Université de Bourgogne, Licence 2

LiChess

Rapport de projet

SALAH Ismail – ROBINET Perrine
01/03/2022

Sommaire

I- Présentation du projet

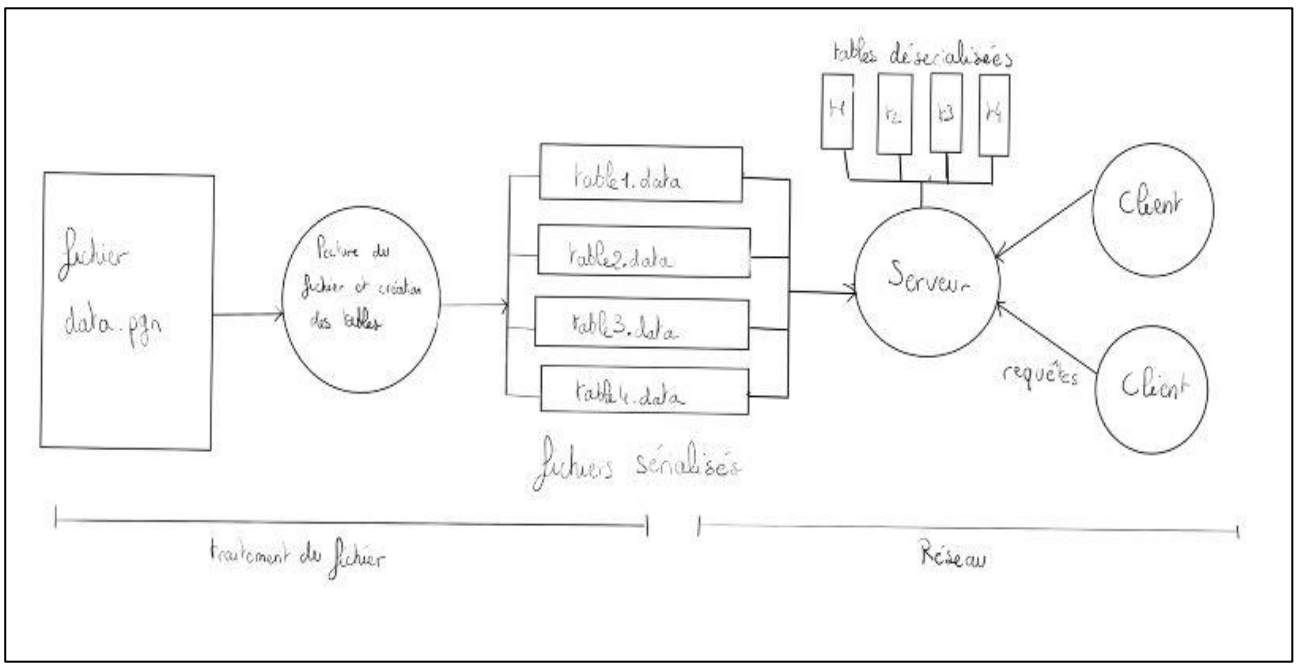
1. Choix du sujet

Nous avons choisi le sujet portant sur la base de données du jeu d'échec en ligne LiChess car nous étions intéressés par le problème que représentait la gestion, le tri d'un fichier de données et le fait de pouvoir retrouver une information précise dans une grande base de données. Nous avons donc divisé le projet en plusieurs parties. La première est le traitement du fichier pgn contenant toutes les informations en brut pour récupérer les données voulues et les organiser. La deuxième porte sur la partie réseau, plus précisément sur la partie traitement des requêtes d'un client.

Pour notre projet, nous voulions qu'un client puisse avoir accès à plusieurs informations tel que le nom des cinq ouvertures les plus jouées ainsi que le nombre de fois où elles ont été jouées. On doit pouvoir obtenir les cinq joueurs ayant joué le plus de parties et le nombre de parties qu'ils ont jouées. L'utilisateur peut également l'identifiant de toutes les parties d'un joueur et consulter l'ensemble des parties où deux joueurs spécifiques se sont affrontés. De même, on doit pouvoir afficher les adversaires de toutes les parties d'un joueur et enfin son elo.

2. Principe du projet

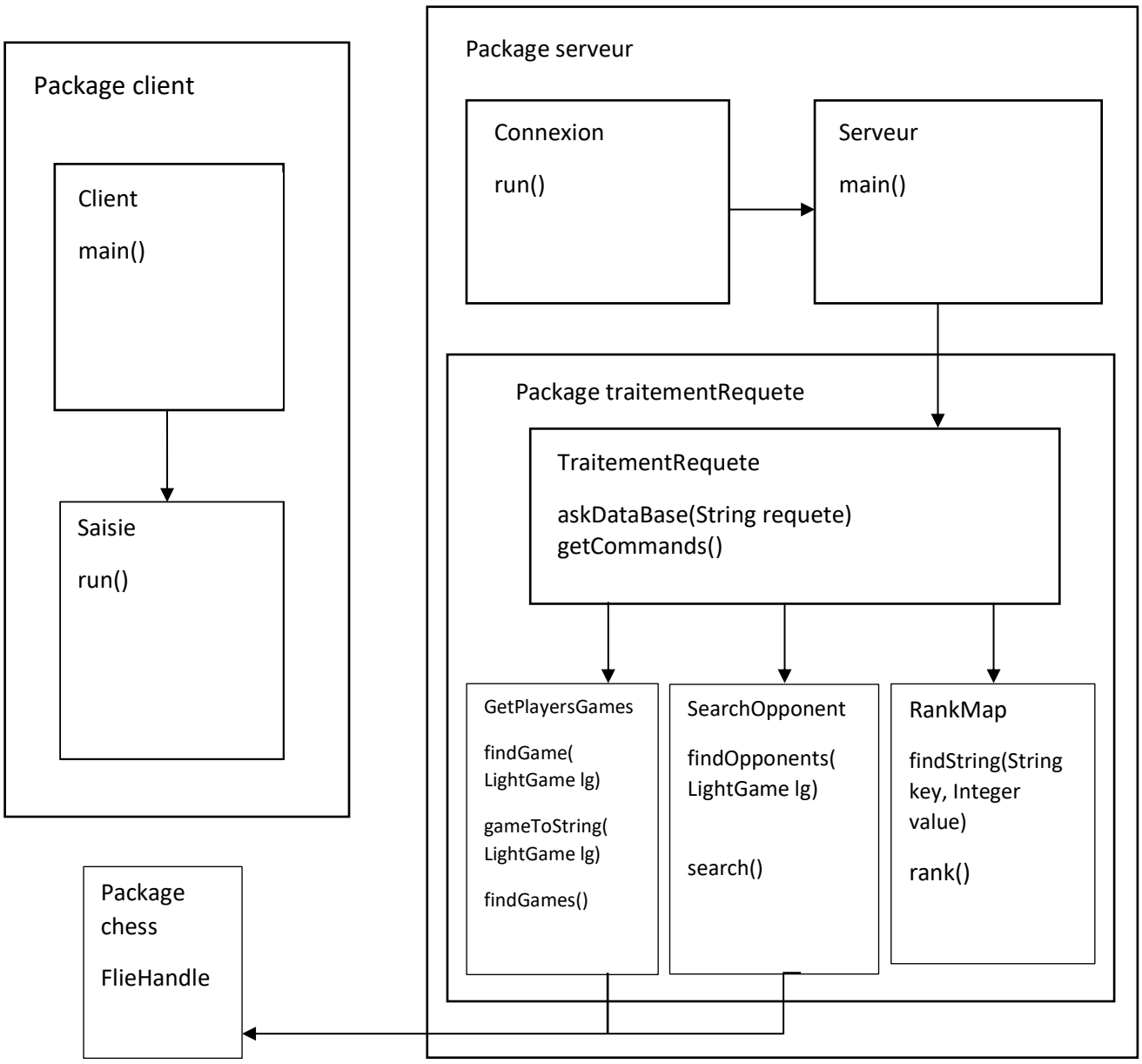
Le principe est de récupérer le fichier data.pgn que l'on parcourt tout en créant au passage les objets correspondant aux données récupérées(Player, Game) que l'on stocke dans des tables que l'on sérialise. Le serveur récupère ensuite les ConcurrentHashMap avec les informations voulues. Lorsqu'un client se connecte au serveur, il peut envoyer une requête. Le serveur la récupère, l'analyse, fait les calculs nécessaires et renvoie l'information souhaitée au client. Il peut y avoir au plus quatre clients connectés au serveur et les clients doivent pouvoir obtenir leurs réponses simultanément.

