

**ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA**

**GRADO EN INGENIERÍA INFORMÁTICA**

**Curso Académico 2022/2023**

**Trabajo Fin de Grado**

**INGENIERÍA DE DATOS CON EL FRAMEWORK DE BIG DATA SPARK Y SCALA**

**Autor**: Milagros Mouriño Ursul

**Director**: Juan Manuel Serrano Hidalgo

# Resumen

# Palabras clave

# Índice

[Resumen 2](#_Toc124408487)

[Palabras clave 2](#_Toc124408488)

[Índice 3](#_Toc124408489)

[1. Introducción 4](#_Toc124408490)

[2. Objetivos 5](#_Toc124408491)

[3. Descripción informática 6](#_Toc124408492)

[3.1. Creación del dataset de partidas 6](#_Toc124408493)

[3.1.1. Especificación 6](#_Toc124408494)

[3.1.2. Arquitectura 7](#_Toc124408495)

[3.1.3. Diseño 8](#_Toc124408496)

[3.2. Consultas con GraphFrames 9](#_Toc124408497)

[4. Experimentos 10](#_Toc124408498)

[4.1. Resultados de las consultas 10](#_Toc124408499)

[4.2. Rendimientos comparados 11](#_Toc124408500)

[5. Conclusiones 12](#_Toc124408501)

[5.1. Trabajo futuro 13](#_Toc124408502)

[6. Bibliografía 14](#_Toc124408503)

[7. Apéndices 15](#_Toc124408504)

# 1. Introducción

(2-3 páginas)

# 2. Objetivos

(1-2 páginas)

# Descripción informática

## Creación del dataset de partidas

(15-20 páginas)

Se ha tomado la decisión de elaborar un *sbt multiproyecto* dado que el programa se compone de la parte de descargas y consultas localizadas en los paquetes *download* y *queries* respectivamente destacados en la Ilustración X. La separación tuvo lugar debido a que ambos poseen distintas funcionalidades. El subproyecto del paquete *download* se encarga de la descarga de la información de las partidas de ajedrez, mientras que el de *queries* realiza consultas sobre el dataset originado a partir de dichos datos.

Interfaz de usuario gráfica

Descripción generada automáticamente

En la Figura X, se observa el *build.sbt* de la aplicación que es el fichero utilizado para la construcción del proyecto donde se indican todas las dependencias de este. Se observa que el valor root representa al proyecto raíz. Se utiliza el método *aggregate*, el cual provoca que ejecutar una tarea en el proyecto agregado también la ejecutará en los subproyectos agregados (*download* y *queries*). Se declaran las configuraciones comunes de ambos proyectos dentro del valor commonSettings donde se incluyen las dependencias comunes a ambos proyectos como la librería caseapp para el parseo de los argumentos de la línea de comandos.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

En la Imagen X, se muestra la declaración de la constante assemblySettings que sirve para la construcción del jar del subproyecto de las *queries*, el cual será necesitado para la ejecución de las consultas en AWS.

Texto

Descripción generada automáticamente

Por último, encontramos en la Figura X la declaración de las constantes download y queries donde se indican todas las configuraciones. Se utilizan las dependencias almacenadas en la constante commonSettings en ambos subproyectos. Asimismo, se indican los nombres de ambos subproyectos: Download y Queries respectivamente. En el primero, se añade el plugin de kind-projector para añadir sintaxis para las expresiones lambda, así como una serie de dependencias como la librería requests para la realización de peticiones HTTP, spray-json para el parseo de objetos JSON y la librería cats que posee múltiples métodos y tipos de datos para programación funcional. Por otro lado, en el proyecto de las queries, se añade la dependencia de spark para la ejecución de las consultas y la constante assemblySettings para la generación del jar previamente dicha.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

