



Universidad  
Rey Juan Carlos

**ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA**

**GRADO EN INGENIERÍA INFORMÁTICA**

**Curso Académico 2023/2024**

**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

El presente trabajo pretende profundizar en Apache Spark, la cual es una de las herramientas más importantes hoy en día existentes para el procesamiento y análisis de *big data*. Permite a las organizaciones el cómputo de grandes conjuntos de datos de manera eficiente para extraer información que permita realizar una toma informada de decisiones y hacer prosperar su negocio. Este motor de computación distribuida está escrito en el lenguaje de programación funcional Scala. Se promueve el desarrollo de código siguiendo las técnicas del paradigma de programación sobre el que apoya, ya que la utilización de funciones matemáticas permite una mejor división de las tareas entre los nodos del clúster respecto a la utilización de iteraciones y estructuras de datos mutables. De igual modo, permite la reutilización de código al usar las APIs de las que consta el *framework* como GraphFrames o GraphSQL y la mantenibilidad de este permite un mantenimiento y una evolución eficientes y rentables del código base a lo largo del tiempo.

## Palabras clave

Big Data, Apache Spark, Programación funcional

# Índice

Resumen .....	2
Palabras clave .....	2
1. Introducción.....	4
2. Objetivos.....	6
3. Descripción informática.....	7
3.1. Creación del dataset de partidas .....	8
3.1.1. Especificación.....	8
3.1.2. Arquitectura.....	9
3.1.3. Diseño .....	10
3.2. Consultas con GraphFrames.....	24
3.3. Despliegue en AWS .....	27
4. Experimentos .....	30
4.1. Resultados de las consultas .....	30
4.2. Rendimientos comparados .....	33
5. Conclusiones.....	37
5.1. Trabajo futuro.....	39
6. Bibliografía .....	40

# 1. Introducción

En la era digital actual, el término *Big Data* ha pasado a ser fundamental en el vocabulario empresarial y tecnológico. Se refiere al vasto conjunto de datos, tanto estructurados como no estructurados, que son generados y recopilados por organizaciones e individuos para mejorar sus procesos y decisiones comerciales. Estos datos provienen de diversas fuentes, como redes sociales, sensores, sistemas transaccionales, dispositivos móviles, entre otros. El sistema de recomendaciones de Amazon no existiría de no ser por el procesamiento y análisis de búsquedas usuales de compradores. Estos datos suelen ser demasiado grandes y complejos para ser procesados y analizados utilizando las herramientas y técnicas tradicionales.

Para aprovechar al máximo estos enormes volúmenes de datos, es esencial contar con herramientas y técnicas especializadas en su procesamiento y análisis. En este contexto, Apache Spark emerge como un elemento clave. Este *framework* de código abierto, desarrollado inicialmente en la Universidad de California a principios de la década de 2010 y donado a la Apache Software Foundation en 2014, se ha convertido en una solución líder para el procesamiento de *Big Data*. Fue creado para el procesamiento de estos datos masivos ejecutándose desde un ordenador local hasta miles de clústeres. Se impuso frente a Apache Hadoop, el cual también permite el procesamiento y análisis por lotes de datos mediante el sistema de archivos distribuidos de Hadoop (HDFS) y el paradigma MapReduce para el cálculo a través del nodo máster y los esclavos, pero el primero es más rápido al permitir almacenar y procesar datos en memoria, mientras que el último está basado en disco al operar con los datos que residen en HDFS. Además, Spark presenta flexibilidad al poseer APIs que permiten a los desarrolladores escribir código y extraer información valiosa sobre los datos en los lenguajes de programación Java, Scala, Python y R. No solo permite el procesamiento por lotes de los datos, sino también el de flujos de datos con la librería Spark Streaming, lo cual puede ser interesante, por ejemplo, para realizar consultas sobre tweets que emergen instantáneamente mediante la API de Twitter. Asimismo, goza de librerías complementarias como SparkSQL para la realización de consultas sobre los datos; MLlib para observar patrones o tendencias en los datos mediante *machine learning* o GraphX para el procesamiento y análisis de grafos [1]. En el presente trabajo, se utiliza Apache Spark para la elaboración de consultas sobre un dataset de partidas de ajedrez obtenido a través de las API REST de Chess.com.

En el contexto de la programación funcional, un enfoque cada vez más adoptado en el desarrollo de aplicaciones distribuidas, Spark ofrece ventajas adicionales. Al utilizar funciones puras y evitar el estado mutable y efectos secundarios en lugar de programación imperativa, es más fácil dividir y distribuir el trabajo de procesamiento de datos a través de varios nodos de un clúster, lo que permite una mayor escalabilidad y rendimiento. Además, el uso de funciones puras y la evitación del estado mutable también ayuda a evitar errores y problemas de concurrencia en el procesamiento de datos distribuidos.

Para la utilización de Spark en la nube, Amazon Web Services (AWS) ofrece una solución conveniente a través de su servicio Elastic MapReduce (EMR). Amazon Web Services (AWS) es una plataforma informática en la nube ofrecida por Amazon. EMR permite ejecutar Spark en clústeres de instancias EC2, lo que proporciona la flexibilidad de escalar recursos según sea necesario y pagar solo por el uso efectivo.

## 2. Objetivos

El objetivo general del presente proyecto es **desarrollar una aplicación de análisis de datos** basada en el procesamiento distribuido utilizando el framework Apache Spark. Esto se llevará a cabo a través de la persecución de los siguientes objetivos:

- La **elaboración de un dataset** consistente en un grafo donde los nodos son jugadores y las aristas son partidas a través de la **descarga de los datos** obtenidos a través de los diferentes *endpoints* que proporciona la **API pública de Chess.com**
- La **realización de consultas utilizando el framework** para programación distribuida de **Spark** y la correspondiente **visualización de los datos** obtenidos su ejecución.
- El **despliegue de la aplicación en la nube** mediante la configuración de un clúster de ordenadores que provee el servicio web **Amazon EC2** y la **comparación del rendimiento** respecto a la ejecución en local mediante la herramienta **Spark UI**

Las consultas implementadas son:

- **Consulta 1:** Partidas en las que se ha jugado la Defensa Siciliana.
- **Consulta 2:** Jugador que tiene más *followers* en la aplicación de Chess.com
- **Consulta 3:** Partidas en las que el jugador de blancas es americano o el jugador de negras es español.
- **Consulta 4:** Jugadores de blancas registrados en la aplicación de Chess.com antes del 12 de septiembre del 2015 a las 00:00 a.m.
- **Consulta 5 y 6:** Ordenar de mayor a menor los jugadores que han disputado más partidas para blancas y negras respectivamente.
- **Consulta 7:** Identificar a los jugadores más importantes basándonos en las partidas jugadas.
- **Consulta 8:** Observar el porcentaje de resultados obtenidos en cada partida, es decir, el porcentaje de victorias, tablas y derrotas según diferentes motivos como puede ser el exceso de tiempo o jaque mate.

### 3. Descripción informática

El proyecto presenta dos partes bien diferenciadas. Por una parte, se encuentra la destinada a las descargas de los datos de la API de Chess.com para la creación del dataset y, por otra, el desarrollo de las consultas. Se ha utilizado la herramienta *sbt* para la construcción y gestión de las dependencias del proyecto. Debido a la clara diferenciación entre las dos partes de la aplicación, se ha decidido crear un sbt multiproyecto. La parte de las descargas se encuentra en el subpaquete *download* mientras que las consultas se encuentran en el subpaquete *queries* tal y como se muestra en la Ilustración 1.

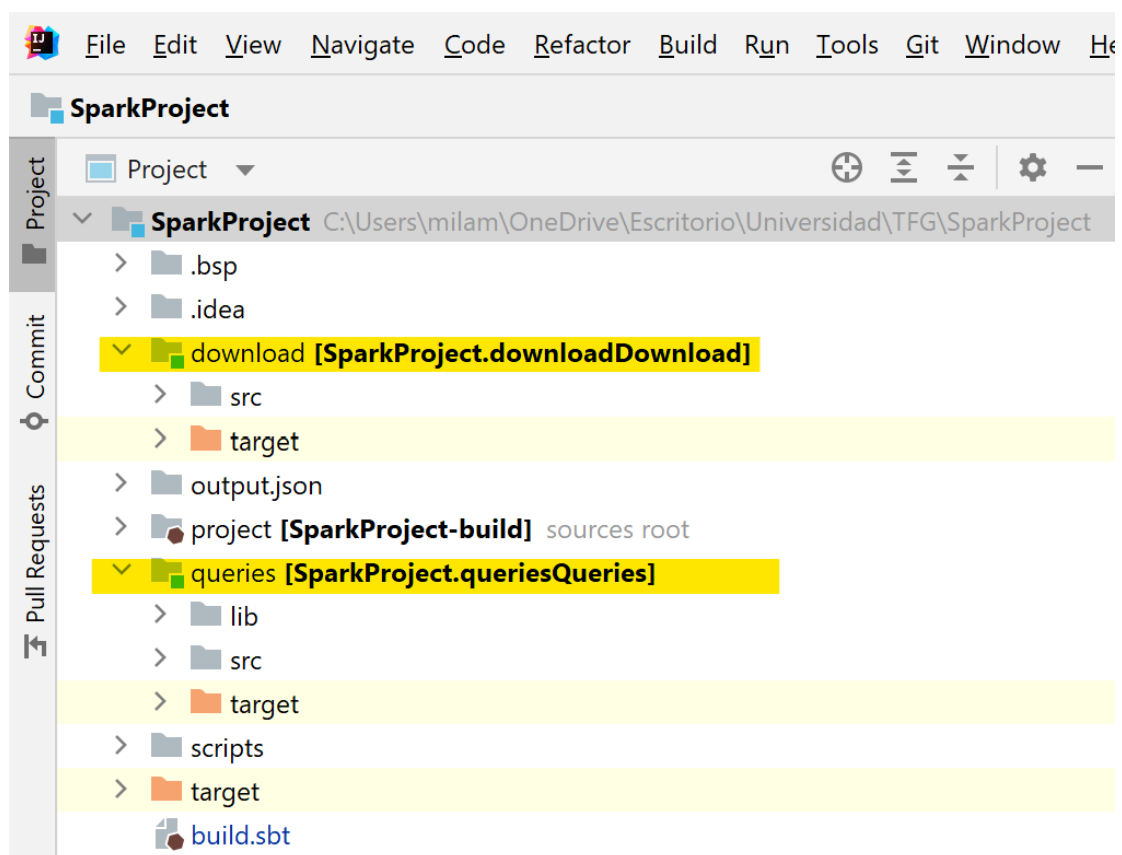


Ilustración 1: Estructura del proyecto raíz

## 3.1. Creación del dataset de partidas

### 3.1.1. Especificación

- **Entrada:** La aplicación obtiene los datos de las partidas y los jugadores a partir de archivos JSON obtenidos de diferentes *endpoints* de los que consta la API pública de Chess.com tal y como se muestra en el esquema de la Imagen 2. Utilizando la URL base `https://api.chess.com/pub/`, se pueden realizar las diversas consultas. Por ejemplo, accediendo al recurso `/titled/{title-abbrev}`, se puede descargar una lista de usuarios que poseen títulos en el ajedrez. Asimismo, la ruta `/player/{username}/tournaments` permite obtener una lista de torneos a los que ha asistido un usuario específico. Además, se pueden acceder a detalles más precisos de los torneos y sus rondas utilizando las rutas `/tournament/{url-ID}` y `/tournament/{url-ID}/{round}`, respectivamente. Para obtener información sobre los grupos de una ronda o las partidas asociadas a un grupo en particular, se pueden utilizar las rutas `/tournament/{url-ID}/{round}` y `/tournament/{url-ID}/{round}/{group}`. Por último, los detalles de un jugador en una partida pueden obtenerse a través de la ruta `/player/{username}`. En resumen, la API ofrece una amplia gama de funcionalidades para acceder y analizar datos relacionados con el ajedrez y los jugadores titulados [2].
- **Salida:** El dataset originado es un grafo en el cual los nodos son los jugadores y las aristas son las partidas obtenidas.