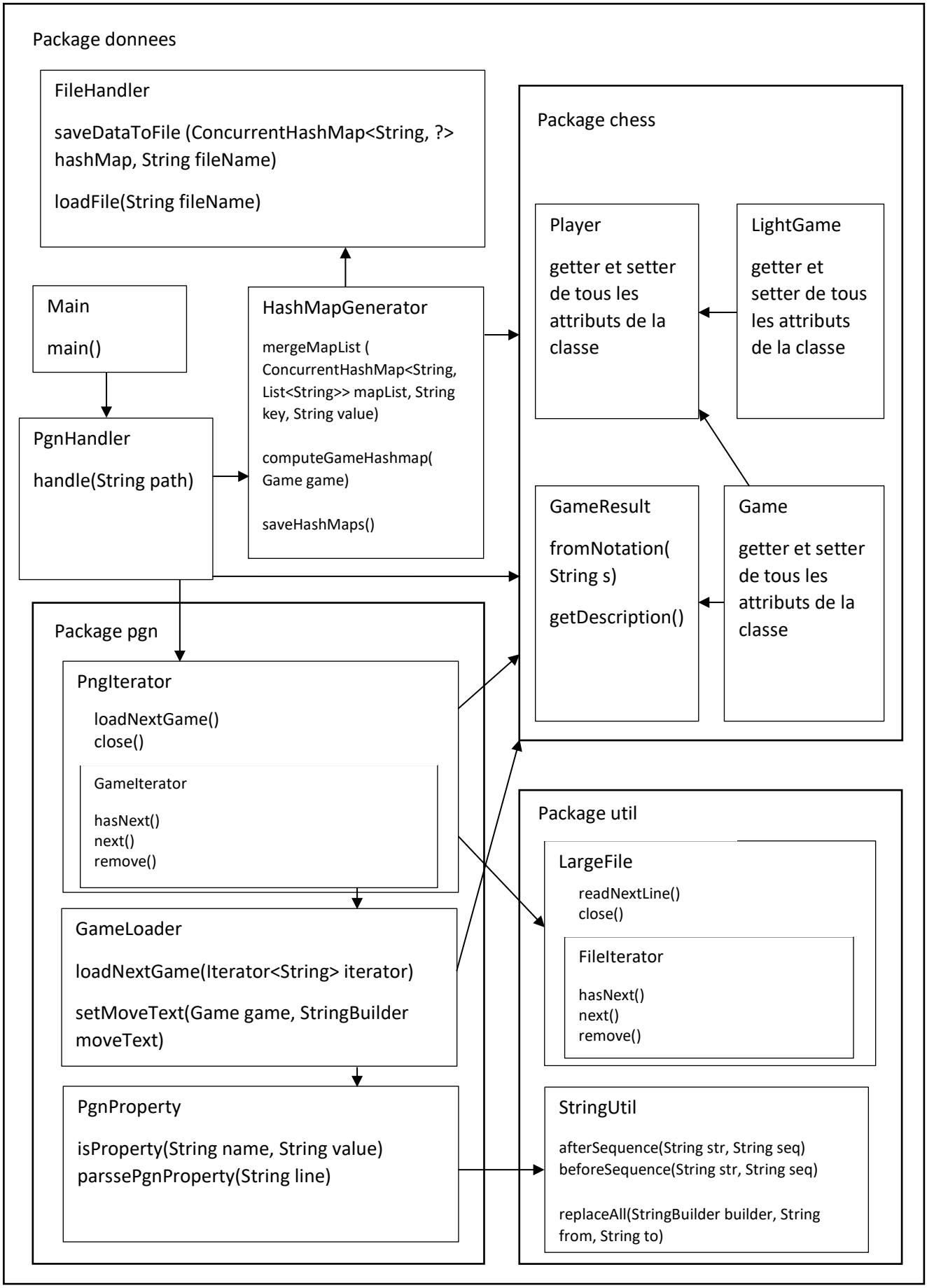


Nous avons été confrontés à plusieurs problématiques, la première étant comment parcourir le fichier et stocker de manière efficace dans les tables. Ensuite, comment faire en sorte que les fichiers sérialisés ne soient pas trop volumineux au niveau de la mémoire ? Quand faut-il charger les fichiers sérialisés au niveau du serveur ? En effet, si cette charge s'effectue à chaque requête, elle prolongerait le délais de réponse. Si cette charge s'effectue au démarrage du serveur, cela lui demanderait plus de temps pour être opérationnelle. Cela revient donc à se demander comment gérer le processeur, la mémoire, les entrées et les sorties.