### 3.1.1. Especificación

### 3.1.2. Arquitectura

En la Imagen X, encontramos la jerarquía de clases del proyecto *download*. Se distinguen cuatro partes principales recogidas en los subpaquetes *main*, *io*, *manager* y *data*. El primero recoge el objeto Download y el case class ArgumentsDownload. Dentro del primer objeto, se encuentra el método *run,* el cual se ejecuta si el parseo de los argumentos de la línea de comandos a un objeto del tipo ArgumentsDownload se ha hecho correctamente. Este parseo se realiza debido a la utilización de la librería caseapp y es por ello por lo que se ha establecido la relación de herencia con CaseApp. Este objeto Download tiene dependencia con Io, ya que la información extraída de partidas y jugadores se guardará en un fichero. Asimismo, depende de Manager que es el que se encargará de realizar las peticiones HTTP y obtener la correspondiente información. En cuanto a la librería *io*, esta se encarga de gestionar la Entrada/Salida, dado que al finalizar el programa se deben escribir las partidas y jugadores a disco. Se utiliza la clase java.util.FileWriter para declarar los atributos descriptores de fichero matchesFile y playersFile. Por otra parte, se encuentra el paquete *data* que porta las clases necesarias para el parseo de los ficheros json obtenidos de Chess. Se encuentra la case class *Match* que posee información de una partida concreta cuyos atributos son white de tipo Player (jugador blanco), black de tipo Player también (jugador negro), eco de tipo String (apertura de la partida) y pgn de tipo String (contiene los movimientos de la partida). La case clase Player contiene información sobre el jugador respecto a la partida como un String con el username y otro String con result (indica si ha ganado o no). En cuanto a Profile, esta contiene información más detallada sobre el jugador un dato de tipo Long llamado player\_id que es un identificador del jugador; el username que es el nombre (tipo String), el title de tipo Option[String] dado que un jugador puede tener un título de ajedrez o no (Candidato a maestro, maestro fide, …), status de tipo String que indica si el usuario es premium o no, un String que representa el country para indicar el país del jugador, followers que indica su número de seguidores en la aplicación, is\_streamer que indica si el jugador es streamer (hace streaming en aplicaciones como Twitch) y joined de tipo Timestamp que hace referencia a la fecha en la que se registró el usuario. La case class MatchArchive contiene un atributo que es una lista de Match (partidas). Esta clase contenedora se necesitó debido al parseo. Por otro lado, nos encontramos con la case class PlayerTournament con el id como atributo de tipo String. Por último, están las case classes Round y Tournament donde la primera tiene como atributo una lista de identificadores de grupos y la segunda de rondas. El paquete manager tiene como objeto principal Manager, que dentro de su método apply contendrá el pipeline donde se parsearán las clases con el método parseJson de spray-json y se enviarán las peticiones HTTP con requests. El método saca como salida un iterador de partidas (Match) y lista de jugadores (Profile). El objeto Getters tiene cada uno de los getters necesarios para obtener los datos del pipeline. El método *getTitledPlayers* recibe como entrada un String con un título de ajedrez y devuelve un iterador de String con los nombres de los jugadores de un título dado; *getPlayerTournaments* que dado un nombre de jugador devuelve un iterador con los torneos de dicho jugador; *getRounds* devuelve un iterador de Strings con las urls de las rondas de un torneo específico; *getGroups* devuelve otro iterador de grupos dado el identificador de una ronda específica; getMatch devuelve definitivamente un iterador de partidas dado un grupo determinado; getPlayer devuelve un Option[Profile], ya que puede ser que falle la petición y, por último, el método *getPlayers* que se utiliza para transformar el Iterador de Match en uno de tuplas de *Match* y List[*Profile*] que es la información necesaria para grabar en disco y generar el dataset para las consultas. Vemos que esta clase depende tanto de Parser (debido a que se necesita parsear del json), de Fold, ya que se utiliza un método fold para atrapar la respuesta del Try del método *get* de requests y UnfoldIterator dado que el *getPlayers* se utiliza dentro del método *unfold*, una función de orden superior que permite mantener un estado S, en este caso de un conjunto que permite saber si ya se han obtenido determinados jugadores para no tener que volver a solicitar la petición HTTP. El objeto Fold contiene una implicit class TryFold que porta el método fold que consiste en un pattern matching. Por otro lado, la clase final UnfoldIterator contiene un companion object con dos implicit classes que contienen dos definiciones de unfold. El segundo de ellos es utilizado para generar el iterador de partidas y se construye a partir del primero. Este último al ser más sencillo se utiliza para la creación de un contador que da feedback acerca del nº de partidas obtenidas. El objeto Parser contiene valores (jsonFormatterProfile, jsonFormatterPlayerTournament, …) que permiten parsear la información de los json a las correspondientes case classes. El método *listJsonWriter* se definió para poder parsear una lista de objetos. Lógicamente depende de todo el paquete data con la información de las case classes y, además contiene un objeto TimestampJsonFormat que permite el parseo del tipo de datos Timestamp redefiniendo los métodos *write* y *read*.

