

**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

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](#_Toc128476285)

[Palabras clave 2](#_Toc128476286)

[Índice 3](#_Toc128476287)

[1. Introducción 4](#_Toc128476288)

[2. Objetivos 6](#_Toc128476289)

[3. Descripción informática 7](#_Toc128476290)

[3.1. Creación del dataset de partidas 7](#_Toc128476291)

[3.1.1. Especificación 7](#_Toc128476292)

[3.1.2. Arquitectura 8](#_Toc128476293)

[3.1.3. Diseño 9](#_Toc128476294)

[3.2. Consultas con GraphFrames 10](#_Toc128476295)

[3.3. Despliegue en AWS 13](#_Toc128476296)

[4. Experimentos 14](#_Toc128476297)

[4.1. Resultados de las consultas 14](#_Toc128476298)

[4.2. Rendimientos comparados 15](#_Toc128476299)

[5. Conclusiones 16](#_Toc128476300)

[5.1. Trabajo futuro 17](#_Toc128476301)

[6. Bibliografía 18](#_Toc128476302)

[7. Apéndices 19](#_Toc128476303)

# 1. Introducción

*Big Data* se refiere al gran volumen de datos estructurados y no estructurados que generan y recopilan organizaciones e individuos para mejorar sus negocios. Estos datos pueden recopilarse a través de redes sociales, sensores, sistemas transaccionales, dispositivos móviles entre otras muchas fuentes. 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.

Apache Spark es un *framework* de código abierto creado para el procesamiento de estos datos masivos. Es escalable, ya que puede ejecutarse desde un ordenador local hasta miles de clústeres. Fue desarrollado en la Universidad de California a principios de la década de 2010 y donado a la Apache Software Foundation en 2014 permitiendo la creación de APIs más potentes y bibliotecas de alto nivel dentro del framework. 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 en los lenguajes de programación Java, Scala, Python y R. No solo permite el procesamiento por lotes de 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; MLib para observar patrones o tendencias en los datos mediante *machine learning* o GraphX para el procesamiento y análisis de grafos.

La programación funcional es un estilo de programación que se centra en el uso de funciones puras y evita el uso de estado mutable y efectos secundarios. En Spark, se pueden utilizar tanto enfoques de programación funcional como de programación imperativa. Sin embargo, el enfoque funcional se adapta mejor a Spark debido a que al utilizar funciones puras, 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. Por lo tanto, el enfoque funcional se considera una buena práctica al trabajar con Spark.

Amazon Web Services (AWS) es una plataforma informática en la nube ofrecida por Amazon. Estos servicios operan desde doce regiones geográficas en todo el mundo. Spark se puede usar con una variedad de fuentes de datos, incluido HDFS, Amazon S3 y sistemas de archivos locales. Se puede ejecutar en una variedad de administradores de clústeres, incluidos Hadoop YARN, Apache Mesos y su propio administrador de clústeres Spark independiente. Sim embargo, AWS proporciona un servicio llamado EMR (Elastic MapReduce) que permite ejecutar Spark en un clúster de instancias EC2. Esto le permite escalar fácilmente su clúster de Spark según sea necesario y pagar solo por los recursos que utiliza.

# 2. Objetivos

El presente proyecto ha tenido como **principales 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.

Se ha **profundizado en el** **paradigma de la programación funcional** mediante el **rediseño funcional** de algunos métodos utilizando las funciones de orden superior *traverse* y *unfold*, así como la **modularización** llevada a caboa partir de la separación de dos proyectos dentro de la aplicación para la **descarga** y las **consultas** respectivamente.

# 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**.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

**Ilustración 1: Estructura del proyecto raíz**

## 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 los diferentes *endpoints* de los que consta la API pública de Chess.com tal y como se muestra en el esquema de la **Imagen 2**.
* **Salida**:El dataset originado es un grafo en el cual los nodos son los jugadores y las aristas son las partidas obtenidas.