3. Architecture du projet

Le projet est donc séparé en plusieurs fichiers et packages. Certaines classes sont à but utilitaire pour déléguer certains évènements ou calculs plus complexes et ainsi, éviter de surcharger certaines classes en code. Le package donnees regroupe toute la partie qui s'occupe du traitrement du fichier pgn. Le package serveur s'occupe de la partie serveur et le package client, de la partie client.




Package donnees
FileHandler
saveDataToFile (ConcurrentHashMap<String, ?> hashMap, String fileName)
loadFile(String fileName)
Main
main()
HashMapGenerator
mergeMapList (ConcurrentHashMap<String, List<String>> mapList, String key, String value)
computeGameHashmap(Game game)
saveHashMaps()
PgnHandler
handle(String path)
Package chess
Player
getter et setter de tous les attributs de la classe
LightGame
getter et setter de tous les attributs de la classe
GameResult
fromNotation(String s)
getDescription()
Game
getter et setter de tous les attributs de la classe
Package pgn
PngIterator
loadNextGame()
close()
GameIterator
hasNext()
next()
remove()
GameLoader
loadNextGame(Iterator<String> iterator)
setMoveText(Game game, StringBuilder moveText)
PgnProperty
isProperty(String name, String value)
parssePgnProperty(String line)
Package util
LargeFile
readNextLine()
close()
FileIterator
hasNext()
next()
remove()
StringUtil
afterSequence(String str, String seq)
beforeSequence(String str, String seq)
replaceAll(StringBuilder builder, String from, String to)

II- Traitement du fichier

Cette partie explique le raisonnement ainsi que les différentes classes utilisées pour réaliser le traitement du fichier pgn contenant les données des différentes parties.

1- Modélisation du jeu d'échec

Le package chess contient quatres classes représentant différents objets liés aux échecs, dans lesquels nous allons pouvoir stocker les données récoltées pour une partie de manière logique. La classe Player a pour attributs le pseudo du joueur ainsi que son elo.

La classe GameResult est une énumération représentant les différents résultats d'une partie, dont l'attribut principal est un String qui représente la description d'un résultat. Cette classe contient également une ConcurrentHashMap qui associe la clé lue dans le fichier à un résultat. On considère quatre cas: les blancs ont gagné, les noirs ont gagné, égalité et en cours. Ce dernier cas est utile lors de la création d'un jeu par défaut, avant que l'on récupère le résultat sur le fichier.

Une partie complète est représentée par la classe Game avec un attribut représentant son identifiant, un joueur blanc, un joueur noir, le résultat de la partie (GameResult), le nom de l'ouverture utilisée ainsi qu'une map qui associe une clé sous forme de String à un autre String. Cet attribut permet de récupérer les propriétés que l'on ne souhaite pas utiliser directement dans notre projet mais permet de modifier aisement le code si besoin. Cette classe contient également un attribut de type StringBuilder. Cette classe permet également de concaténer des chaînes de caractères en allouant un bloc de mémoire dès le départ pour ensuite y ajouter des caractères, ce qui permet une meilleur efficacité. Cet attribut permet de récupérer tous les coups d'une partie au fur et à mesure.

La classe Game contient beaucoup d'objets et d'informations donc nous avons décidé de faire une classe LightGame plus légère, par conséquent plus optimale pour un enregistrement dans un fichier. Elle a comme attributs: l'identifiant de la partie, son résultat, le joueur noir, le joueur blanc et l'ouverture. Elle ne possède pas les coups de la partie et les autres propriétés récupérées dans Game. Toutes ces classes ont été implémentées avec l'interface Serializable afin de pouvoir sérialiser les objets ainsi que les ConcurrentHashMap dans lesquels ils vont être stockés.

2- Outils dédiés à la lecture du fichier

Le package util contient les classes StringUtil et LargeFile qui vont faciliter la lecture du fichier. La classe StringUtil contient trois méthodes utilitaires pour gérer les chaînes de caractères. La méthode afterSequence permet de retourner la partie après une séquence donnée dans une chaîne de caractères. La méthode beforeSequence permet de retourner la partie avant une séquence donnée dans une chaîne de caractères. Enfin, elle possède une méthode replaceAll qui permet de changer une partie d'une chaîne de caractères par une autre chaîne.

La classe LargeFile permet de gérer les fichiers lourds. En effet, charger le fichier d'un seul coup poserait des soucis au niveau de la mémoire, elle pourrait même ne pas du tout le charger. Pour résoudre le problème, cette classe permet de lire un fichier ligne par ligne. Pour cela, on utilise

l'interface Iterable. Cette interface permet de signaler que la classe qui l'implémente est composée d'un ensemble de sous-éléments que l'on peut parcourir. Dans notre cas, ces éléments sont des lignes de texte. La classe LargeFile a pour attribut un BufferedReader qui va être utilisé pour simplifier la lecture du fichier et une chaîne de caractères qui va permettre de stocker la ligne suivante. Elle contient le constructeur d'un Iterator de la sous-classe FileIterator détaillée ci-dessous et une méthode permettant d'aller à la ligne suivante qui est utilisée par cette même sous-classe.

La sous-classe FileIterator avec l'interface Iterator va permettre d'itérer les éléments de la classe à parcourir, c'est-à-dire les lignes. Cette sous-classe contient une méthode pour savoir s'il y a encore une ligne après la ligne courante, une méthode qui permet de changer de ligne en retournant la ligne suivante et une méthode pour enlever un élément.

3- Construction des objets à partir des données du fichier

En examinant le fichier de base, nous avons de suite remarqué la forme récurrente qu'une partie prenait. En effet, toutes les parties sont de la forme :

[Event "Rated Blitz game"]
[Site "https://lichess.org/ijPYwgVx"]
[White "ljubisa2810"]
[Black "amirmahdi"]
[Result "1-0"]
[UTCDate "2014.03.31"]
[UTCTime "22:01:18"]
[WhiteElo "1806"]
[BlackElo "1644"]
[WhiteRatingDiff "+7"]
[BlackRatingDiff "-7"]
[ECO "B40"]
[Opening "Sicilian Defense: Delayed Alapin Variation"]
[TimeControl "180+0"]
[Termination "Normal"]

1. e4 e6 2. Nf3 c5 3. c3 b6 4. Bb5 Bb7 5. O-O Bxe4 6. d4 Bc6 7. a4 Bxb5 8. axb5 Nf6 9. Rfe1 cxd4 10. Nxd4 Bc5 11. b4 Bxd4 12. Qxd4 d5 13. Bg5 O-O 14. Nd2 h6 15. Bh4 Nbd7 16. Re3 e5 17. Qd3 g5 18. Rg3 e4 19. Qe3 Kh7 20. Bxg5 hxg5 21. Qxg5 Ng4 22. Rh3+ Nh6 23. Rxh6# 1-0

A partir de là, nous avons cherché un moyen de découper les données lues dans le fichier en propriétés compréhensibles par l'application. C'est le rôle de la classe PngProperty. Dans cette classe, on utilise un attribut de type Pattern qui va permettre de retrouver toutes les propriétés de la forme [name "value"]. On déclare également deux chaînes de caractères qui vont permettre de récupérer le nom du type de données et sa valeur. Cette classe contient également la méthode isProperty(String line) qui va permettre de vérifier que la ligne courante correspond au paterne et la méthode parsePgnProperty(String line) qui va permettre de découper la ligne courante en remplaçant les crochets par des espaces. De plus, grâce aux méthodes de la classes StringUtil, on récupère les chaînes de caractères avant et après l'espace central pour avoir les attributs name et value et on retourne une instance de PgnProperty avec ces valeurs.