Diagrama, Esquemático

Descripción generada automáticamente

En la Figura X, se observa la jerarquía para el subproyecto *queries*. Es análogo al subpaquete main de las descargas. Dentro del método run del objeto Queries se realizarán las correspondientes consultas y este está asociado a ArgumentQueries, cuyos atributos son los argumentos que se parsean por la línea de comandos. El usuario decide el nombre del fichero de partidas, el de jugadores y el de salida. Este último contendrá la información extraída tras realizar las consultas. La herencia de CaseApp se debe a que se utiliza dicha librería para el parseo.

Diagrama

Descripción generada automáticamente

### 3.1.3. Diseño

(irá el mayor número de páginas – se explica el código)

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Respecto al subpaquete *download*, en la Figura X se muestra el objeto Download que extiende de CaseApp para especificar a través de línea de comandos los atributos de la clase ArgumentsDownload vistos en la Imagen X, los cuales son *number* que representa el nº de partidas a obtener; *matchesFile* donde el usuario puede especificar el nombre del fichero json de las partidas y *playersFile* donde se puede hacer lo mismo, pero para el fichero de los jugadores. A la clase Io se le pasa el nombre de los ficheros de partidas y jugadores. Se ejecuta el método apply de manager que devuelve un Iterator[Match, List[Profile]]. El método *take* se utiliza para obtener el nº preciso de partidas (indicado por args.number) y no tener que procesar más de lo debido. Por otra parte, para cada una de esas partidas con dichos jugadores asociados (*foreach*) se escribe la información en los respectivos ficheros con el método *write* de la instancia io.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

En la Imagen X, se muestra la clase Io donde el constructor recibe el nombre del fichero de partidas y jugadores. Los atributos son los descriptores de fichero. Consta de dos métodos: *write* recibe una tupla Tuple2[Match, List[Profile]] y lo que hace es escribir en el correspondiente fichero su información. En cuanto a *close*, este sirve para cerrar los descriptores.

De las Figuras X-Y, se muestran las case classes del subpaquete *data* comentadas en el apartado de arquitectura necesarias para realizar el parseo de la información del json obtenido a partir de la petición HTTP.

Texto

Descripción generada automáticamente Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Interfaz de usuario gráfica, Aplicación, Word

Descripción generada automáticamente Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamenteInterfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamenteInterfaz de usuario gráfica, Texto, Aplicación, Sitio web