Diagrama

Descripción generada automáticamente

**Ilustración 2: Fuente de los datos**

### Arquitectura

En la **Figura 3**, se muestra la jerarquía de clases para la **modularización** llevada a cabo para la **descarga** de los datos. Algunas de las librerías y plugins utilizados son:

* **Caseapp**: El parseo de los argumentos de la línea de comandos.
* **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 JSONDiagrama, Esquemático

  Descripción generada automáticamente

**Ilustración 3: Jerarquía de clases de la descarga**

### Diseño

Tal y como se observa en la figura anterior, se distinguen cuatro partes principales dentro del paquete *download* recogidas en los subpaquetes *main*, *io*, *manager* y *data*. El primero recoge el objeto Download y la case class ArgumentsDownload. Dentro del primer objeto se encuentra el método *run*, el cual se ejecuta si el parseo de los argumentos por la línea de comandos se ha realizado adecuadamente.

En cuanto al paquete *io*, este se encarga de gestionar la entrada y salida del programa porque al finalizar se deben escribir las partidas y jugadores a disco. 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.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

**Ilustración 4: Pipeline de ejecución**

En cuanto al paquete *manager*,el objeto principal *Manager* define el método *apply* en el que define el pipeline visto en la **Imagen 4**. El método tiene como salida un iterador de tuplas de partidas y lista de jugadores que se guardará en dos ficheros JSON 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. En cuanto al objeto *Getters* tiene los *getters* necesarios para obtener los datos a lo largo del pipeline. El método *getTitledPlayers* devuelve un iterador con los nombres de los jugadores de un título dado; *getPlayerTournaments* que retorna un iterador con los torneos de dicho jugador; *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 Option[Profile] por si falla la petición y, por último, el método *getPlayers* que se utiliza para transformar el iterador de partidas en otro de tuplas de partidas y listas de dos jugadores que es la información necesaria para generar el dataset de las consultas. La clase *Getters* está asociada con *Parser* dado que se necesita parsear el JSON a una case class del paquete *data*; *Fold*, al utilizarse el método *fold* para atrapar la respuesta en un Try al ejecutar el método *get* de requests; y, por último, *UnfoldIterator* que contiene el método *unfold* utilizado por *getPlayers* y es una HOF que permite mantener un estado S mediante un conjunto para saber si ya se han procesado determinados jugadores y no tener que volver a solicitar la petición HTTP. Cabe señalar que el objeto *Fold* contiene una clase implícita denominada *TryFold* con el método *fold* constistente en un *pattern matching*, es decir, si se obtiene un éxito se ejecuta una rama y, en caso contrario, la otra. Por otro lado, la clase *UnfoldIterator* posee un *companion object* con dos clases implícitas 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 para proporcionar *feedback* acerca del número de partidas obtenidas. El objeto *Parser* permite parsear la información de los JSON a las correspondientes *case classes* del paquete *data*. 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.

## Consultas con GraphFrames

El objeto *Queries* contiene las consultas a realizar sobre el dominio ajedrecístico. Se produce el parseo de los argumentos por la línea de comandos especificando el nombre de los ficheros de entrada JSON obtenidos en la descarga y 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 5**, 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. Por otra parte, en la **Figura 6**, 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 del *Graphframe*. 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*.

Interfaz de usuario gráfica, Tabla

Descripción generada automáticamente con confianza media

**Ilustración 5:** Dataframe de los jugadores

**Interfaz de usuario gráfica, Texto

Descripción generada automáticamente con confianza media**

**Ilustración 6**: 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 7**, 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.

Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

Descripción generada automáticamente

**Ilustración 7**: Consulta de los porcentajes de las diferentes combinaciones de resultados en una partida

Por otra parte, en la **Imagen 8**, 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.

Interfaz de usuario gráfica

Descripción generada automáticamente

**Ilustración 8**: Consulta de las partidas de la defensa Siciliana

## 3.3. Despliegue en AWS

# 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