On a donc tous les outils nécessaires pour convertir les données lues dans le fichier en un objet concret. Deux classes ont pour rôle de créer les parties. Premièrement, la classe GameLoader qui va permettre de créer les instances de Game. Ensuite, la classe PgnIterator qui contient un itérateur pour les lignes (récupérées à partir d'une instance de LargeFile) et une sous-classe GameIterator qui va permettre de parcourir toutes les parties une fois que celles-ci ont été lues dans le fichier d'entrée.

Commençons par la classe PngIterator. Elle contient une méthode loadNextGame() qui permet de générer la partie suivante à partir des données du fichier. L'Iterateur de parties permet, lorsqu'on a fini de constituer une partie, de passer à la suivante et de récupérer celle qu'on vient de créer grâce à la méthode next() qui fait appel à la méthode loadNextGame(). Il permet donc de passer à la partie suivante tout en gardant la ligne où l'on s'était arrêté grâce à l'Itérateur de LargeFile. La partie créée est stockée dans GameInterator.

Nous allons maintenant préciser comment la classe GameLoader construit les instances de Game. Cette classe est constituée de deux méthodes: la méthode loadNextGame(Iterator<String> Iterator) et la méthode setMoveText(Game game, StringBuilder moveText). La méthode loadNextGame(Iterator) est la méthode principale de la classe. Cette méthode contient des variables locales de type Game, deux de type Player, un StringBuilder et un boolean pour savoir si on a déjà récupéré les coups de la partie. Cette dernière variable est nécessaire car cette information est la seule qui ne correspond pas au paterne décrit plutôt. Il a donc fallu trouver un moyen de signaler le fait d'avoir déjà récupéré ces informations. Tant qu'il y a encore une ligne suivante dans le fichier, on la récupère. On enlève les espaces à la fin de la ligne avec la commande trim() et si la ligne correspond au paterne, on crée une nouvelle instance de PngProperty à partir de la ligne. Ensuite, on récupère la variable name qui va nous informer sur le type d'information que l'on récupère. Selon l'information récupérée, on va stocker sa valeur dans l'instance de Game (parfois par l'intermédiaire d'instance d'autres objets comme Player). Si la ligne obtenue ne correspond pas au paterne, on récupère la ligne de coups et si elle se termine par le score, on utilise la fonction setMoveText qui, grâce à la fonction replaceAll définie dans StringUtil, remplace le score par un espace vide. A la fin de ce if, on a récupéré toutes les informations de la partie. On sort de la boucle grâce à l'instruction break et on retourne l'instance de Game créée et complétée.

4- Construction des structures de données organisées

Après avoir résolu les problèmes liés à la lecture et l'analyse du fichier et avoir réussi à extraire les informations voulues, il nous a fallu organiser les objets obtenus en plusieurs ConcurrentHashMap. Cela permet de rechercher efficacement une information particulière, ce qui était nécessaire pour la partie réseau du projet. Les ConcurrentHashMap présentent plusieurs avantages par rapport aux HashTable. En effet, bien que les deux structures soient capable de gérer la concurrence liée aux threads. Lorsqu'un thread accède à une ConcurrentHashMap, celle-ci ne bloque que la partie appelée segment de la map auquel le thread accède, au contraire d'une HashTable qui bloque entièrement l'accès aux autres threads sur toute la table.

Les ConcurrentHashMap permettent donc à de multiples threads d'y accéder en même temps ce qui va permettre un tri des données bien plus rapide et efficace. De plus, il est possible d'utiliser la classe Iterator de manière sûre ce qui permet de faciliter également les recherches lors du traitement de la requête du client lors de la partie réseau. Elles sont également sérialisables ce qui est exactement le type de structure que l'on recherchait pour ce projet.

La classe HashMapGenerator permet de gérer la création des différentes ConcurrentHashMap. Nous avons décidé de quatre ConcurrentHashMap différentes: une qui associe l'identifiant d'une partie à une version allégée de la partie, une qui associe le pseudo d'un joueur à son objet Player plus complet, une qui associe le pseudo d'un joueur à une liste contenant les identifiants de toutes les parties auxquelles il a participé et enfin, une qui associe un nom d'ouverture à la liste des pseudos des joueurs qui l'ont exécutée. La classe contient deux méthodes essentielles au projet: computeGameHashmap(Game game) et saveHashMap(). La première sert à remplir les ConcurrentHashMap à partir d'une partie. Nous avons rencontré un problème dans cette fonction: remplir les valeurs associées aux clés lorsque celles-ci étaient sous forme de liste. En effet en utilisant la fonction put(key, value), nous écrasions la valeur précédente contenue dans value. Nous avons cependant réussi à trouver une solution en créant la fonction ci-dessous :

```
public void mergeMapList( ConcurrentHashMap<String, List<String>> mapList, String
key, String value) {
        if(!mapList.containsKey(key)) {
            List<String> list = new ArrayList<>();
            list.add(value);
            mapList.put(key, list);
        } else {
            mapList.get(key).add(value);
        }
    }
```

Si la clé n'était pas présente dans la ConcurrentHashMap, on crée une nouvelle liste dans laquelle on ajoute la valeur associée à la clé lue, puis on associe la liste comme valeur à cette clé dans la HashMap. Sinon, on récupère la liste déjà associée à la clé et on ajoute la valeur souhaitée.

La classe contient également un compteur qui va nous permettre d'afficher, à chaque appel de la fonction, le numéro de la partie qui vient d'être mise dans la ConcurrentHashMap dans un but purement esthétique. Comme plusieurs threads vont utiliser cette fonction en même temps, cette variable est une ressource sur laquelle il y a une concurrence. On a donc ajouté un moniteur pour éviter que deux parties aient le même numéro. Cependant, cela impacte fortement l'efficacité lors de l'exécution du programme. La dernière méthode de cette classe est donc saveHashMaps qui va permettre d'enregistrer les ConcurrentHashMap sous forme de fichier sérialisé grâce à la méthode saveDataToFile et d'afficher, sur le terminal sur lequel on a lancé l'exécution, des messages pour vérifier que les données s'enregistrent bien.