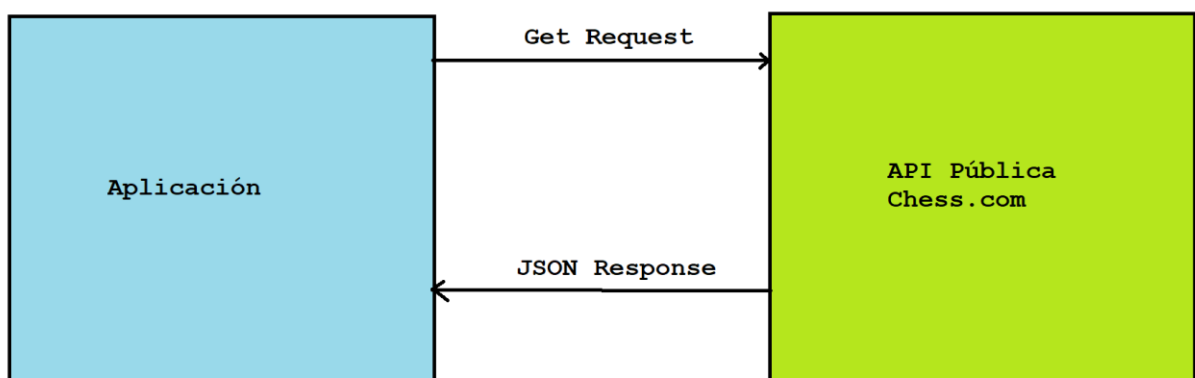


Ilustración 2: Fuente de los datos



### 3.1.2. Arquitectura

Algunas de las librerías y plugins utilizados por el proyecto son:

- **Case-app:** El parseo de los argumentos de la línea de comandos, que aporta flexibilidad a la hora de elegir el tamaño del dataset y explorar el rendimiento del programa.
- **Requests:** La realización de peticiones HTTP Get mediante la que se consigue la información para crear el grafo de salida.
- **Spray-json:** El parseo de objetos JSON obtenidos en las respuestas a las peticiones web.
- **Cats:** Posee múltiples métodos y tipos de datos para programación funcional. En concreto, se utiliza el método *traverse* de la librería para obtener a los jugadores ajedrecistas.
- **Kind-projector:** La adición de sintaxis para las expresiones lambda dentro de las funciones de orden superior.
- **FileWriter:** Escritura de la información sobre partidas y jugadores en ficheros JSON.

### 3.1.3. Diseño

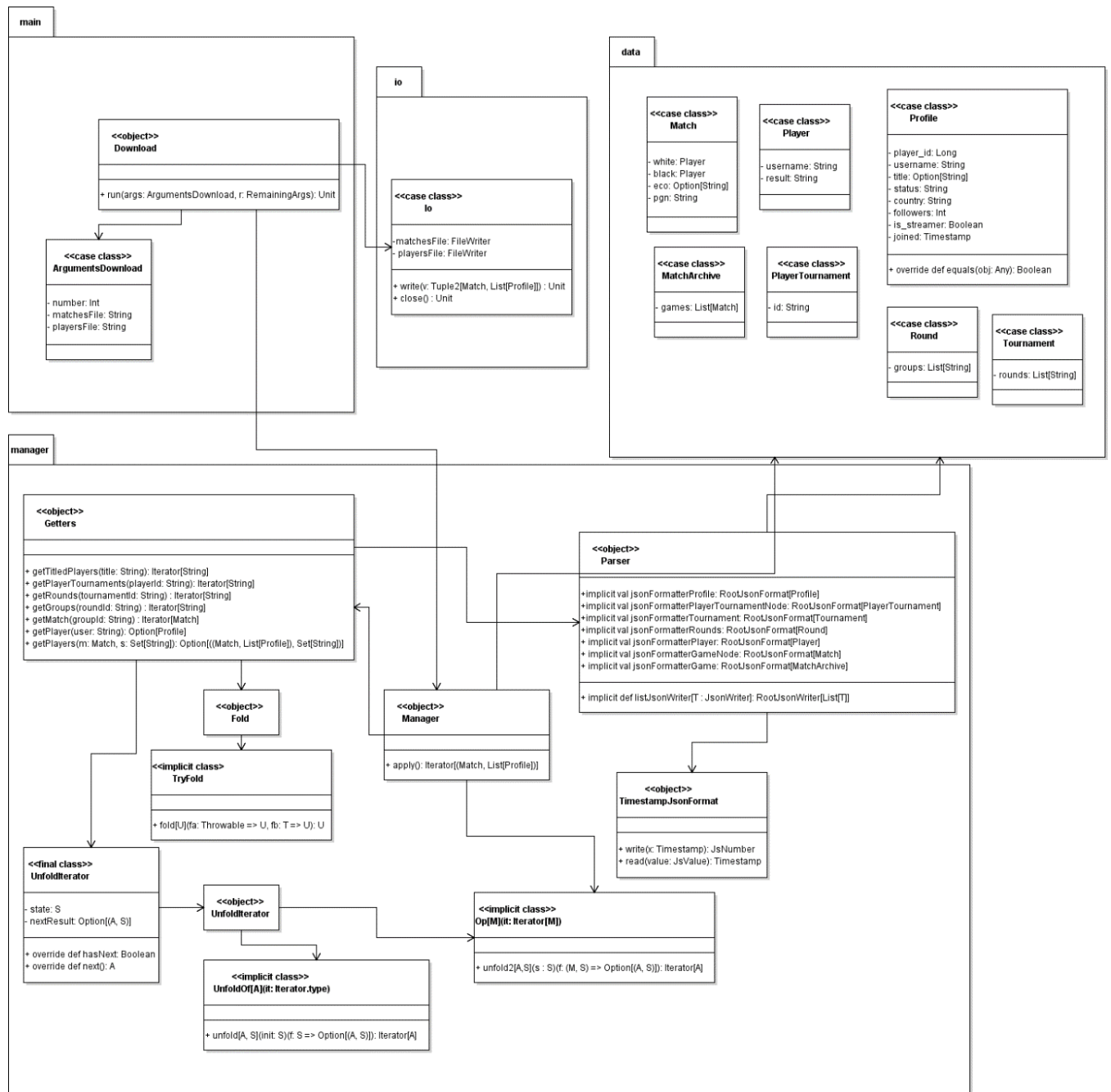


Ilustración 3: Diagrama de diseño

Tal y como se observa en la Figura 3, se distinguen cuatro partes principales dentro del paquete `download` recogidas en los subpaquetes `main`, `io`, `manager` y `data`. El paquete `main` recoge el objeto `Download` cuyo fin es el de realizar la descarga de los datos necesarios para generar el dataset de partidas y jugadores. Extiende de `CaseApp[ArgumentsDownload]` para permitir el parseo de argumentos por línea de comandos [3]. En la Imagen 4, se observa que

se puede elegir el número de partidas que se desean, el nombre del fichero JSON de partidas y el de jugadores.



```
1 package main
2
3 case class ArgumentsDownload(
4     number: Int = 100,
5     matchesFile: String = "matches.json",
6     playersFile: String = "players.json"
7 )
```

Ilustración 4: Parámetros de entrada por defecto

En la siguiente Ilustración 5, se muestran los argumentos que se pasaron para generar el dataset de partidas. Se especificó que el fichero tuviese un millón de partidas.

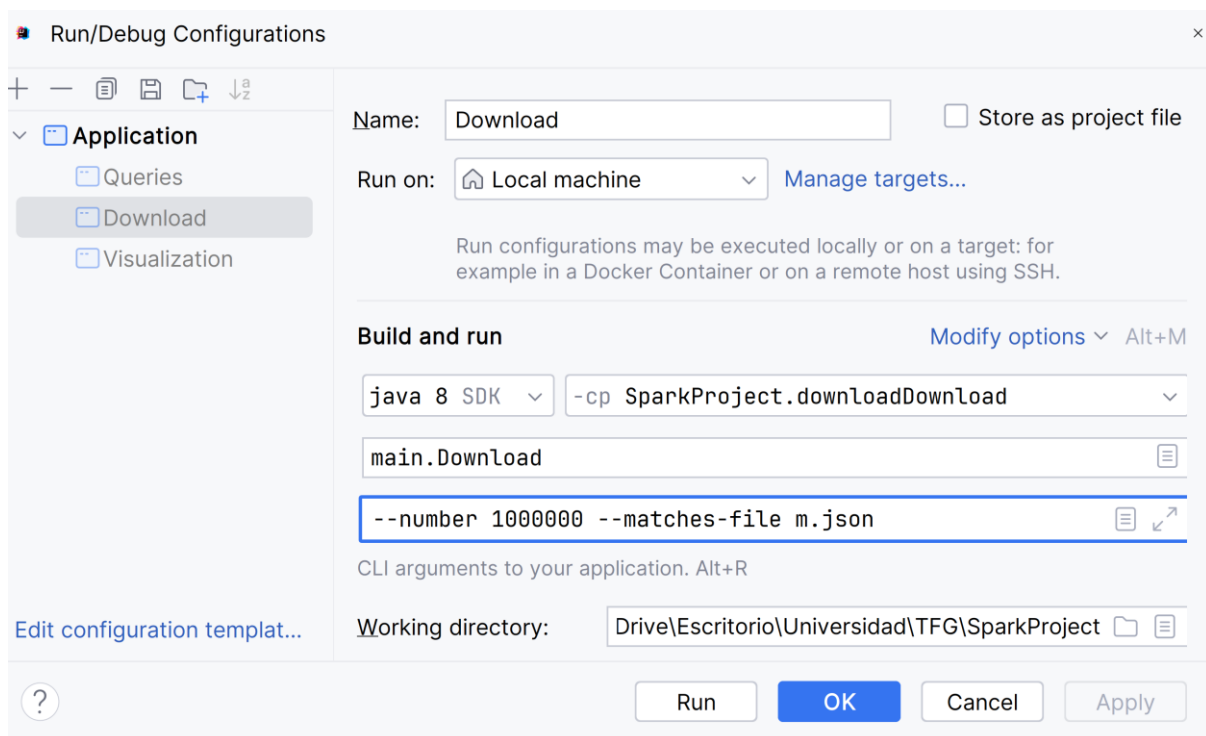


Ilustración 5: Parseo por línea de comandos desde el IDE

El paquete *io* se encarga de gestionar la entrada y salida del programa porque al finalizar se deben escribir las partidas y jugadores en disco para poder utilizar esos datos en el subproyecto de las consultas. Se utiliza la clase `java.util.FileWriter` para declarar los atributos descriptores necesarios para escribir en fichero.

Por otra parte, se encuentra el paquete *data* que contiene las clases necesarias para el parseo de los ficheros JSON obtenidos a través de la API. Contiene las case classes: *Match* que posee información de una partida concreta; *Player* que contiene información sobre el jugador respecto a la partida; *Profile*, esta contiene información más detallada sobre el jugador; *MatchArchive* que una clase contenedora necesaria para el parseo de los datos provenientes de la API y consiste en una lista de partidas; *PlayerTournament*, *Round* y *Tournament* para el registro de la información de los torneos del jugador, las rondas y los torneos respectivamente.

```

7  object Manager {
8
9  def apply(): Iterator[(Match, List[Profile])] = {
10
11    List("GM", "WGM", "IM", "WIM", "FM", "WFM", "CM", "WCM")
12      .iterator
13      .flatMap(t => {
14        println("Getting " + t + " titled players...")
15        getTitledPlayers(t)
16      })
17      .flatMap(p => {
18        println("Obtaining " + p + " tournaments... ")
19        getPlayerTournaments(p)
20      })
21      .flatMap(r => {
22        println("Obtaining from " + r + " round's info... ")
23        getRounds(r)
24      })
25      .flatMap(g => {
26        println("Getting from " + g + " group's info... ")
27        getGroups(g)
28      })
29      .flatMap(getMatch)
30      .unfold2(Set[String]())(getPlayers)

```

Ilustración 6: Pipeline de ejecución

En cuanto al paquete *manager*, el objeto principal *Manager* define el método *apply* del pipeline visto en la Imagen 6. El método tiene como salida un iterador de tuplas de partidas y lista de dos jugadores cuyos elementos se guardarán en dos ficheros JSON (uno de partidas y otro de jugadores) para generar el grafo de partidas a consultar. Se ha decidido utilizar

iteradores, ya que es una estrategia útil cuando se trabaja con grandes conjuntos de datos al procesar los elementos de la colección de una manera más eficiente en memoria, al cargar y procesar solo un elemento a la vez [4]. Para ilustrar mejor la idea que tiene esta clase, se muestra la Imagen 7 donde a partir de una lista de títulos de maestros de ajedrez (Gran Maestro, Maestro Internacional, Maestro FIDE...) se van obteniendo los diferentes datos mediante llamadas REST a los diferentes *endpoints* ofrecidos por Chess.com. En primera instancia, se construye un iterador de nombres de jugadores titulados llamando a `https://api.chess.com/pub/titled/{title-abbrev}`. A su vez, a través de las llamadas a los *endpoints* `https://api.chess.com/pub/player/{username}/tournaments`, `https://api.chess.com/pub/tournament/{url-ID}`, `https://api.chess.com/pub/tournament/{url-ID}/{round}` y `https://api.chess.com/pub/tournament/{url-ID}/{round}/{group}` se obtiene un iterador con los nombres de los torneos, otro iterador con la información de dichos torneos, un iterador con las rondas de los torneos, un iterador con los grupos de las rondas de un torneo y el iterador de partidas necesario para construir el dataset. Mediante la llamada a `https://api.chess.com/pub/player/{username}`, se obtiene información relativa al jugador de blancas y negras de una partida con lo que se genera el iterador de tuplas de partida y jugadores.

