executed just once per day (so frequency(R2)<<x), it is an asynchronous procedure sensitive to execution time since it needs to lock the entire collection, make it unavailable to users for a while.

As seen before, let us compare DBMS performances with and without a *country* index.

db.user.find({country:"Italy"}).explain("executionStats")

Immagine che contiene testo, monitor, screenshot, elettronico

Descrizione generata automaticamenteConsidering again about 250k users, without an index we need to scan the whole database, which means a medium-high execution time for each request.

Immagine che contiene testo, screenshot, monitor, elettronico

Descrizione generata automaticamente

On the contrary, we have a very high increase of performances introducing and index on *country*: execution time is about 58 times lower and the only documents examined are the ones that must be returned.

To summarize, considering the difference in frequency between reads and writes and the high decrease of execution time, an index on *country* has been introduced.

Pokémon’s NAME

Queries on Pokémon’s name:

W1-) Insert a new Pokémon into the Database

W2-) Delete a Pokémon from the Database

R1-) Search a Pokémon by name in the Pokédex

R2-) Browse a Pokémon by name in *Catch’Em’All* in order to try to catch it

R3-) Check name’s uniqueness of each Pokémon when added to the database

Again, W1 and W2 are rare and admin-related operations: this means that this queries will not require a frequent update of the index. On the contrary R1 and especially R2 are very frequent gameplay queries inside the application: we can estimate that R1+R2 frequency will be several orders of magnitude higher than W1+W2 one.

R3 instead is a query always required before W1, but it can be managed by DBMS adding a unique property to the index, thus reducing computational cost of the operation itself

In terms of execution time, the final report is the following:

Immagine che contiene testo, elettronico, screenshot, schermo

Descrizione generata automaticamenteImmagine che contiene testo, screenshot, monitor, elettronico

Descrizione generata automaticamente

Find with index

Find without index

Even if we have little changes on execution time due to the limited number of Pokémon, we can see how the index permits to decrease very much the number of examined documents.

For the reasons explained before and because of the very high ratio between reads and writes, we consider this little improvement enough relevant for the application purposes.

## 3.3 Graph Database

### 3.3.1 Queries handled

|  |  |
| --- | --- |
| APPLICATION QUERIES | GRAPH QUERIES |
| Insert a user into the system at registration time | Insert a new USER node into the graph |
| Create a new Pokémon (admin only) | Insert a new POKEMON node into the graph |
| Insert a Pokémon into a team | Add a OWNS relationship between a USER node and a POKEMON node |
| Create a new Post | Add a new POST node into the graph |
| Create a new Answer | Add a new POST node into the graph |
| Create a Follow relationship | Add a new FOLLOW edge |
| Add a Pokémon to the favorites | Add a new IS FAVORITE edge |
| Retrieve team information based on user | Given USER u, retrieve all the POKEMON nodes connected to u through a HAS edge |
| Retrieve recommended users | Match all the USER nodes at distance 2 from the given USER node U according to one of these patterns(the first one has precedence)  (U)—FOLLOWS🡪(USER f)—FOLLOWS🡪(USER recomm.)  (U)—LIKES🡪(POKEMON p)🡨LIKES—(USER recomm.) |
| Retrieve list of a user’s friends | Match all the USER nodes linked to U by an outcoming FOLLOWS edge |
| Retrieve all the posts relative to a Pokémon | Match all the POST nodes which are linked to the POKEMON node P through a TOPIC edge |
| Retrieve all the answers to a post | Match all the POST nodes which are linked to the POSTnode P through a TOPIC edge |
| Retrieve user’s favorite Pokémon | Match all the POKEMON nodes which are linked to the USER node U through a LIKES edge |
| Analytics: find % of users that owns a Pokémon | Count outcoming HAS edges from the POKEMON node p.  Divide the result by the total number of USER nodes |
| Remove a user (admin only) | Delete a USER node |
| Remove a Pokémon (admin only) | Delete a POKEMON node |
| Remove a post (only admin and post’s owner) | Delete a POST node |
| Remove a follow relationship | Delete a FOLLOWS edge |
| Remove a Pokémon from the favorite ones | Delete a LIKES edge |
| Analytics: ranking of most popular Pokémon in world/each country | Count ni= number of HAS incoming edges of POKEMON node pi, for each POKEMON node.  Sort k highest n1…nk and return relative p1…pk |

3.3.2 Entities handled

3.3.3 Graph Structure

3.3.4 Indexes

3.4 Redundancies and consistency management

3.5 Db properties

3.5.1 Availability

3.5.2 Replicas

3.5.3 Eventual consistency

3.5.4 Sharding

3.5.5 Pros and drawbacks

3.6 Clients, servers, daemon threads

3.7 Technologies and frameworks

# 4 Implementation Stage

4.1 Package structure and information hiding

4.1.1 Packaging strategy and information hiding

4.1.2 UML package diagram

4.2 APIs and SPIs

4.3 Main tools

4.3.1 GSON

4.3.2 Caching mechanism and multimedia management

4.3.3 Password Encryptor

4.3.4 Logger

4.4 Analytics queries

4.4.1 User Rankings

4.4.2 Pokémon Rankings

4.4.3 Usage Statistics

4.4.4 Dynamic Catch Rate

4.5 Business logic

4.5.1 Points computing

4.5.2 Dynamic Catch Rate computing

# 5 Test stage

5.1 Privacy and security

5.2 Unit Tests

5.3 Robustness

5.4 Performance