La méthode saveDataToFile fait partie de la classe FileHandler. Cette classe est une classe utilitaire dédiée à la gestion des flux pour les fichiers, plus précisément aux ConcurrentHashMap sérialisés. Pour la méthode saveDataToFile(ConcurrentHashMap<String, ?> hashmap, String fileName), on utilise la classe FileOutputStream pour créer un flux sortant et la classe ObjectOutputStream pour écrire sur le fichier les données des ConcurrentHashMap en utilisant le principe de la sérialisation puis on ferme les flux ouverts. Le ? est du au fait que l'on a différents types de valeurs possibles. La deuxième méthode de la classe est loadFile(String fileName). Pour cette méthode on utilise les classes FileInputStream et ObjectInputStream qui permettent de récupérer le fichier et le désérialiser pour ensuite récupérer la ConcurrentHashMap.

Enfin, les deux dernière classes de cette partie du projet sont la classe PgnHandler et la classe Main qui lance le traitement du fichier. Cette classe gère les threads qui vont servir à traiter le fichier

avec les méthodes décrites tout au long de cette partie du rapport. Cette classe comporte une unique méthode handler(String path) :

```
try {
            File file = new File(path);
            if(!file.exists()) {
                throw new Exception("Le fichier "+file.getName()+" n'existe pas");
            }
            HashMapGenerator hashMapService = new HashMapGenerator();

             //Récupération du nombre maximal de thread disponible pour la création
du pool de thread
            int processors = Runtime.getRuntime().availableProcessors();
            ExecutorService executor = Executors.newFixedThreadPool(processors);
            PgnIterator games = new PgnIterator(file.getAbsolutePath());

            for (Game game: games) {
                executor.execute(() -> hashMapService.computeGameHashmap(game));
            }

            executor.shutdown();

            //On attend que tous les threads se terminent pour continuer l'execution
            var correctlyTerminated = executor.awaitTermination(10,
                                              TimeUnit.MINUTES);

            if(correctlyTerminated) {
                System.out.println("Le traitement a bien terminé");

                //On enregistre les hashtables sur le disque.
                hashMapService.saveHashMaps();
            } else {
                System.out.println("Le traitement des données a dépassé le
                                    délais");
        } catch (Exception e) {
            e.printStackTrace();
        }
```

Dans cette méthode on récupère le fichier pgn puis si le fichier existe bien, on crée une nouvelle instance de HashMapGenerator puis on récupère le nombre de processeurs de la machine virtuelle Java disponibles. On utilise ensuite la classe ExecutorService où l'on crée une nouvelle ThreadPool d'une taille fixe qui correspond dans notre cas au nombre maximum de processeurs disponibles dans la machine virtuelle Java. Autrement dit, le nombre maximum de threads disponibles dans la piscine ne dépasse jamais le nombre de processeurs disponibles.

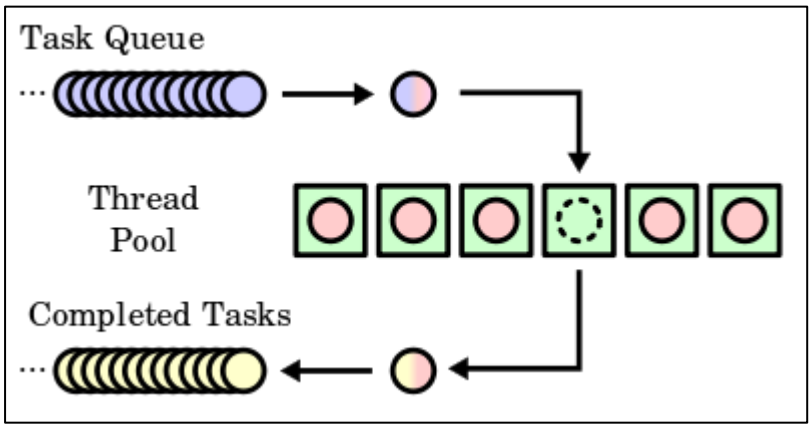
Cette ligne : `executor.execute(() -> hashMapService.computeGameHashmap(game));`

N'est qu'une expression « lambdas »disponible depuis java 8 afin de simplifier le code suivant(disponible que si la classe implémente l'interface runnable ou):

```
executor.excute(new Runnable() {
      @Override public void run(){
            //fonction du thread
      }
})
```

Ou les deux premières parenthèses font référence à la fonction run et à l'intérieur des accolades on définie la fonction des threads.

Nous allons maintenant détailler un peu plus en détail le fonctionnement des threads pool pour expliquer notre choix d'en utiliser tout au long du projet. Les threads pool sont des collections de threads pré-initialisés. Cette pré-initialisation permet de faciliter de crée un d'un certain nombre de tâches qui doivent utiliser les mêmes threads pour s'exécuter: par exemple, dans notre cas, faire exécuter la fonction computeGameHashMap pour chaque partie. S'il y a plus de tâches à faire que de threads disponibles, alors les tâches sont mises dans une file d'attente qui est gérée en FIFO.



Cette façon de faire présente donc plusieurs avantages: le premier est qu'il n'est pas nécessaire de lancer manuellement tous les threads. De plus, on peut avoir un maximum de threads disponibles par rapport à la capacité du processeur et donc répartir au maximum la charge de travail. Comme on optimise au maximum le processeur, on traite l'entièreté du fichier bien plus efficacement car les threads fonctionnnent en parallèle. On crée ensuite une nouvelle instance de PgnIterator et pour chaque partie on fournit au thread pool la tâche de remplir les ConcurrentHashMap avec les informations de cette même partie. Pour cela, on utilise la fonction execute qui permet d'exécuter la tâche par les threads sans obtenir de valeur retour. Une fois que l'on a donné cette tâche pour toutes les parties, on utilise la méthode shutdown pour signaler au thread pool qu'aucune nouvelle tâche sera accepteée puis on attend que toutes les tâches entrées au préalable soient exécutées par les threads. Si c'est le cas on indique, que tout va bien. Sinon, on indique qu'il y a un problème. Enfin, le Main sert à récupérer le fichier (saisi par l'utilisateur) et à lancer la fonction handle.

III- Réseau

On va maintenant traiter de la partie réseau du projet, de l'envoi de la rêquete du client jusqu'au traitement de la requête côté serveur. Le principe de cette partie est simple: un utilisateur lance le client et tape une ligne de commande spécifique selon les informations qu'il souhaite avoir. Le serveur récupère la requête, l'analyse et cherche la réponse auprès des ConcurrentHashMap. Une fois que la réponse a été trouvée, on la renvoie au client.