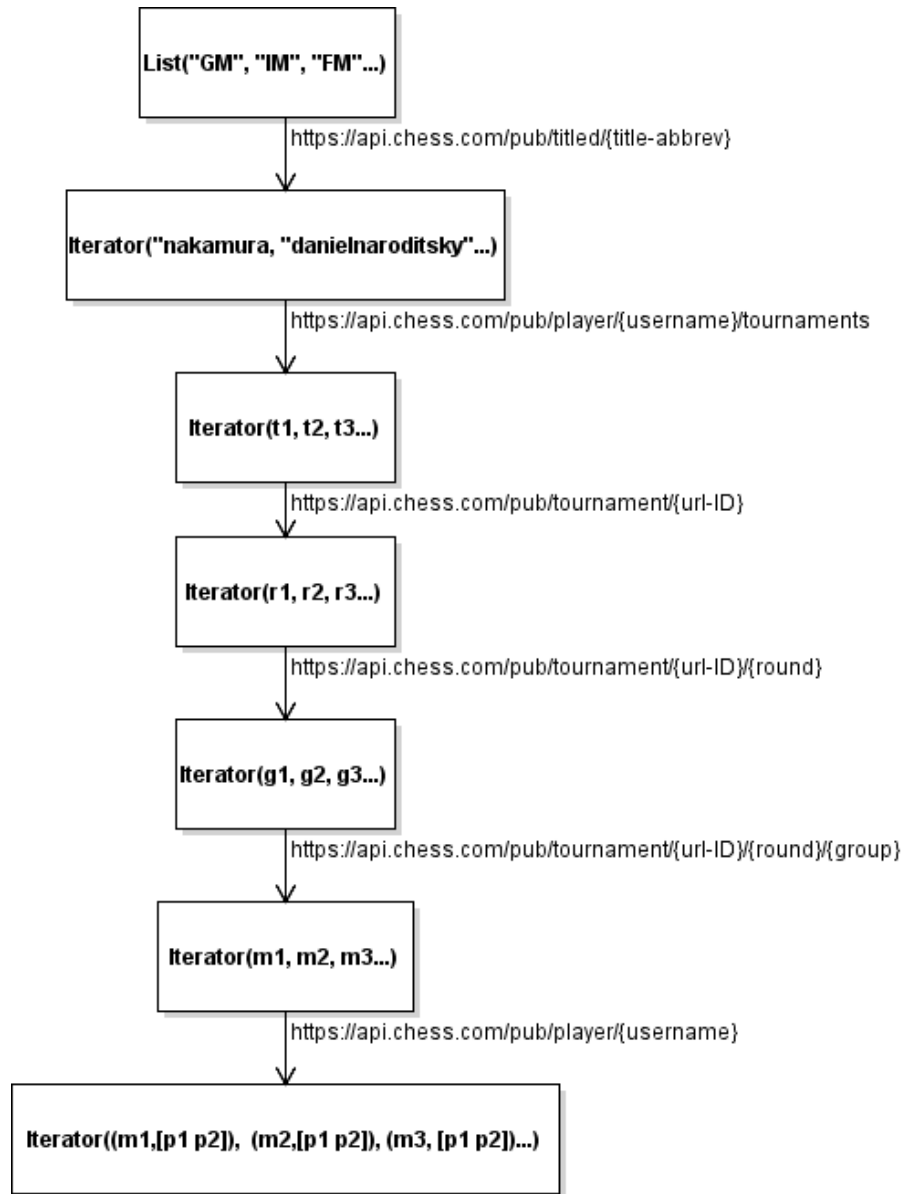


Ilustración 7: Diagrama de secuencia del pipeline para la construcción del dataset

En cuanto al objeto *Getters* tiene los *getters* necesarios para obtener los datos de los diferentes *endpoints* ofrecidos por la API de Chess.com y construir el dataset. Se utiliza el método *parseJson* de *spray-json* para serializar los objetos obtenidos de la API [5]. El método *getTitledPlayers* devuelve un iterador con los nombres de los jugadores de un título dado tal y como se muestra en la Figura 8; *getPlayerTournaments* que retorna un iterador con los torneos de dicho jugador;

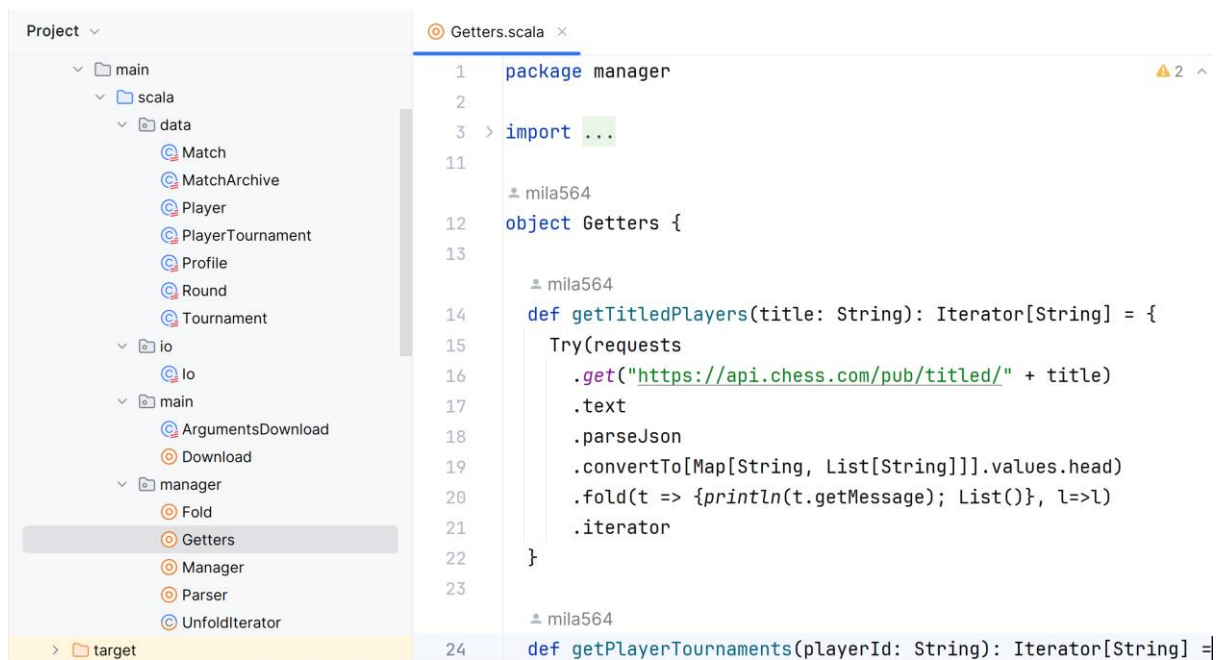


Ilustración 8: Método para obtener los jugadores titulados

*getRounds* devuelve un iterador con la url de las rondas de un torneo específico; *getGroups* devuelve otro iterador de grupos dado el identificador de una ronda específica; *getMatch* devuelve un iterador de partidas; *getPlayer* devuelve un jugador en un `Option[Profile]` por si no se puede obtener el jugador y, por último, *getPlayers* ilustrado en la Imagen 9 es el más complejo y transforma el iterador de partidas en otro de tuplas de partidas y listas de dos jugadores que es la información necesaria para construir el dataset.

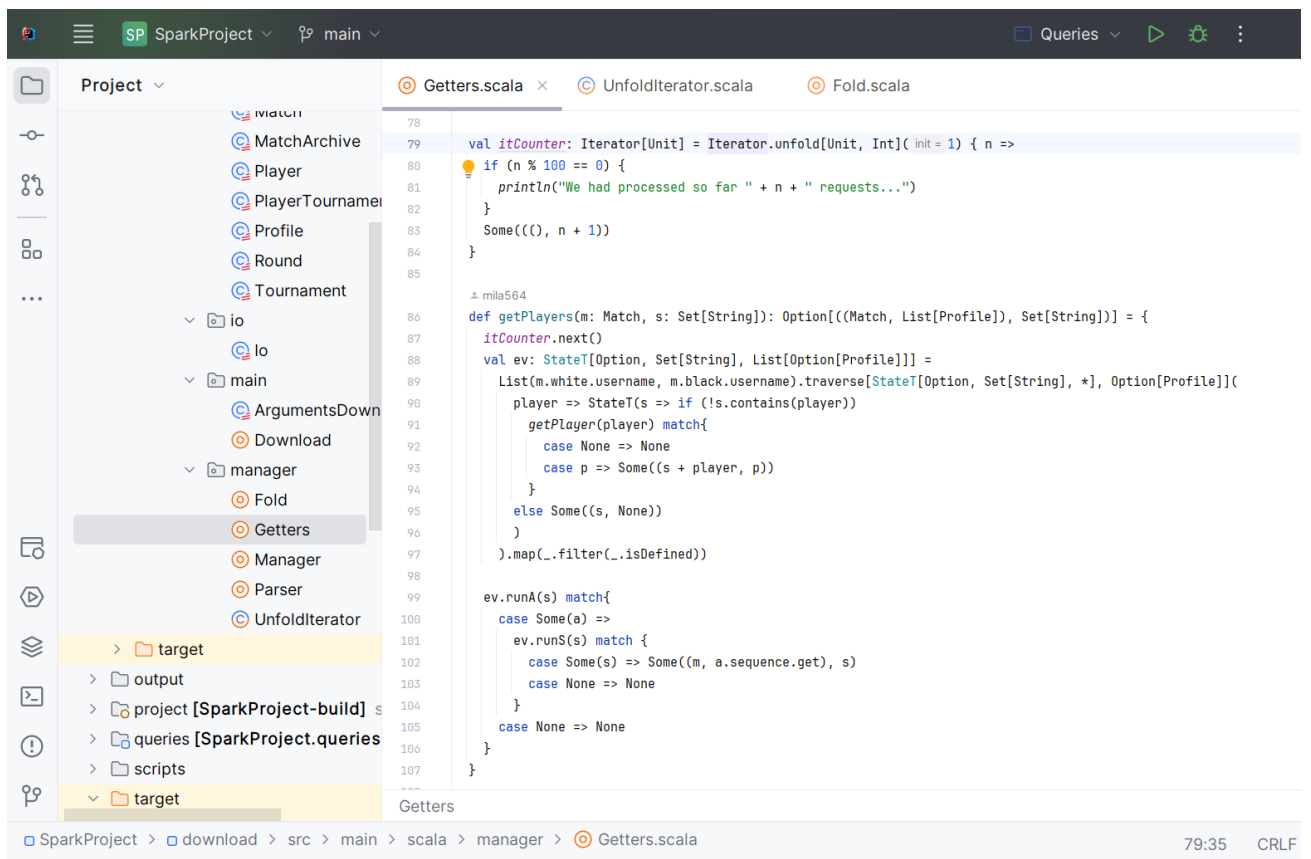


Ilustración 9: Método para obtener partidas y jugadores necesarios para generar el dataset

Se utiliza un iterador de enteros comenzando desde 1 para que en cada iteración se incremente su valor en 1 y proporcionar feedback de cuántas partidas se han procesado hasta el momento imprimiéndose un *log* cada 100.

```
def unfold[A, S](init: S)(f: (S) => Option[(A, S)]): Iterator[A]
```

Creates an Iterator that uses a function *f* to produce elements of type *A* and update an internal state of type *S*.

**A**            Type of the elements

**S**            Type of the internal state

**init**        State initial value

**f**            Computes the next element (or returns None to signal the end of the collection)

**returns**     an Iterator that produces elements using *f* until *f* returns None

**Definition Classes**    [Iterator](#) → [IterableFactory](#)

Ilustración 10: Signatura del método unfold



Este iterador se crea utilizando el método *unfold* mostrado en la Ilustración 10 implementado en la clase *UnfoldIterator*. Permite generar elementos a partir de un estado inicial y una función donde *A* es el tipo de elemento del iterador que se está generando; *S* representa el tipo del estado interno del proceso de generación del iterador; *init* es el valor inicial del estado y *f* es una función que toma el estado actual y devuelve un *Option* que contiene en una tupla el siguiente elemento del iterador y el próximo estado. Si la función devuelve *None*, el iterador se ha agotado.