Descripción generada automáticamente

En cuanto al paquete manager, en la Figura X se presencia el objeto Manager cuyo método *apply* devuelve un iterador de partidas y sus jugadores. A partir del iterador se coge un número determinado de partidas desde el método *run* del objeto Download. A partir de una lista de títulos de ajedrez (Gran Maestro, Gran Maestro Femenina, Maestro Internacional, Maestra Internacional Femenina, Maestro Fide, Maestro Fide Femenina, Candidato a Maestro y Candidato a Maestro Femenina). La HOF *flatmap* primero mapea cada elemento usando una función de *map*, luego aplana el resultado (*flat*). Dentro de cada método se imprime un mensaje de feedback para el usuario para saber qué se está procesando y se ejecuta un *getter*. En primer lugar, se ejecuta el *flatmap* que utiliza el *getTitledPlayers*, el cual devolverá otro Iterador de Strings con el nombre de los jugadores que poseen títulos; *getPlayerTournaments* obtendrá el identificador de los torneos de los cuales participan los jugadores, por lo que el flatmap devuelve un Iterator[String]; con el método *getRounds* se mapearán las urls de los torneos en las rondas de dichos torneos; *getGroups* mapeará las url de las rondas en sus correspondientes urls de los grupos de las rondas; *getMatch* obtendrá las rondas de cada grupo; *unfold2* es otro método de orden superior que a partir de un conjunto que inicialmente es vacío e irá guardando los identificadores de los jugadores que ya hayan sido procesados, es decir, de los cuales ya se ha obtenido su perfil, con el método *getPlayers* se mapeará el Iterator[Match] en Iterator[Match, List[Profile]]. La única diferencia con el *flatMap* es que se va teniendo constancia del estado, es parecido a un *foldLeft*.

Texto

Descripción generada automáticamente

En la Imagen X, se muestra el objeto Getters que contendrán todos los *getters* que utiliza el objeto Manager. En primer lugar, se observa la función *getTitledPlayers* que dado un String (el título de ajedrez correspondiente) devuelve un Iterator[String]. Esto se debe a que se devuelven los nombres de los jugadores para dicho título. Con el *flatMap* del objeto Manager se aplanarán los iteradores obtenidos para todos los títulos. Se lanza una petición HTTP con el método *get* de requests y se parsea el contenido del json con el método *parseJson* y *convertTo* de la librería spray-json. Todo esto se encierra dentro de un Try que con el método fold imprimirá un mensaje (para saber que se ha producido un error al obtener la info.) y se devuelve la lista vacía, mientras que si todo ocurre adecuadamente se devuelve la lista obtenida, que con el método *iterator* se convierte en un iterador. El resto de *getters* son similares, pero parsean a distintos tipos de objetos tal y como se muestra en el método *getPlayerTournaments*.

Texto

Descripción generada automáticamente

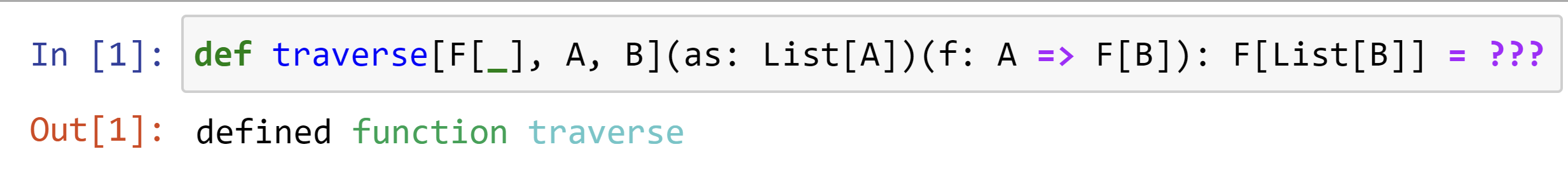
En la Imagen X, se muestra el *getter* del *unfold2*, que sirve para convertir el Iterator[Match] en Iterator[Match, List[Profile]]. En primer lugar, se observa que se invoca el método *next* del valor itCounter. Este sirve para proporcionar feedback acerca del número de partidas que ya han sido procesado y utiliza el método *unfold*. Volviendo al *getPlayers*, se utiliza la HOF *traverse* de la librería cats para que, a partir de una lista con los identificadores de los dos jugadores del *match*, se quiere obtener como resultado un conjunto actualizado (el de los identificadores de jugadores que ya han sido procesados) y el *output* con la tupla de la partida que teníamos antes y sus correspondientes jugadores.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Los métodos *foldLeft* y *foldRight* son métodos de iteración flexibles, pero nos requieren mucho trabajo para definir acumuladores y funciones combinadoras. La clase Traverse es una herramienta de nivel superior que aprovecha los Applicatives (Future, Option, …) para proporcionar una forma más eficiente de computar el resultado. En este caso, el Applicative utilizado es StateT, un tipo de datos que ofrece cats, que es sinónimo de S => F[(S, A)], es decir, es una función lambda donde dado un estado S se obtiene una tupla con el estado y el resultado aplicándole el Applicative.