1- Client

Cette partie comporte deux classes: la classe Saisie et la classe Client. La classe Saisie est un thread qui a pour but de gérer la saisie au clavier. On crée un nouveau socket et à partir de ce socket, on crée un nouvel InputStreamReader et un nouveau BufferedReader ce qui va nous permettre de récupérer les réponses aux requêtes que l'utilisateur va envoyer. On crée également un OutputStreamWriter et un nouveau PrintWriter qui va permettre d'envoyer la requête au serveur. On crée un nouveau thread de saisie et on l'initialise puis on démarre une boucle: tant que l'utilisateur n'a pas signalé la fin de la communication, on récupère la réponse du serveur et on l'affiche sur le terminal de l'utilisateur. Pour la classe Saisie, on récupère le PrintWriter du client grâce au constructeur de la classe et on a également un BufferedReader lié à un InputStreamReader qui prend en paramètre System.in pour pouvoir récupérer la ligne de commande saisie dans le terminal. Dans le run, tant que l'utilisateur n'a pas saisi la requête "END", on récupère la liste des commandes disponibles que le serveur envoie et on l'affiche. Puis, on récupère la requête de l'utilisateur et on l'envoie au serveur.

2- Serveur

Il y a deux problèmes principaux concernant le serveur: il faut d'une part que plusieurs clients puissent se connecter en même temps et que leurs requêtes soient traitées simultanément et d'une autre part, il faut gérer le traitement de la requête en lui-même, c'est-à-dire faire les calculs nécessaires pour retrouver les informations souhaitées. Commençons d'abord par la partie communication. Les classes Serveur et Connexion permettent de s'occuper de cette partie. Pour le serveur, on déclare un nombre maximum de connections pour éviter une surcharge du chargeur. Un tableau de PrintWriter va contenir le PrintWriter pour chaque client. Un entier servant de numéro d'identification et une variable de type TraitementRequete va servir à traiter les requêtes mais nous reviendrons sur cette classe un peu plus tard TraitementRequete. La classe Serveur ne contient que le main. On crée un nouveau serveur et on affiche un message lorsque le serveur a bien été mis en route. Tant que le prochain numéro d'identification client est inférieur ou égale au nombre maximum de connections, on crée un nouveau socket à l'écoute d'une potentielle connection puis on crée une nouvelle instance de Connexion. La classe gère la connection avec un client puis on incrémente le numéro d'identifiant pour la connection suivante et on commence l'exécution du thread qui gère la connection avec le client.

La classe Connexion est donc la classe qui va gérer la connection avec un client. Il y a un avantage à concevoir les connections sous forme de threads: cela permet de gérer plusieurs connections de clients en parallèle, ce qui permet aux utilisateurs de ne pas avoir à attendre que le serveur est fini de traiter la requête du client précédent pour passer à la leurs. C'est donc un gain de temps considérable. La classe Connexion a pour attributs le socket auquel la connection est associée, son identifiant, un BufferedReader et le PrintWriter qui vont permettre la communication

entre le serveur et le client. On crée une boucle qui tourne tant que le client n'envoie pas la commande "END". Dans la boucle, on envoie au client la liste de commandes disponibles et on récupère la requête du client. Si la requête est END, on sort de la boucle sinon on trouve la réponse à la requête grâce à la fonction askDataBase de la classe TraitementRequete puis on envoie la réponse au client.

3- Traitement de la requête du client

Passons maintenant à la partie liée au traitement de la requête. Toutes les classes utiles à cette fonction sont dans le package traitementRequete. Cette classe utilise notamment le package chess et la classe FileHandler décrits plus tôt. La classe principale de ce package est la classe TraitementRequete. Cette classe va charger les ConcurrentHashMap sérialisées à l'appel du constructeur ce qui par conséquent, fait charger les ConcurrentHashMap à l'ouverture du serveur. C'est un choix que nous avons pris car, bien que cela cause un délai avant que le serveur ne soit opérationnel, le traitement des réponses pour le client est quasiment instantané et donc apporte un confort non négligeable pour l'utilisateur. La classe contient deux méthodes essentielles: la méthode getCommands() et la méthode askDataBase(String requete). La première retourne une chaîne de caractères avec la liste des commandes disponibles ainsi que leur format. La deuxième est une sorte de menu qui va permettre d'analyser la requête et de lancer la recherche correspondante dans les ConcurrentHashMap. Les calculs plus complexes sont délégués à d'autres classes plus spécialisées. La requête étant formulée est transmise sous forme de chaîne de caractères. Nous avons dû trouver un moyen de récupérer chaque partie de la requête sachant que toutes les requêtes sont de la forme "Commande paramètre1 …" avec le nombre de paramètres adapté à la commande voulue. Nous avons décidé d'utiliser un StringTokenizer pour découper la requête en plusieurs token avec comme critère de découpe les espaces. On récupère d'abord la commande qui va nous permettre de faire un switch en fonction de la commande. L'uilisation d'un switch est pratique car il permet de modifier facilement la fonction pour enlever ou rajouter des commandes. Selon les différentes commandes, on récupère les tokens suivants nécessaires pour trouver les informations. Pour trouver une information simple dans une ConcurrentHashMap, on utilise la commande get(..). C'est le cas pour les commandes AllGames, GetEloPlayer et GetGameById. Dans le cas des commandes qui demandent les cinq clés associées aux listes les plus grandes comme les cinq joueurs ayant le plus joués (c'est-à-dire les cinq joueurs ayant la plus grande liste d'adversaires), les calculs sont délégués à la classe RankMap.