Volviendo al código de la Figura 9, podemos ver que el tipo de datos que se genera es *Unit*, ya que no el iterador no se está utilizando para generar elementos, sino que se utiliza para trabajar con el estado interno *S* cuyo tipo de datos es *Int*. El valor inicial es 1 y *f* es una función que toma el entero *n* del estado actual y devuelve *Some(((), n + 1))*, donde el primer elemento es *()* que se podría asemejar a un *void* de otros lenguajes de programación, ya que el iterador no devuelve elementos y el siguiente estado es *n + 1*. En el código encontramos la ramificación para imprimir un mensaje 100 partidas procesadas.

Ese es el motivo por el cual la clase *Getters* está asociada a la clase *UnfoldIterator* mostrado en la Figura 11, ya que el método *getPlayers* utiliza ese contador para realizar un seguimiento de las peticiones generadas.

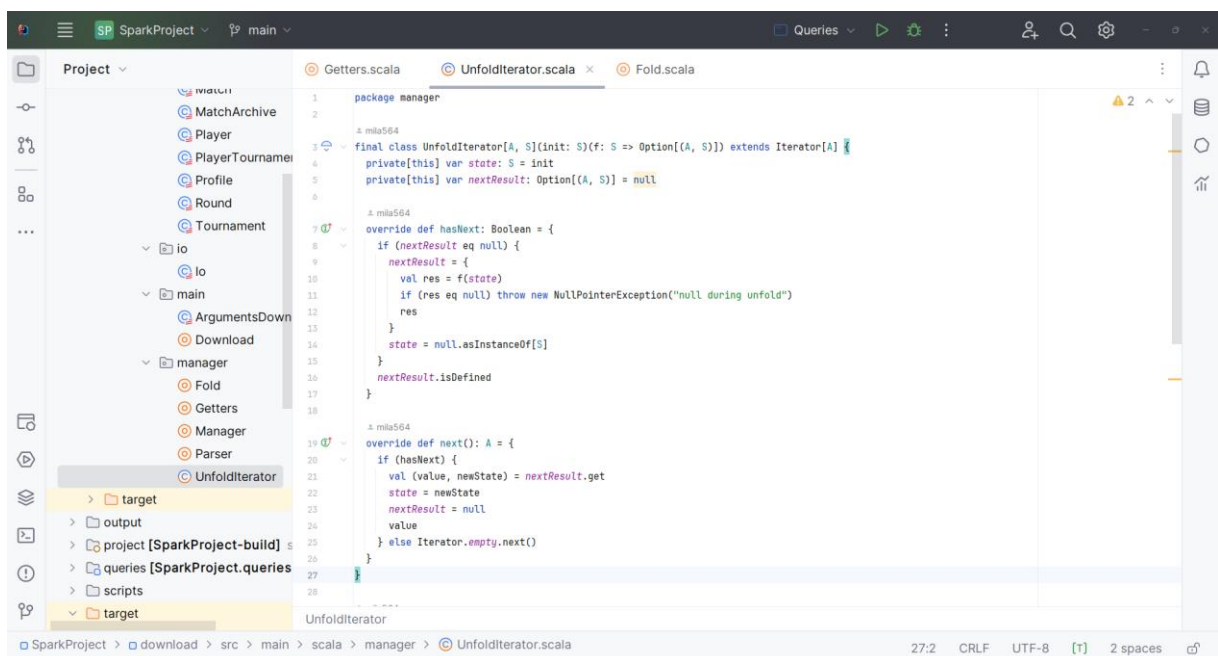


Ilustración 11: Clase *UnfoldIterator* que extiende *Iterator* al implementar los métodos *hasNext* y *next*

En concreto la clase de la Ilustración 11 extiende la interfaz *Iterator[A]* que permite definir el método *unfold* en el correspondiente objeto del *UnfoldIterator* tal y como se muestra en el

diagrama de clases de la Figura 3 con lo que se generan elementos de tipo  $A$ . Tiene un constructor que recibe un estado inicial y una función  $f$  que toma el estado actual y devuelve un `Option` de una tupla que contiene el próximo elemento y el nuevo estado. Tiene dos variables privadas, una para almacenar el estado actual y otra para almacenar el próximo resultado generado. El método `hasNext` verifica si hay más elementos en el iterador. Si no hay siguiente resultado, calcula el siguiente utilizando la función y actualiza el estado. Retorna `true` si hay un próximo resultado y `false` en caso contrario. El método `next` devuelve el próximo elemento del iterador. Si hay un próximo resultado, extrae el valor, actualiza el estado y reinicia `nextResult`. Si no hay próximo resultado, lanza una excepción o devuelve el próximo elemento de un iterador vacío. El objeto `UnfoldIterator` observado en la Ilustración 12 contiene dos clases implícitas denominadas `UnfoldOf` y `Op`, las cuales proporcionan métodos adicionales para trabajar con iteradores. `UnfoldOf` extiende la clase `Iterator` y proporciona un método `unfold` explicado anteriormente para crear un `UnfoldIterator` a partir de un estado inicial y una función.

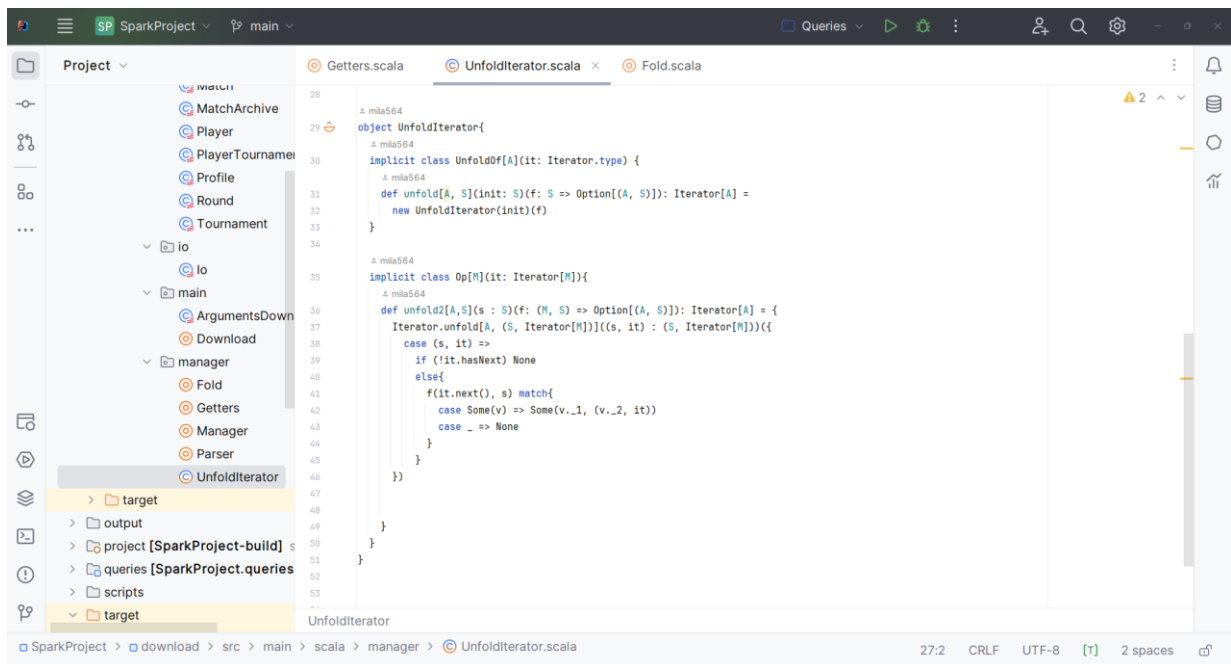


Ilustración 12: Objeto `UnfoldIterator` en el cual se definen los métodos `unfold`

`Op` proporciona el método `unfold2` de la ilustración anterior que es una forma de implementar el `unfold` mencionado anteriormente con algunas variaciones. Se utiliza en la línea 30 del método `apply` de `Manager` de la Imagen 6. Al igual que el `unfold`, recibe como parámetros un estado inicial  $S$  y una función que permite generar el siguiente elemento del iterador y actualizar al estado siguiente. No obstante, cambia el tipo de datos de la función respecto al

*unfold*. Ahora en lugar de ser una función que recibe un  $S$  y devuelve un  $\text{Option}[(A,S)]$  tiene como parámetro de entrada una tupla  $(M,S)$  y retorna un  $\text{Option}[(A,S)]$ . Se ha definido así para que los tipos coincidan con los del método *getPlayers* de la Figura 9. Como se ve en la cabecera de la función de la imagen, recibe un parámetro  $m$  de tipo *Match* y un  $s$  de tipo  $\text{Set[String]}$  y devuelve  $\text{Option}[(\text{Match}, \text{List[Profile]}, \text{Set[String]})]$ . Dentro de esta función primero se invoca al método *next* de *itCounter* para incrementar el estado del iterador contador. La parte principal de la función implica el uso del método *traverse* de la librería *cats* que es un método que se aplica en este caso a una lista de dos elementos que constituyen el jugador de blancas y el de negras de la partida del parámetro  $m$  de entrada [6]. Se aplica la función a cada elemento del contenedor y se acumulan los resultados en un nuevo contenedor. La signature del método es la siguiente:

```
def traverse[G[_]: Applicative, A, B](fa: F[A])(f: A => G[B]): G[F[B]]
```

$G[_]: \text{Applicative}$ : Esta sintaxis indica que  $G$  debe tener una instancia de la clase de tipo *Applicative*. En términos sencillos, un *Applicative* es un tipo de dato que soporta las operaciones *pure* y *ap*. *Pure* permite en el caso del primero envolver el valor en el constructor de tipos; para *Option* podría ser *Some(\_)* y *ap* es responsable de aplicar una función encapsulada en un contexto a un valor encapsulado en el mismo contexto, es decir, que se puedan aplicar funciones como *Some(3) + Some(4)* igual a *Some(7)*

$fa: F[A]$ : Este es el tipo de dato al cual se le va a aplicar el *traverse* (por ejemplo, una lista, opción, etc.) que contiene valores de tipo  $A$  que se desean convertir o mapear.

$f: A \Rightarrow G[B]$ : Es la función que se aplicará a cada elemento  $A$  devolviéndose un valor de tipo  $F[B]$

El tipo de retorno  $G[F[B]]$  indica que la operación *traverse* acumula los resultados en el mismo  $F$  de entrada.

```

Out[1]: import $ivy.$

In [3]: import cats.implicit._, cats.data._

Out[3]: import cats.implicit._, cats.data._

In [4]: List(2,4,6,8).traverse[Option, Int](
        n =>
          if (n % 2 == 0) Some(n)
          else None
        )

Out[4]: res3: Option[List[Int]] = Some(List(2, 4, 6, 8))

In [5]: List(2,4,6,8).traverse[Option, String](
        n =>
          if (n % 2 == 0) Some(n.toString)
          else None
        )

Out[5]: res4: Option[List[String]] = Some(List("2", "4", "6", "8"))

In [6]: List(2,4,6,8,999999).traverse[Option, String](
        n =>
          if (n % 2 == 0) Some(n.toString)
          else None
        )

Out[6]: res5: Option[List[String]] = None

```

Ilustración 13: Aplicación de la HOF *traverse*

En la Imagen 13, se observan usos sencillos de la función *traverse* en un notebook de Jupyter para mayor legibilidad de esta.

La lista original contiene números pares (2, 4, 6, 8). La función proporcionada al *traverse* convierte cada número par en un *Some(n)* y cada número impar en *None*. La firma del *traverse* especifica que el contexto monádico es *Option* y el tipo objetivo es *Int*. La salida será *Some(List(2, 4, 6, 8))*, ya que todos los números en la lista original son pares. El segundo código es similar al primero, pero en lugar de convertir los números pares en *Some(n)*, convierte cada número par en su representación de cadena utilizando el método *toString*. Los números impares se convierten en *None*. La firma del *traverse* especifica que el contexto monádico es *Option* y el tipo objetivo es *String*. La salida será *Some(List("2", "4", "6", "8"))*. Este código es similar al segundo, pero con la adición de un número impar en la lista original (999999). La función de *traverse* convierte los números pares en sus representaciones de cadena y omite el número impar. La firma del *traverse* especifica que el contexto monádico es *Option* y el tipo objetivo es *String*. La salida será *None*, ya que hay un número impar en la lista original. En definitiva, realmente estamos mapeando los elementos de entrada de una lista a otra lista de salida y se combinan en el aplicativo. En este caso, dada la lista de enteros

de entrada se mapeó cada uno a una lista de `Option[Int]` en el primer caso (cada uno sería un `Some(n)` si  $n$  es par o `None` en caso contrario) y luego se combina todo ello en el aplicativo `Option` (la salida de la función *traverse* es el aplicativo sobre la lista o la estructura que contiene los datos de entrada), en este caso si todos los elementos eran pares se obtenía `Some(List(2,4,6,8))`.



```

def getPlayers(m: Match, s: Set[String]): Option[(Match, List[Profile]), Set[String]] = {
  itCounter.next()
  val ev: StateT[Option, Set[String], List[Option[Profile]]] =
    List(m.white.username, m.black.username)
      .traverse[StateT[Option, Set[String], *], Option[Profile]](
        player => StateT(s => if (!s.contains(player))
          getPlayer(player) match {
            case None => None
            case p => Some((s + player, p))
          }
        else Some((s, None))
      )
    ).map(_._filter(_.isDefined))

  ev.runA(s) match {
    case Some(a) =>

```

rs > getPlayers(...)

