

**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](#_Toc128039840)

[Palabras clave 2](#_Toc128039841)

[Índice 3](#_Toc128039842)

[1. Introducción 4](#_Toc128039843)

[2. Objetivos 6](#_Toc128039844)

[3. Descripción informática 7](#_Toc128039845)

[3.1. Creación del dataset de partidas 8](#_Toc128039846)

[3.1.1. Especificación 8](#_Toc128039847)

[3.1.2. Arquitectura 8](#_Toc128039848)

[3.1.3. Diseño 9](#_Toc128039849)

[3.2. Consultas con GraphFrames 12](#_Toc128039850)

[3.3. Despliegue en AWS 13](#_Toc128039851)

[4. Experimentos 14](#_Toc128039852)

[4.1. Resultados de las consultas 14](#_Toc128039853)

[4.2. Rendimientos comparados 15](#_Toc128039854)

[5. Conclusiones 16](#_Toc128039855)

[5.1. Trabajo futuro 17](#_Toc128039856)

[6. Bibliografía 18](#_Toc128039857)

[7. Apéndices 19](#_Toc128039858)

# 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 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.com. Estos servicios operan desde 12 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:

* **Realización de consultas** sobre un dominio específico **utilizando** el framework para programación distribuida de **Spark**.
* **Propulsión** del **paradigma de la programación funcional** al permitir una **modularización**, **reutilización** y **legibilidad** del código. Dado que Spark se basa en la división de las tareas en distintos procesadores, el diseño funcional es **eficiente** al reducir los tiempos de ejecución respecto al código iterativo e imperativo.
* **Despliegue de la aplicación** mediante la configuración de un clúster de ordenadores que provee el servicio web **Amazon EC2**
* **Comparación del rendimiento** en local y en la nube mediante la herramienta **Spark UI**
* **Visualización de los datos** obtenidos tras ejecutar las consultas

Para ello se determinaron los siguientes subobjetivos:

* **Descarga de los datos** obtenidos **a través de** los diferentes *endpoints* que posee la **API pública de Chess.com**
* **Elaboración de un dataset** consistente en un grafo donde los nodos son ajedrecistas y las aristas son partidas
* **Rediseño funcional** de métodos utilizando las funciones de orden superior *traverse* y *unfold*
* **Modularización** con la distinción de dos subproyectos dentro de la aplicación. Uno de ellos dedicado a la descarga y el otro a las consultas.

# Descripción informática

La herramienta *sbt* sirve para la construcción y gestión de las dependencias del proyecto. En la **Ilustración 1**, se muestra la creación de un ***sbt multiproyecto*** al componerse el programa de **dos partes distinguidas**. La parte de las descargas se encuentra en el subpaquete *download* para obtener la información de las partidas de ajedrez a partir de Chess.com. Por otro lado, la parte de las consultas que se encuentra en el subpaquete *queries*, el cual se encarga de realizar consultas sobre el dataset originado.

Interfaz de usuario gráfica

Descripción generada automáticamente

**Ilustración 1: Estructura del proyecto raíz**

## Creación del dataset de partidas

### 3.1.1. Especificación

* **Entrada**: La aplicación obtiene los datos 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 JSON

Diagrama, 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 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 por la línea de comandos se ha realizado adecuadamente. *Download* está asociado con *Io*, ya que la información extraída de partidas y jugadores se guarda en un fichero JSON. Asimismo, se relaciona con *Manager* que realizará las peticiones HTTP y obtendrá la correspondiente información.

En cuanto al paquete *io*, este se encarga de gestionar la entrada y salida del programa, dado que 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 porta 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.

En cuanto al paquete *manager*,este tiene como objeto principal *Manager*, que posee un método *apply*, el cual contiene el pipeline donde se parsearán las clases con el método *parseJson* de spray-json y se enviarán las peticiones HTTP con requests. El método tiene como salida un iterador de tuplas de partidas y lista de jugadores. 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], ya que puede ser que falle 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. Esta clase 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 dado que el método *get* de requests puede fallar; y, por último, *UnfoldIterator* que contiene el método *unfold* utilizada por *getPlayers* es una función de orden superior 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*, la cual contiene 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.

## 3.2. Consultas con GraphFrames

(15-20 páginas)

## 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