La classe RankMap contient deux méthodes principales: la méthode findSize(String key, Integer value) et la méthode rank(). La méthode findSize() permet de remplir une nouvelle ConcurrentHashMap en gardant les clés de la ConcurrentHashMap que l'on veut classer et en mettant en valeur la taille de la liste que la valeur de la HashMap que l'on veut classer contenait. Voici le code de la méthode rank dont nous allons expliquer le principe.

```
public ArrayList rank() {
        try{

            //Récupération du nombre maximal de thread disponible pour la création
du pool de thread
            int processors = Runtime.getRuntime().availableProcessors();
            ExecutorService executor = Executors.newFixedThreadPool(processors);
            for (String key: keyToValue.keySet()) {
                executor.execute(() -> findSize(key,
                                        keyToValue.get(key).size()));
                }
                executor.shutdown();

            //On attend que tous les threads se terminent pour continuer l'execution
                    var   correctlyTerminated   =   executor.awaitTermination(10,
TimeUnit.MINUTES);

            if(correctlyTerminated) {

                                    List<Entry<String,   Integer>>   list   =   new
ArrayList<>(keyToNbValue.entrySet());
                list.sort(Entry.comparingByValue());

                //recuperation des 5 derniers elements de la hashtable
                for(int i=keyToValue.size()-5;i<keyToValue.size();i++){
                    result.add(list.get(i));
                }
            } else {
                System.out.println("Le traitement des données a dépassé le délais");
            }
        }catch (Exception e) {
                e.printStackTrace();
                }

        return result;
        }
```

On reprend la méthode vue plus tôt pour traiter une grande quantité de données à l'aide d'un thread pool. On crée un nouveau thread pool pour remplir la nouvelle ConcurrentHashMap. Pour chaque clé de la HashMap à classer, on récupère la taille de la liste associée et on met à jour la nouvelle HashMap grâce à la fonction findSize. Si tous les thread du thread pool sont finis, on crée une nouvelle instance de List<Entry<String, Integer>> qui, à chaque clé de la map, associe sa valeur. L'avantage d'utiliser cet objet est qu'il est possible de trier la liste par ordre croissant des valeurs. On trie donc la liste grâce à la fonction sort() et on ajoute les cinq derniers éléments de la liste dans la liste de résultat. On utilise le même principe de recherche pour les autres classes. Dans la classe SearchOpponent qui permet de trouver les adversaires de toutes les parties d'un joueur, la fonction findSize() est remplacée par la fonction findOpponents qui va ajouter dans la liste des résultats le joueur blanc si le joueur entré était le joueur noir ou inversement. De même, la classe GetPlayersGames permet de récupérer une liste contenant une description de toutes les parties que deux joueurs ont joué ensemble. Dans cette classe, le thread pool exécute la fonction findGame() qui va récupérer la description des parties souhaitées à partir de la ConcurrentHashMap regroupant l'ensemble des identifiants des parties qui sont associé à la partie allégée. On parcourt, de cette manière, l'ensemble des parties et lorsqu'une partie a les deux joueurs voulus, on ajoute la description de la partie à la liste des résultats.

IV- Jeux tests

1- Traitement du fichier pgn

Jeu nÂ°810452 : ZMAJ vs pseudoknight

Jeu nÂ°810453 : Tanjus-TR vs ShPad

Jeu nÂ°810454 : yasminoah vs pikero

Jeu nÂ°810455 : Kobylka vs accelix

Jeu nÂ°810456 : belja vs shahrokh20

Jeu nÂ°810457 : sidacotel vs sportloto

Jeu nÂ°810458 : Doctor2002 vs Drummied

Jeu nÂ°810459 : jjebolin vs SvSvHristov

Jeu nÂ°810460 : Daihatsu vs marinkatomb

Jeu nÂ°810461 : nikolam vs RYAPS

Jeu nÂ°810462 : CONTRADIQUE vs mightymiti

Jeu nÂ°810463 : HarmonySmite vs zinderi

Le traitement a bien terminÃ©

Enregistrement des jeux... OK

Enregistrement des joueurs... OK

Enregistrement des joueurs associÃ©s aux jeux... OK

Enregistrement des ouvertures des joueurs... OK

2- Terminal du client

LiChess>java Client localhost

Pour obtenir toutes les parties(id) d'un joueur: AllGames nomJoueur
Pour obtenir l'elo d'un joueur: GetEloPlayer nomJoueur
Pour obtenir une partie Ã partir d'un id: GetGameById partieId
Pour obtenir les 5 joueurs ayant realises le plus de partie: MostPlayedPlayers
Pour obtenir les 5 ouverture les plus jouees: MostPlayedOpens
Pour obtenir les parties entre deux joueurs: GetGamesByPlayers joueur1 joueur2
Pour obtenir les adversaires de toutes les parties d'un joueur: GetPlayerOpponents joueur
Pour signaler que vous avez fini vos recherche: END

AllGames Plifplaf
[f581180c-d3a7-43e5-9fe6-5f75cbd6df86]

Pour obtenir toutes les parties(id) d'un joueur: AllGames nomJoueur

Pour obtenir l'elo d'un joueur: GetEloPlayer nomJoueur
Pour obtenir une partie Ã partir d'un id: GetGameById partieId
Pour obtenir les 5 joueurs ayant realises le plus de partie: MostPlayedPlayers
Pour obtenir les 5 ouverture les plus jouees: MostPlayedOpens
Pour obtenir les parties entre deux joueurs: GetGamesByPlayers joueur1 joueur2
Pour obtenir les adversaires de toutes les parties d'un joueur: GetPlayerOpponents joueur
Pour signaler que vous avez fini vos recherche: END

GetEloPlayer Plifplaf
1500

Pour obtenir toutes les parties(id) d'un joueur: AllGames nomJoueur
Pour obtenir l'elo d'un joueur: GetEloPlayer nomJoueur
Pour obtenir une partie Ã partir d'un id: GetGameById partieId
Pour obtenir les 5 joueurs ayant realises le plus de partie: MostPlayedPlayers
Pour obtenir les 5 ouverture les plus jouees: MostPlayedOpens
Pour obtenir les parties entre deux joueurs: GetGamesByPlayers joueur1 joueur2
Pour obtenir les adversaires de toutes les parties d'un joueur: GetPlayerOpponents joueur
Pour signaler que vous avez fini vos recherche: END

GetGameById f581180c-d3a7-43e5-9fe6-5f75cbd6df86
Black player: Reinfield
White player: Plifplaf
Ouverture: French Defense: Pelikan Variation
RÃ©sultat: BLACK_WON

Pour obtenir toutes les parties(id) d'un joueur: AllGames nomJoueur
Pour obtenir l'elo d'un joueur: GetEloPlayer nomJoueur
Pour obtenir une partie Ã partir d'un id: GetGameById partieId
Pour obtenir les 5 joueurs ayant realises le plus de partie: MostPlayedPlayers
Pour obtenir les 5 ouverture les plus jouees: MostPlayedOpens
Pour obtenir les parties entre deux joueurs: GetGamesByPlayers joueur1 joueur2
Pour obtenir les adversaires de toutes les parties d'un joueur: GetPlayerOpponents joueur
Pour signaler que vous avez fini vos recherche: END

MostPlayedOpens
[Owen Defense=12272, Scandinavian Defense: Mieses-Kotroc Variation=13348, Horwitz Defense=15072, Modern Defense=15405, Van't Kruijs Opening=23439]