Ilustración 14: Método para obtener las tuplas de partidas y listas de dos jugadores dadas las partidas

Volviendo al código del método *getPlayers* visto en la Imagen 14, recordemos que en el *traverse* se tienen que especificar el tipo de datos del Applicative y elemento de salida. En este caso, observamos en la definición de la función que el Applicative es la mónada `StateT`. En programación funcional, se evitan cambios directos en variables (se busca la inmutabilidad). En lugar de modificar variables, se crean nuevas versiones o estados. `StateT` nos proporciona una forma de trabajar con estados o mutabilidad de manera funcional [7]. En concreto, para nuestro programa nos interesa tener un conjunto de cadenas, el `Set[String]` anteriormente mencionado, para guardar los nombres de los *players* que ya han sido procesados para no tener que volver a realizar la petición HTTP y obtener sus datos. Su signatura es la siguiente:

```
final case class StateT[F[_], S, A](runS: S => F[(S, A)])
```

- `F[_]`: Representa el tipo de la mónada subyacente. Por ejemplo, puede ser `Option`, `List`, `Future`, etc.
- `S`: Es el tipo del estado que la mónada `StateT` va a manipular

- A: Es el tipo del valor que se produce dentro de la mónada StateT.

En el ejemplo de la Figura 15, se observa que para el estado inicial 5, el siguiente estado es el 6 (se obtiene la mutabilidad que se deseaba y actualización de un estado) y el elemento generado realizando un casting del entero a String.

```
In [7]: val stateT: StateT[Option, Int, String] = StateT(s => Some((s + 1, s.toString)))
Out[7]: stateT: StateT[Option, Int, String] = cats.data.IndexedStateT@7073a5e7

In [8]: val result: Option[(Int, String)] = stateT.run(5) // Ejecutar con estado inicial 5
Out[8]: result: Option[(Int, String)] = Some((6, "5"))
```

Respecto al tipo de datos de salida, este es un Option[Profile], ya que por cada jugador blanco y negro se va a querer obtener información sobre su perfil en la case class “Profile” (es un Option porque puede que falle la petición HTTP). Como hemos dicho antes, los resultados de los elementos de la lista se mapean de acuerdo a la función lamda que le estamos pasando al *traverse* de la lista de String a la lista de Option[Profile] y luego los resultados “se combinan” en el StateT[List[Option[Profile]]]. En cuanto a la función que se pasa como parámetro al *traverse*, nos encontramos con que dado el “*player*” (cadena de texto), la función va a devolver el correspondiente aplicativo definido en la signatura (StateT). El StateT viene definido por la función lamda vista anteriormente. Tiene como parámetro de entrada el estado *s* definido el conjunto con los nombres de los jugadores y de salida un Option que contiene una tupla con el estado siguiente actualizado añadiendo al estado o conjunto si se ha procesado un nuevo jugador con el método *getPlayer* y el jugador *p* correspondiente en un Some. Si la petición falla devuelve un None y no se añaden tanto el jugador de blancas como negras dentro de la tupla de partidas y listas de perfiles de jugador del pipeline del *apply* del manager. Ejecutando el método *runA* sobre el StateT pasándole como parámetro el estado *s* (el conjunto con los nombres de los jugadores ya procesados) se obtendrá el Option con la tupla donde el valor 1 tiene el conjunto actualizado que se deberá tener en cuenta cuando se siga consumiendo el iterador la lista de perfiles para esos los dos jugadores dados.

También está asociada a *Parser*, ya que los *getters* tienen que convertir el JSON devuelto por el endpoint en una case class del paquete *data*; *Fold*, al utilizarse el método *fold* visto en la Figura 16. Este consiste en un *pattern matching* para controlar posibles errores al realizar la

petición HTTP a los *endpoints* de la API de Chess.com.

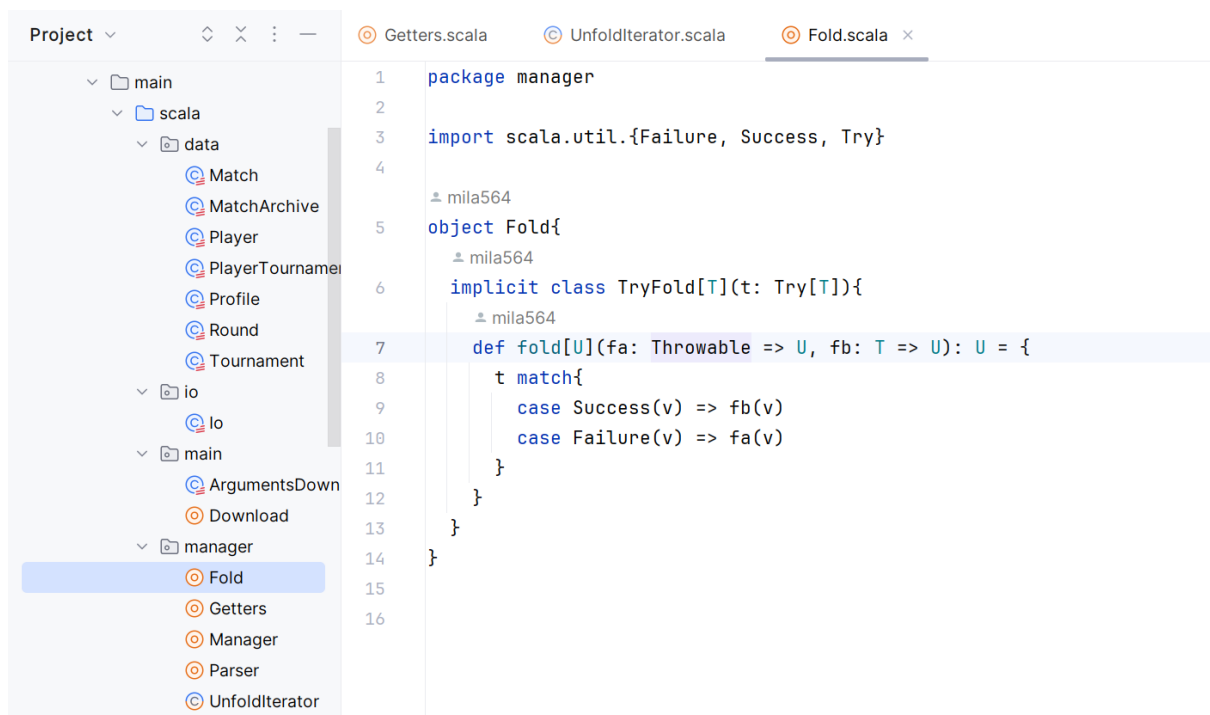


Ilustración 15: Objeto Fold con la definición del HOF fold

Cabe señalar que el objeto *Fold* contiene una clase implícita denominada *TryFold* que define el método *fold*, el cual consiste en un *pattern* donde si se obtiene un éxito se ejecuta una función y, en caso contrario, otra.

El método *listJsonWriter* se definió para poder parsear una lista de objetos y un objeto denominado *TimestampJsonFormat* para el parseo del tipo de datos *Timestamp* para fechas.

## 3.2. Consultas con GraphFrames

El objeto *Queries* contiene las consultas a realizar sobre el dominio ajedrecístico. Se introducen los argumentos por la línea de comandos especificando el nombre de los ficheros de entrada JSON obtenidos en la descarga (el fichero de partidas y jugadores) y el archivo de salida sobre el que se escriben los resultados de la ejecución de las consultas. Utilizando la función *read* del paquete *org.apache.spark.sql*, se parsea la información del JSON en su correspondiente *Dataframe* de Spark. En la Ilustración 16, se especifica el esquema del *Dataframe* para los jugadores donde se hallan columnas con información sobre el identificador, el nombre de usuario, su título si posee, su estado, su país, sus seguidores, si es streamer y la fecha de registro.

The screenshot shows a Scala IDE with a file named `Queries.scala`. The code defines a `StructType` schema for a DataFrame of players. The schema includes fields for `player_id`, `username`, `title`, `status`, `country`, `followers`, `is_streamer`, and `joined`. Below the code, a preview of the DataFrame is shown, displaying the first 10 rows of data.

```
46 val schemaProfiles = StructType(Array(  
47   StructField("player_id", LongType, nullable = false),  
48   StructField("username", StringType, nullable = false),  
49   StructField("title", StringType, nullable = true),  
50   StructField("status", StringType, nullable = false),  
51   StructField("country", StringType, nullable = false),  
52   StructField("followers", LongType, nullable = false),  
53   StructField("is_streamer", BooleanType, nullable = false),  
54   StructField("joined", TimestampType, nullable = false)  
55 ))
```

player_id	username	title	status	country	followers	is_streamer	joined
8541612	soccerhero08	null	closed	IN	0	false	2012-08-14 12:12:50
16097422	norik966	null	basic	AM	0	false	2014-02-17 17:11:20
18800602	123lt	GM	premium	CN	99	false	2014-09-07 05:09:21
16263800	robercik512	null	basic	PL	0	false	2014-03-01 08:40:01
10322550	alexander_donchenko	GM	premium	DE	490	false	2013-01-06 19:45:07
61881326	mikasinski	FM	premium	RS	351	false	2019-06-03 23:41:10
39421944	manukyan_artak	NM	premium	AM	152	false	2017-10-13 20:58:13
39436542	dubs123	null	premium	IE	58	false	2017-10-14 10:52:15
13228102	talstactician	NM	premium	XE	25	false	2013-08-21 15:44:26
156418877	peraprot2	null	basic	RS	5	false	2021-09-30 13:09:07

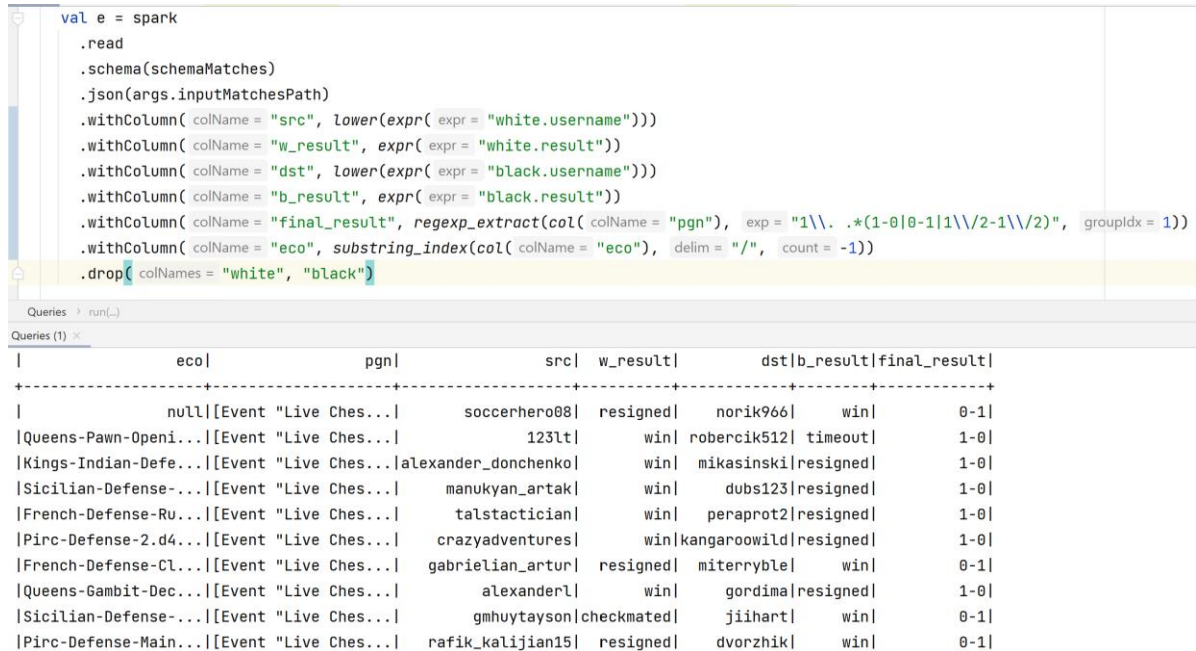
only showing top 10 rows

Ilustración 16: *Dataframe* de los jugadores

Por otra parte, en la Figura 17, se observa la estructura del *Dataframe* de las partidas tras aplicar una serie de transformaciones. El campo *src* simboliza al jugador de blancas y *dst* al de negras. Este renombramiento es necesario para la construcción de la estructura de datos de



tipo *Graphframe* [8]. Además, se registra el *pgn* que es el formato en el que se guarda la partida en forma de String, así como el resultado obtenido por las blancas, las negras, el final y la apertura jugada en la columna *eco*.



```

val e = spark
  .read
  .schema(schemaMatches)
  .json(args.inputMatchesPath)
  .withColumn( colName = "src", lower(expr( expr = "white.username")))
  .withColumn( colName = "w_result", expr( expr = "white.result"))
  .withColumn( colName = "dst", lower(expr( expr = "black.username")))
  .withColumn( colName = "b_result", expr( expr = "black.result"))
  .withColumn( colName = "final_result", regexp_extract(col( colName = "pgn"), exp = "1\\..*(1-0|0-1|1\\//2-1\\//2)", groupIdx = 1))
  .withColumn( colName = "eco", substring_index(col( colName = "eco"), delim = "/", count = -1))
  .drop( colNames = "white", "black")