En la Imagen X, se observa la signatura del método. Dada una lista de A’s y una función que transforma esas A’s en Applicatives de B’s, se retorna una lista de B’s rodeados del Applicative.



Se observa en el ejemplo de la Figura X la aplicación de la función *traverse* a una lista de enteros. El Applicative en este caso es un Option y el signo de interrogación del plugin de kind-projector sirve para currificar la función, es decir, transformar una función que utiliza múltiples argumentos en una secuencia de funciones que utilizan un único argumento tal y como se observa en la Imagen X.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

En el ejemplo de la Imagen X, se muestra que, a partir de una lista de enteros a la que se le aplica un *traverse*. Dada la signatura vista previamente, sabemos que devolverá un Option[List[Boolean]]. Se define la función que dado el entero, devolverá un Option[Boolean]. Si alguno de estos elementos no cumple la función (retorna None), el *traverse* retornará None. De lo contrario, devolverá un Some con la lista de booleanos que es lo que se observa en res17. Esto es el resultado de combinar el resultado de la función dada a todos los enteros de la lista.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

En la Imagen X, se muestra otro ejemplo con *traverse*, pero ahora utilizando el Applicative State. Se utiliza porque esta guarda información sobre un estado que es necesaria para aplicar el *unfold* (type State[S, A] = S => (S, A)). Dada la signatura, la función debería retornar algo como State[Set[Int], List[Boolean]]. Sin embargo, vemos en res30 que devuelve un IndexedStateT[Eval, Set[Int], Set[Int], List[Boolean]] siendo Eval un tipo de datos que aúna el conjunto y el resultado.

Texto

Descripción generada automáticamente

Tal y como se muestra en la Figura X, el método *run* permite pasarle un conjunto previamente (en este caso un conjunto donde se indica que ya han sido procesados el 2 y 3). Se observa que esto retorna el Eval que contiene la información importante: el conjunto S actualizado y la lista de booleanos List[Boolean], resultado de aplicar la función a los cuales se puede acceder simplemente con un *map*.

Texto

Descripción generada automáticamente

Volviendo al método *getPlayers* de la Figura X, este utiliza el Applicative StateT (type StateT[F[\_], S, A] = S => F[(S, A)]). Difiere del State en que a la tupla del conjunto y la lista resultado se le aplica otro Applicative. Dada la lista de los identificadores de los jugadores de la partida, el *run* retornará Eval[Set[String], Option[List[Profile]]] dada la signatura. Dentro de la función que se le pasa al *traverse*, se pregunta si el identicador no está dentro del conjunto (queriendo significar que ya se ha obtenido previamente la información del usuario). En ese caso, se ejecutará el getter para obtener la información del jugador de Chess. Si se ha producido un error, quiere decir ya esa partida no tiene sentido añadirlo al grafo, ya que se formaría uno inconsistente por lo que se devuelve None (y no se procesaría ya el otro usuario). En caso contrario, se devolverá un Some con el conjunto añadiendo el identificador del jugador recientemente procesado y el jugador en una tupla.

En la Figura X, se observa el objeto Fold, el cual se compone de una clase implícita denominada TryFold y contiene el método fold. Por dentro realiza un *pattern matching* del objeto Try. Si es un éxito (Success) ejecuta la función fb, de lo contrario (Failure) el método fa.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

## 3.2. Consultas con GraphFrames

(15-20 páginas)

# 4. Experimentos

## 4.1. Resultados de las consultas

(3-4 páginas)

## 4.2. Rendimientos comparados

(2 páginas)

# 5. Conclusiones

(2-3 páginas)

## 5.1. Trabajo futuro

(1 página)

# Bibliografía

(1 página)

Chambers, B., & Zaharia, M. (2018). Spark: The definitive guide: Big data processing made simple. " O'Reilly Media, Inc.".

# Apéndices