Pour obtenir toutes les parties(id) d'un joueur: AllGames nomJoueur
Pour obtenir l'elo d'un joueur: GetEloPlayer nomJoueur
Pour obtenir une partie Ã partir d'un id: GetGameById partieId
Pour obtenir les 5 joueurs ayant realises le plus de partie: MostPlayedPlayers
Pour obtenir les 5 ouverture les plus jouees: MostPlayedOpens
Pour obtenir les parties entre deux joueurs: GetGamesByPlayers joueur1 joueur2
Pour obtenir les adversaires de toutes les parties d'un joueur: GetPlayerOpponents joueur
Pour signaler que vous avez fini vos recherche: END

MostPlayedPlayers
[aavza50=2430, badcow=3152, Geol020=3516, karimov=3731, ALFA22=3741]

Pour obtenir toutes les parties(id) d'un joueur: AllGames nomJoueur
Pour obtenir l'elo d'un joueur: GetEloPlayer nomJoueur

Pour obtenir une partie Ã partir d'un id: GetGameById partieId
Pour obtenir les 5 joueurs ayant realises le plus de partie: MostPlayedPlayers
Pour obtenir les 5 ouverture les plus jouees: MostPlayedOpens
Pour obtenir les parties entre deux joueurs: GetGamesByPlayers joueur1 joueur2
Pour obtenir les adversaires de toutes les parties d'un joueur: GetPlayerOpponents joueur
Pour signaler que vous avez fini vos recherche: END

GetPlayerOpponents amirmahdi
[ljubisa2810, Gilmarx, welcome0001, ognjen, erez, heidar_chaloos, heidar_chaloos, Cockburn, freddyraul57, freddyraul57, freddyraul57, wiegleben, wiegleben, Spartakas, bub08, dunduk00, AndyLion, Blondes, tolia, oktoos77, formatc, A-777, homemteca, sebola, Malkavian, bernes, ARASH4000, bernes, FifthCategory, sasan_____, RONNER, Perfectos, BbIE6Y, ostap, QxV, SandroRoy, sport, ULTRASIETE, nebojsa1234, stuyck, richandi, Queenkiller, Queenkiller, carljan, TheMagBumper, Bouchard, norimyxxxo, tadjvid19, tisdag15, Zeitnot, tisdag15, waca, Wannaplayme, Wannaplayme, enki60, badcow, HomeBurger, grabek, puffetta, BIBAC, mv163, ostap, mv163, velimaxi, WINEK, porta, nikolsk, nikolsk, nikolsk, taifun, Gambito13193, norimyxxxo, TheLordOfTheFlies, zh3000, zh3000, zh3000, zh3000, gera22, gera22, gera22, gera22, gera22, zh3000, alihaleh, alihaleh, ali111, Mental, Mental, Mental, Mental, Mental, Mental, raowl, Mental, hophman, Mental, ALFA22, ALFA22, ALFA22, ALFA22, amir1381, ALFA22, shueardm, BbIE6Y, 711111]

Pour obtenir toutes les parties(id) d'un joueur: AllGames nomJoueur
Pour obtenir l'elo d'un joueur: GetEloPlayer nomJoueur
Pour obtenir une partie Ã partir d'un id: GetGameById partieId
Pour obtenir les 5 joueurs ayant realises le plus de partie: MostPlayedPlayers
Pour obtenir les 5 ouverture les plus jouees: MostPlayedOpens
Pour obtenir les parties entre deux joueurs: GetGamesByPlayers joueur1 joueur2
Pour obtenir les adversaires de toutes les parties d'un joueur: GetPlayerOpponents joueur
Pour signaler que vous avez fini vos recherche: END

GetGamesByPlayers amirmahdi Queenkiller
[Black player: amirmahdi
White player: Queenkiller
Opening: English Opening: Agincourt Defense
Result: BLACK_WON

, Black player: Queenkiller
White player: amirmahdi
Opening: Van't Kruijs Opening
Result: WHITE_WON

]

Pour obtenir toutes les parties(id) d'un joueur: AllGames nomJoueur
Pour obtenir l'elo d'un joueur: GetEloPlayer nomJoueur
Pour obtenir une partie Ã partir d'un id: GetGameById partieId
Pour obtenir les 5 joueurs ayant realises le plus de partie: MostPlayedPlayers
Pour obtenir les 5 ouverture les plus jouees: MostPlayedOpens
Pour obtenir les parties entre deux joueurs: GetGamesByPlayers joueur1 joueur2
Pour obtenir les adversaires de toutes les parties d'un joueur: GetPlayerOpponents joueur
Pour signaler que vous avez fini vos recherche: END

END

V- Conclusion

Nous avons réussi à produire ce que nous voulions avec notre application cependant elle présente quelques limites qui pourraient être des faiblesses pour une application plus large avec un fichier plus gros. La première faiblesse repose sur l'utilisation de la classe ExecutorService avec l'utilisation des threads pool. Cette classe donne une capacité illimité à la file d'attente de tâche ce qui cause un risque de surcharge de la RAM et du CPU si les tâches sont trop nombreuses. Une solution possible à ce problème est d'utiliser la classe ThreadPoolExecutor à la place de ExecutorService qui est une classe de gestion des threads de plus bas niveau qui va limiter le nombre de tâches maximum. Le deuxième problème est la taille des ConcurrentHashMap. Comme elles contiennent des objets, elles deviennent très vites très volumineuses, ce qui poserait un problème avec un fichier pgn plus lourd car cela risquerait au mieux de causer des temps d'attente très long au niveau de la sérialisation, au pire de saturer la RAM. Une solution pour parer à ce problème serait de stocker les octets du fichier où l'on peut retrouver les informations d'une partie dans la ConcurrentHashMap, plutôt que des objets, en utilisant un compteur et la fonction getBytes(). Cette solution combinée à l'utilisation des executors permettrait un traitement extrêmement rapide du fichier mais aussi de stocker beaucoup plus d'informations que l'on a dû mettre de côté pour alléger les ConcurrentHashMap comme la date, l'heure et les coups joués d'une partie et donc avoir plus de commande disponible pour répondre aux recherches des utilisateurs.

Enfin, le programme s'exécute uniquement sur Windows, car au niveau du constructeur de la classe TraitementRequete on utilise des chemin pour récupérer les fichier sérialisés, et dans ces chemins, sous Windows, les noms des répertoires et éventuel fichier sont séparés par un anti*slash «* / » (alors qu'on utilise un *slash* « \ » sous Linux). Afin de palier à ce problèmes on aurait pu définir sur quel système d'exploitation on aimerait exécuter notre projet au niveau des commandes et ensuite vérifier ce qui a été indiquer pour qu'au final exécuter le code avec les bons chemins selon l'OS.

FIN