```

	eco	pgn	src	w_result	dst	b_result	final_result
	null	[Event "Live Ches...]	soccerhero08	resigned	norik966	win	0-1
	Queens-Pawn-Openi...	[Event "Live Ches...]	123lt	win	robercik512	timeout	1-0
	Kings-Indian-Defe...	[Event "Live Ches...]	alexander_donchenko	win	mikasinski	resigned	1-0
	Sicilian-Defense-...	[Event "Live Ches...]	manukyan_artak	win	dubs123	resigned	1-0
	French-Defense-Ru...	[Event "Live Ches...]	talstactician	win	peraprot2	resigned	1-0
	Pirc-Defense-2.d4...	[Event "Live Ches...]	crazyadventures	win	kangaroowild	resigned	1-0
	French-Defense-Cl...	[Event "Live Ches...]	gabrielian_artur	resigned	miterryble	win	0-1
	Queens-Gambit-Dec...	[Event "Live Ches...]	alexanderl	win	gordima	resigned	1-0
	Sicilian-Defense-...	[Event "Live Ches...]	gmhuytayson	checkmated	jiihart	win	0-1
	Pirc-Defense-Main...	[Event "Live Ches...]	rafik_kalijian15	resigned	dvorzshik	win	0-1

Ilustración 17: Dataframe de las partidas

Una vez obtenidos los *Dataframes* de los jugadores y las partidas, se crea un objeto *GraphFrame*. La idea es crear un *dataset* que se corresponda con un grafo donde los vértices son los ajedrecistas y las aristas son las partidas.

Como se ilustra en la Figura 18, se ha desarrollado una consulta explotando las funciones de agregación que ofrece la librería de SparkSql con el objetivo de observar el porcentaje de resultados obtenidos en cada partida, es decir, el porcentaje de victorias blancas, tablas, victorias por exceso de tiempo entre otras opciones. En primer lugar, se invoca el método *groupBy* para agrupar en función de los resultados obtenidos para el jugador de blancas y negras correspondientemente (*w\_result* y *b\_result*). Después de agrupar, se aplica la función de agregación que en este caso es un *count* para almacenar el número para cada posible combinación. En la segunda parte de la consulta, se observa la creación de una nueva columna denominada *percentage* que se calcula dividiendo por cada grupo su correspondiente valor de la columna *count* entre la suma de todos los valores obtenidos. Mediante *coalesce* se especifica el número de particiones y se escribe en disco.

```

val results =
  g
    .edges
    .groupBy( col1 = "w_result", cols = "b_result")
    .count()
    .orderBy(desc( columnName = "count"))

results
  .withColumn( colName = "percentage",
    col( colName = "count" ) /
    lit(results.select(sum( columnName = "count")).collect()(0).getLong(0)))
  .coalesce( numPartitions = 1)
  .write
  .mode(SaveMode.Append)
  .json(args.outputPath)

```

: Dataset[Row]  
 : sql.DataFrame  
 : Dataset[Row]  
 : DataFrameWriter[Row]  
 : DataFrameWriter[Row]  
 : Unit

*Ilustración 18: Consulta de los porcentajes de las diferentes combinaciones de resultados en una partida*

Por otra parte, en la Imagen 19, se muestra una consulta que consiste en averiguar las partidas en las que se ha jugado la defensa siciliana. Asimismo, se hace el correspondiente *coalesce* y se registra el resultado de la consulta.

```

g
  .edges
  .where(col( colName = "eco").contains("Sicilian-Defense"))
  .coalesce( numPartitions = 1)
  .write
  .mode(SaveMode.Overwrite)
  .json(args.outputPath)

```

: GraphFrame  
 : sql.DataFrame  
 : Dataset[Row]  
 : Dataset[Row]  
 : DataFrameWriter[Row]  
 : DataFrameWriter[Row]  
 : Unit

*Ilustración 19: Consulta de las partidas de la defensa Siciliana*

### 3.3. Despliegue en AWS

En primer lugar, para ejecutar la aplicación en EMR, se ha instalado el cliente AWS en local y confirmamos que está correctamente instalada con el comando `aws --version`. Nos registramos en la plataforma de AWS Learning y por cada sesión renovamos las credenciales del apartado AWS CLI en el archivo en local `./aws/credentials` tal y como se observa en la Figura 20.

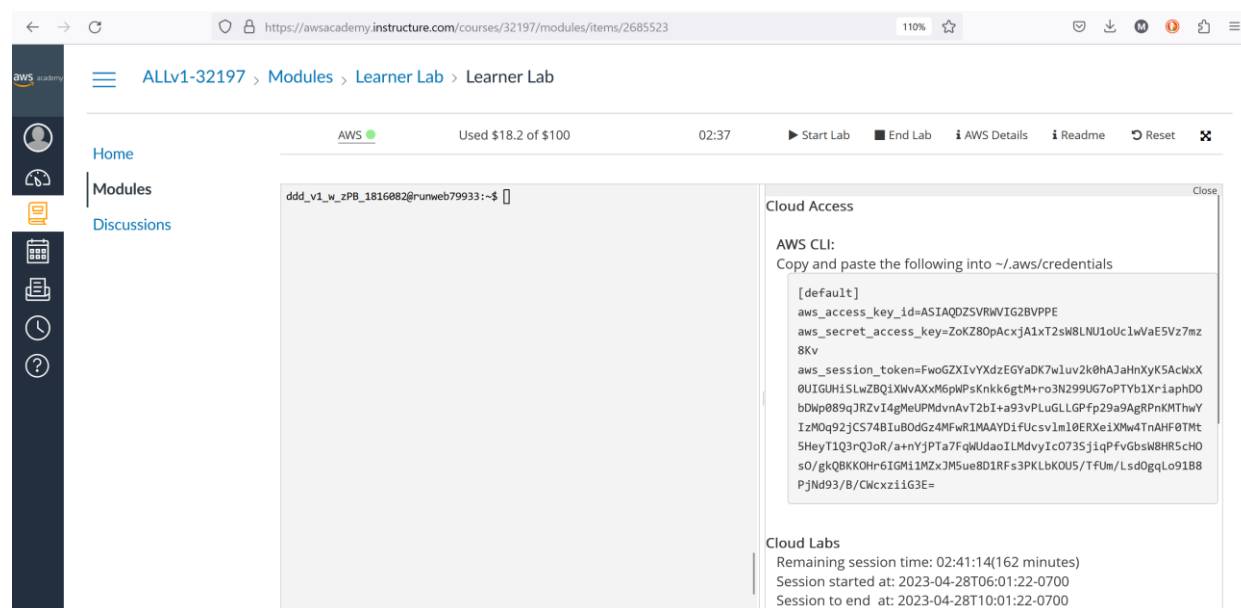


Ilustración 20: Renovación de las credenciales

En el apartado de AWS SSO, descargamos la URL que nos permite acceder a los diferentes servicios de nuestra cuenta de AWS (S3, EC2, EMR, ...). En primera instancia, creamos el *bucket* en S3 que nos permitirá almacenar ficheros de las partidas y los jugadores obtenidos en el programa Download, así como el JAR que contiene el programa con las queries a ser ejecutado en el clúster. Para ello se utiliza el comando `aws s3://<my-bucket> --region us-east-1 --endpoint-url https://s3.us-east-1.amazonaws.com`. En este caso, el nombre de mi bucket es “tfg-chess-milagros”. A continuación, accedemos a S3 mediante el link obtenido en AWS SSO y cargamos los tres archivos anteriormente mencionados que se muestran en la Imagen 21.

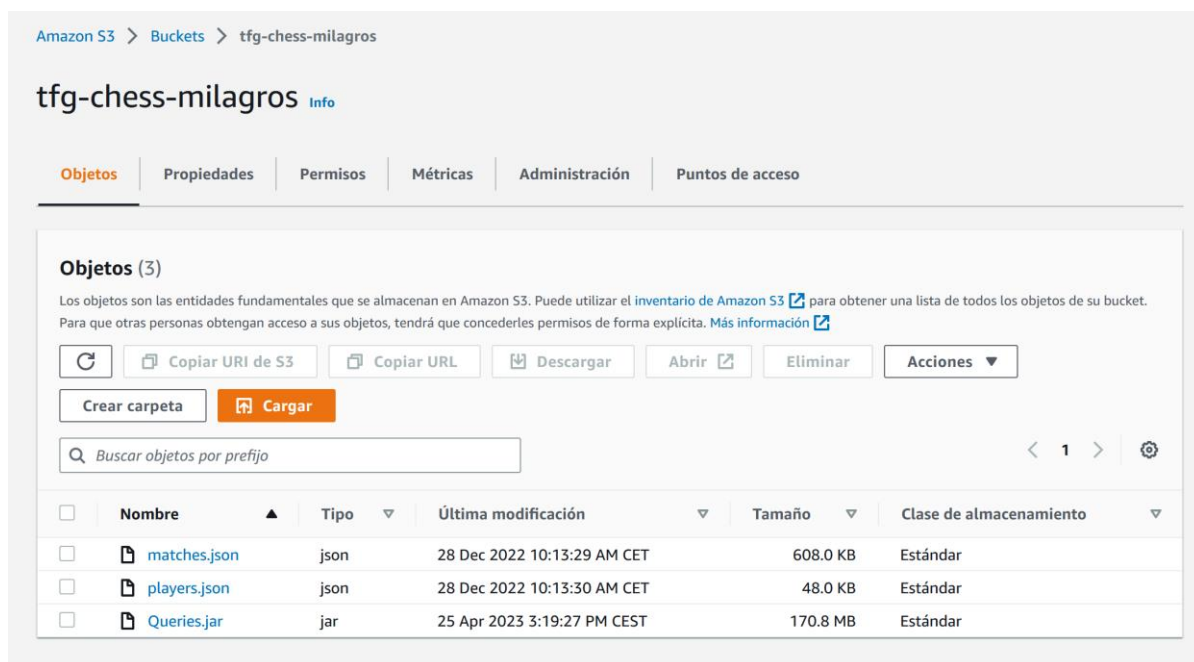


Ilustración 21: Amazon S3

En la consola de Amazon EMR, la cual se ilustra en la Figura 22, se hace clic en “Crear clúster” y se selecciona la correspondiente configuración. En este caso como la aplicación realiza consultas con Spark 2.4.3 se usa EMR-5.26.0.

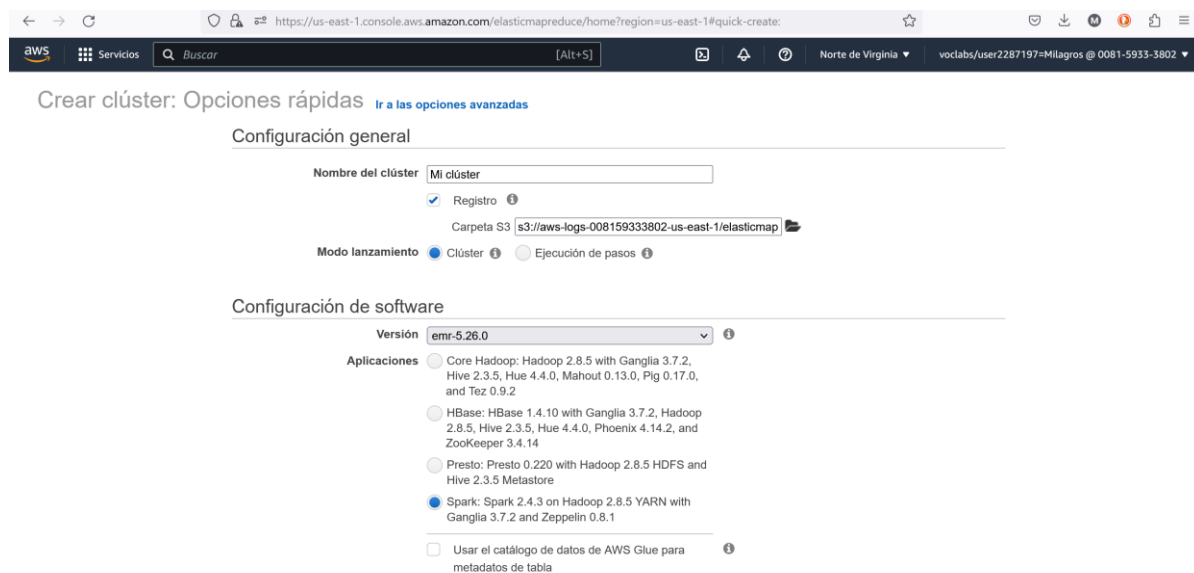


Ilustración 22: Amazon EMR

Se selecciona el tipo de instancia m5.xlarge, se continua sin par de claves EC2 y se crea el clúster. Una vez arrancado, ejecutamos el script de la Imagen 23, el cual nos permite ejecutar el JAR en el clúster, pasándole el identificador de clúster y el nombre del bucket [9].

```
#!/bin/bash

CLUSTER_ID=$1
MAIN_JAR=s3://$2/Queries.jar
MATCHES_PATH=s3://$2/matches.json
PLAYERS_PATH=s3://$2/players.json
OUTPUT_PATH=s3://$2/output
PROFILE=default

aws emr add-steps \
  --cluster-id $CLUSTER_ID \
  --steps Type=Spark,Name="My program",ActionOnFailure=CONTINUE,
  Args[--class,main.Queries,$MAIN_JAR,"--inputMatchesPath",$MATCHES_PATH,"--inputPlayersPath",$PLAYERS_PATH,"--outputPath",$OUTPUT_PATH]
  --profile $PROFILE \
  --region "us-east-1"
```

*Ilustración 23: Script para la ejecución del programa en Amazon EMR*

### 4.1. Resultados de las consultas

[illegible]

En la Figura 25, se observa otro diagrama que refleja el número de usuarios registrados desde el 2007 hasta el 2015. Podemos ver que conforme ha avanzado la tecnología con el lanzamiento de módulos como Fritz se ha experimentado un gran *boost* en el número de registros en la aplicación de Chess.com a partir del 2014.



En la Imagen 26, se pueden ver los diferentes resultados obtenidos en cada una de las partidas en el gráfico circular. Se muestra que la parte del gráfico que hace referencia a “win-resigned” ocupa gran parte. Esta etiqueta se refiere a las partidas en las que el jugador de blancas gana porque su contrincante se rinde. Asimismo, otro gran porcentaje lo acapara “resigned-win” que se da cuando las negras ganan por desistimiento de las blancas. En menor proporción está el trozo etiquetado con “win-checkmated” que se da cuando el jugador gana por jaque mate. Se observa que gran parte del gráfico se ocupa por tablas como “stalemate-stalemate” que se produce cuando el jugador de quien es el turno no tiene jugadas legales para realizar y el rey no se encuentra en estado de jaque, “repetition-repetition” que se da cuando la misma posición en el tablero ocurre al menos por tercera vez durante una partida, “insufficient-insufficient” se produce cuando no hay suficientes piezas para dar jaque mate al oponente y el juego terminará en empate de inmediato, mientras que “timevinsufficient-timevinsufficient” se da porque si un jugador no tiene suficiente material, el juego continúa porque el otro jugador todavía tiene la posibilidad de hacer jaque mate y a este último se le agota el tiempo entonces no pierde y la partida se queda en este estado, “agreed-agreed” son tablas por mutuo consentimiento y, por último, “50-move-50move” se produce cuando un jugador ha hecho los últimos 50 movimientos consecutivos sin que haya habido ningún movimiento de peón ni captura de pieza. Por lo tanto, se puede concluir que, dados los datos recogidos desde Chess.com y con los que se generó el dataset de partidas, un amplio porcentaje de las partidas acaban en tablas.

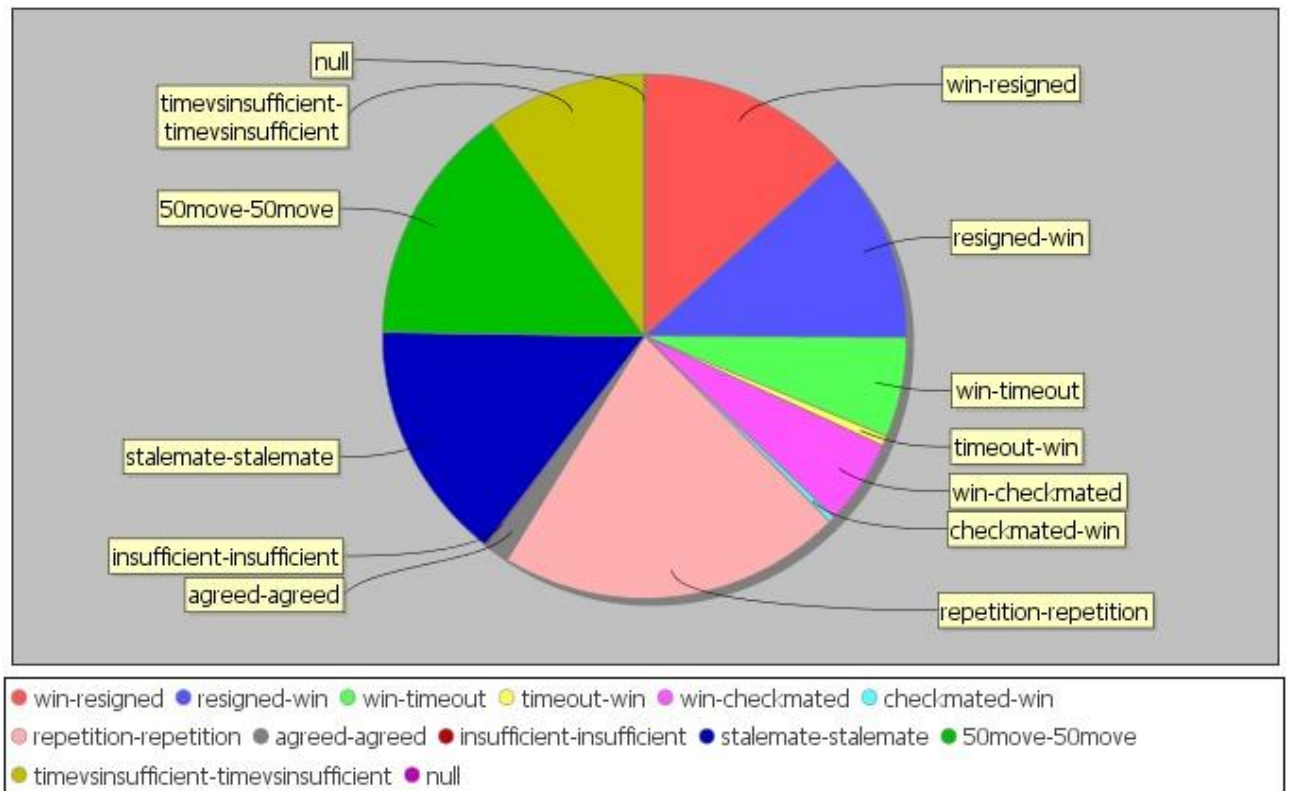


Ilustración 26: Visualización de la séptima consulta



## 4.2. Rendimientos comparados

La elaboración de este apartado tiene la finalidad de mostrar las ventajas que tiene la ejecución en el clúster respecto a la local mediante la comparación de rendimientos vista en la UI de Spark.<sup>1</sup> Se comprobó que cuanto menor era el tamaño del dataset de las partidas y jugadores, peor era el aprovechamiento de la potencia del clúster. En la Ilustración 27, se observan distintas ejecuciones en local que son las del usuario milam y hadoop la del clúster. En concreto, para un fichero de 1000 partidas se han obtenido tal y como se ilustra tiempos en local de 42s, 50s, 1.1min y 54s, mientras que el clúster donde aparentemente tendría que haberse ejecutado más rápidamente se ha demorado 1.4min, 1.5min y 1.6 min. No se cuenta la inicialización de los nodos ejecutores del clúster, simplemente se contempla la ejecución. El *delay* puede deberse a que la paralelización de los datos en los distintos clústeres siendo la entrada muy pequeña puede ser contraproducente.

2.4.3	<a href="#">local-1683128395323</a>	Chess	2023-05-03 17:39:53	2023-05-03 17:40:47	54 s	milam	2023-05-03 17:40:47	<a href="#">Download</a>
2.4.3	<a href="#">local-1682694775241</a>	Chess	2023-04-28 17:12:53	2023-04-28 17:13:56	1.0 min	milam	2023-04-28 17:13:56	<a href="#">Download</a>
2.4.3	<a href="#">local-1682694403201</a>	Chess	2023-04-28 17:06:41	2023-04-28 17:07:45	1.1 min	milam	2023-04-28 17:07:45	<a href="#">Download</a>
2.4.3	<a href="#">application_1682694178019_0001</a>	Chess	2023-04-28 17:05:52	2023-04-28 17:07:30	1.6 min	hadoop	2023-04-28 17:19:39	<a href="#">Download</a>
2.4.3	<a href="#">application_168268873597_0001</a>	Chess	2023-04-28 15:42:55	2023-04-28 15:44:26	1.5 min	hadoop	2023-04-28 15:48:08	<a href="#">Download</a>
2.4.3	<a href="#">local-1682508578454</a>	Chess	2023-04-26 13:29:36	2023-04-26 13:30:26	50 s	milam	2023-04-26 13:30:26	<a href="#">Download</a>
2.4.3	<a href="#">application_1682428507169_0001</a>	Chess	2023-04-25 15:23:20	2023-04-25 15:24:47	1.4 min	hadoop	2023-04-25 15:27:34	<a href="#">Download</a>
2.4.3	<a href="#">local-1682352590040</a>	Chess	2023-04-24 18:09:48	2023-04-24 18:10:31	42 s	milam	2023-04-24 18:10:31	<a href="#">Download</a>

Ilustración 27: Rendimiento mostrado en la Spark UI con dataset pequeño

Para ello, se ejecutó el programa de las descargas para que generase un dataset de un millón de partidas. Esto supuso un inconveniente al principio dado que se lanzaban excepciones por timeout. De acuerdo con las políticas de uso de la API vista en la Figura 28, no se indica nada respecto al volumen de datos.

**Rate Limiting**

Your serial access rate is unlimited. If you always wait to receive the response to your previous request before making your next request, then you should never encounter rate limiting.

However, if you make requests in parallel (for example, in a threaded application or a webserver handling multiple simultaneous requests), then some requests may be blocked depending on how much work it takes to fulfill your previous request. You should be prepared to accept a "429 Too Many Requests" response from our server for any non-serial request that you make.

In some cases, if we detect abnormal or suspicious activity, we may block your application entirely. If you supply a recognizable user-agent that contains contact information, then if we must block you application we will attempt to contact you to correct the problem.

Ilustración 28: Políticas de uso Api Chess.com

Solo se sugiere que en caso de realizarse solicitudes en paralelo la petición se rechazara dado que la aplicación no presenta concurrencia. Quizás por la actividad sospechosa indicada en el último apartado. Se intentó varias veces hasta que se consiguió la descarga.

2.4.3	local-1691876134939	Chess	2023-08-12 23:35:33	2023-08-13 03:00:49	3.4 h	milam	2023-08-13 03:00:53	<a href="#">Download</a>
2.4.3	local-1691854755935	Chess	2023-08-12 17:39:14	2023-08-12 21:07:59	3.5 h	milam	2023-08-12 21:08:05	<a href="#">Download</a>
2.4.3	local-1691846665444	Chess	2023-08-12 15:24:23	2023-08-12 15:25:25	1.0 min	milam	2023-08-12 15:25:25	<a href="#">Download</a>
2.4.3	local-1691322133283	Chess	2023-08-06 13:42:11	2023-08-06 17:18:36	3.6 h	milam	2023-08-06 17:18:43	<a href="#">Download</a>
2.4.3	local-1691298549439	Chess	2023-08-06 07:09:08	2023-08-06 10:42:43	3.6 h	milam	2023-08-06 10:42:56	<a href="#">Download</a>
2.4.3	local-1691269689267	Chess	2023-08-05 23:08:07	2023-08-06 02:49:36	3.7 h	milam	2023-08-06 02:49:46	<a href="#">Download</a>

*Ilustración 29: Rendimiento Spark UI cuando se producía error de memoria*

Como se muestra en la Ilustración 29, se obtuvieron tiempos de ejecución muy altos y finalizaban con un `java.lang.OutOfMemoryError: GC overhead limit exceeded`. Observando los logs se solventó el error cambiando el parámetro de la tolerancia del algoritmo del pagerank de 0.01 a 0.05 visto en la Imagen 30 para eliminar mayor número de nodos y que no se llenase la memoria.

```

g
    .pageRank
    .resetProbability(value = 0.15)
    .tol(value = 0.05)
    .run()
    .vertices
    .distinct()
    .orderBy(desc(columnName = "pagerank"))
    .coalesce(numPartitions = 1)
    .write
    .mode(SaveMode.Overwrite)
    .format(source = "json")
    .save(outputPath + "/q6")

: GraphFrame
: PageRank
: PageRank
: PageRank
: GraphFrame
: sql.DataFrame
: Dataset[Row]
: Dataset[Row]
: Dataset[Row]
: DataFrameWriter[Row]
: DataFrameWriter[Row]
: DataFrameWriter[Row]
: Unit


```

*Ilustración 30: Algoritmo de la sexta consulta*

Finalmente, se obtuvieron los rendimientos esperados. En local la ejecución resultó de 12 min en comparación a los 5 min del clúster como se muestra en la Figura 31.

Version	App ID	App Name	Started	Completed	Duration	Spark User	Last Updated	Event Log
2.4.3	local-1691939678226	Chess	2023-08-13 17:14:36	2023-08-13 17:26:41	12 min	milam	2023-08-13 17:26:41	<a href="#">Download</a>
2.4.3	application_1691938478806_0001	Chess	2023-08-13 16:56:25	2023-08-13 17:01:24	5.0 min	hadoop	2023-08-13 17:14:38	<a href="#">Download</a>
2.4.3	local-1691911934891	Chess	2023-08-13 09:32:13	2023-08-13 09:44:20	12 min	milam	2023-08-13 09:44:20	<a href="#">Download</a>

Ilustración 31: Rendimientos SparkUI para dataset de un millón de partidas

	2.4.3	Jobs	Stages	Storage	Environment	Executors	SQL / DataFrame	Chess application UI
---	-------	------	--------	---------	-------------	-----------	-----------------	----------------------

### Executors

[Show Additional Metrics](#)

#### Summary

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Excluded
Active(5)	0	0.0 B / 10.7 GiB	0.0 B	8	0	0	7313	7313	26 min (2.3 min)	30.1 GiB	2 GiB	1.6 GiB	0
Dead(2)	0	0.0 B / 5.2 GiB	0.0 B	4	0	0	616	616	1.8 min (6 s)	1.5 GiB	3.7 MiB	2.6 MiB	0
Total(7)	0	0.0 B / 15.9 GiB	0.0 B	12	0	0	7929	7929	27 min (2.4 min)	31.6 GiB	2 GiB	1.6 GiB	0

#### Executors

Show 20 entries


Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Logs
driver	ip-172-31-46-145.ec2.internal:38989	Active	0	0.0 B / 414.4 MiB	0.0 B	0	0	0	0	0	5.0 min (0.0 ms)	0.0 B	0.0 B	0.0 B	
1	ip-172-31-46-125.ec2.internal:40021	Dead	0	0.0 B / 2.6 GiB	0.0 B	2	0	0	306	306	53 s (3 s)	772.1 MiB	1.1 MiB	1.4 MiB	<a href="#">stdout</a> <a href="#">stderr</a>
2	ip-172-31-46-175.ec2.internal:44783	Active	0	0.0 B / 2.6 GiB	0.0 B	2	0	0	1937	1937	6.6 min (38 s)	10.4 GiB	547.9 MiB	356 MiB	<a href="#">stdout</a> <a href="#">stderr</a>
3	ip-172-31-46-125.ec2.internal:39139	Dead	0	0.0 B / 2.6 GiB	0.0 B	2	0	0	310	310	55 s (3 s)	770 MiB	2.6 MiB	1.2 MiB	<a href="#">stdout</a> <a href="#">stderr</a>
4	ip-172-31-46-175.ec2.internal:41491	Active	0	0.0 B / 2.6 GiB	0.0 B	2	0	0	2043	2043	5.5 min (36 s)	7.5 GiB	516.7 MiB	431.2 MiB	<a href="#">stdout</a> <a href="#">stderr</a>
5	ip-172-31-46-125.ec2.internal:34593	Active	0	0.0 B / 2.6 GiB	0.0 B	2	0	0	1646	1646	4.4 min (34 s)	6.3 GiB	472.5 MiB	424 MiB	<a href="#">stdout</a> <a href="#">stderr</a>
6	ip-172-31-46-125.ec2.internal:33571	Active	0	0.0 B / 2.6 GiB	0.0 B	2	0	0	1687	1687	4.2 min (32 s)	6 GiB	469.2 MiB	394 MiB	<a href="#">stdout</a> <a href="#">stderr</a>

Showing 1 to 7 of 7 entries

Previous **1** Next

Ilustración 32: Ejecutores utilizados para ejecutar la aplicación en el clúster

En la Ilustración 32, se observan las características del clúster EC2 utilizado. El nodo driver o principal en el cual se coordinan todas las operaciones del clúster no presentaba ningún core mientras que los otros seis nodos de trabajo encargados de realizar todas las tareas asignadas por el nodo driver presentan dos cores cada uno.

	2.4.3	Jobs	Stages	Storage	Environment	Executors	SQL / DataFrame	Chess application UI
---	-------	------	--------	---------	-------------	-----------	-----------------	----------------------

### Executors

[Show Additional Metrics](#)

#### Summary

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Excluded
Active(1)	0	0.0 B / 1.9 GiB	0.0 B	8	0	0	7929	7929	12 min (0.0 ms)	86.3 GiB	10.9 GiB	1.6 GiB	0
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0.0 ms (0.0 ms)	0.0 B	0.0 B	0.0 B	0
Total(1)	0	0.0 B / 1.9 GiB	0.0 B	8	0	0	7929	7929	12 min (0.0 ms)	86.3 GiB	10.9 GiB	1.6 GiB	0

#### Executors

Show 20 entries

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write
driver	Mila61583	Active	0	0.0 B / 1.9 GiB	0.0 B	8	0	0	7929	7929	12 min (0.0 ms)	86.3 GiB	10.9 GiB	1.6 GiB

Showing 1 to 1 of 1 entries

Previous **1** Next

Ilustración 33: Ejecutores utilizados para ejecutar la aplicación en local

Por otro lado, en la Imagen 33, se muestra que la ejecución en local simplemente se utilizó el nodo driver para realizar el conjunto de las 7929 tareas.

## 5. Conclusiones

En cuanto a la consecución de los objetivos, la descarga de los datos de la API de Chess.com se ha realizado con éxito mediante la utilización de la programación funcional. Ha sido la fase más compleja y la cual ha requerido más tiempo, ya que ha precisado de la profundización en los conceptos de este paradigma y uso de estructuras como los iteradores para optimizar la ejecución del programa, así como el aprendizaje de la librería de *cats* para la reutilización de funciones de orden superior.

Por otra parte, la elaboración de consultas utilizando Scala con Spark ha sido un reto, ya que desconocía el tipo de datos *Graphframe* que es con el cual se ha construido el dataframe obtenido a partir de los ficheros de jugadores y partidas de la fase anterior. Asimismo, dispone de una serie de algoritmos complicados no intuitivos como el *motif finding*. Se ha tenido que pensar bien qué algoritmos de los que ofrece el tipo de datos tiene sentido emplear para el dominio del ajedrez.

El despliegue en la nube supuso algo novedoso, debido a mi inexperiencia con los servicios de AWS. Sin embargo, supuso la fase más mecánica debido a la ejecución de pasos bien definidos para conseguir que el archivo jar del subproyecto de las consultas se ejecutase en el clúster. El único bloqueo importante que se ha obtenido ha supuesto el error de memoria obtenido al ejecutar el algoritmo del *PageRank* para ver los jugadores que más habían jugado del dataset obtenido, ya que la tolerancia era demasiado baja. Cuando el dataset aumentaba en gran medida, se generaban demasiados nodos obteniéndose el error en tiempo de ejecución. Se tardó en llegar a la solución del problema. En cuanto a la comparación de los resultados en la Spark UI, esta fase también fue muy directa en el sentido de que se presenta de forma clara lo que demora un programa. Al principio resultaba complicado interpretar lo que eran las tareas y sus correspondientes stages. Analizar esta parte de la SparkUI ha permitido comprender por qué se utiliza este *framework* y su utilización para optimizar la ejecución de las tareas mediante la paralelización en los diferentes nodos del clúster (en local los diferentes *cores* del ordenador).

La elaboración del presente proyecto ha resultado útil para entender por qué se ha profundizado en el paradigma de la programación funcional frente al paradigma imperativo con la utilización de bucles, lo cual no promueve la reutilización de código y también se traduce en mayor tiempo de codificación y, por ende, menor eficiencia. Se ha hecho el

rediseño funcional de algunos métodos utilizando las funciones de orden superior *traverse* y *unfold*, así como la modularización llevada a cabo a partir de la separación de dos proyectos dentro de la aplicación para la descarga y las consultas respectivamente.

La curva de aprendizaje ha sido alta debido al desconocimiento del uso práctico de iteradores, el rediseño funcional de ciertas funciones usando el paradigma funcional supuso una mejora continua para terminar de pulir el programa y, en general, las tecnologías utilizadas en el proyecto. Asimismo, hubo bastantes bloqueos los cuales no supe gestionar bien en determinados momentos de elaboración del presente proyecto, lo cual se tradujo en la postergación de la entrega del TFG.

## 5.1. Trabajo futuro

De cara al futuro, una mejora esencial es el afrontamiento de los bloqueos de manera más eficiente para no retrasar de manera excesiva la realización de cualquier otro proyecto. Asimismo, otra mejoría sobre el presente trabajo podría ser el mejor análisis de los algoritmos que ofrece la librería de *Graphframes* para la elaboración de otras consultas interesantes. Me gustaría profundizar en el paradigma de la programación funcional donde para implementar de manera correcta se requiere de un buen entendimiento de los conceptos. Además, el rediseño y *refactor* de las funciones implementadas promueve la reutilización de código y legibilidad. No estaría demás analizar en mayor profundidad los *stages* de la ejecución del programa para contemplar posibles mejoras en eficiencia. Esta ha pasado a segundo plano y no debería haber sido así, ya que al plantear cualquier programa se deben tener en consideración el tiempo de ejecución. Resultaría interesante utilizar otros componentes y librerías que ofrece Spark a los programadores además del Structured APIs con Spark Sql y Dataframes utilizados en la elaboración del presente trabajo como el Structured Streaming que permite el procesamiento por lotes o los algoritmos de machine learning incluidos en el módulo de Advanced Analytics. Por otro lado, la escasez de librerías para visualización de datos en Scala y su facilidad de uso hizo que fuese más complejo elaborar dicho apartado. Tengo la impresión de que se hubiese podido visualizar de mejor forma los resultados con Python al haber un número más elevado de librerías de visualización de datos.

## 6. Bibliografía

1. Chambers, B., & Zaharia, M. (2018). Spark: The definitive guide: Big data processing made simple. " O'Reilly Media, Inc."
2. API de Chess.com <https://www.chess.com/news/view/published-data-api>
3. Case-app <https://github.com/alexarchambault/case-app>
4. Iteradores [https://www.scala-lang.org/api/2.13.6/scala/collection/Iterator\\$.html](https://www.scala-lang.org/api/2.13.6/scala/collection/Iterator$.html)
5. Spray-json <https://github.com/spray/spray-json>
6. Traverse <https://typelevel.org/cats/typeclasses/traverse.html>
7. StateT <https://typelevel.org/cats/datatypes/statet.html>
8. Graphframes [https://graphframes.github.io/graphframes/docs/\\_site/user-guide.html](https://graphframes.github.io/graphframes/docs/_site/user-guide.html)
9. Documentación para ejecutar en Amazon EMR <https://github.com/jserranohidalgo/spark-intro/tree/master/workflow-example>
10. Plotly para visualización de datos en Scala <https://github.com/alexarchambault/plotly-scala>