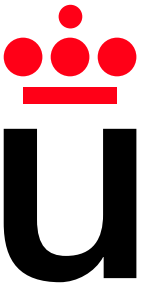


Universidad
Rey Juan Carlos

Escuela Técnica Superior
de Ingeniería Informática

Grado en Ingeniería del Software

Curso 2024-2025

Trabajo Fin de Grado

OPTIMIZACIÓN Y DESPLIEGUE EN LA NUBE DE PROCESOS DE BIG DATA EN SPARK

Autor: Raúl Velasco Rubio
Tutor: Juan Manuel Serrano Hidalgo

10/10/2024

Agradecimientos

A mamá y Álvaro.

Resumen

Este proyecto es la continuación de un trabajo previo que comenzó con la descarga de datos de la API de World of Warcraft. Ahora, se centra en el procesamiento y análisis de grandes volúmenes de datos utilizando Apache Spark y AWS, con el objetivo de estructurar y optimizar estos datos para un análisis profundo. World of Warcraft sirve como un caso de estudio ideal para aplicar estas tecnologías en el análisis de datos masivos en el contexto de videojuegos.

Palabras clave:

- Spark
- AWS
- EMR
- Cloud
- World of Warcraft
- Big Data
- Dataset
- Queries
- Parquet

Índice de contenidos

Índice de figuras

Índice de códigos

1 Introducción

Este proyecto es la continuación de una iniciativa más amplia que comenzó con la descarga y manejo de datos utilizando programación funcional. En esta segunda fase, nos enfocaremos en el procesamiento y análisis de grandes volúmenes de datos, empleando herramientas y técnicas avanzadas de Big Data.

El proyecto inicial se centró en la creación de un cliente para obtener datos de la API pública de World of Warcraft[1]. En esta fase, esos datos serán manipulados y transformados para generar un dataset estructurado y optimizado para análisis. Aquí es donde Apache Spark entra en juego. Spark[2, 3] es una plataforma de procesamiento de datos en clústeres open-source que permite realizar operaciones distribuidas de manera eficiente. Su capacidad para manejar grandes datasets en memoria y ejecutar consultas SQL distribuidas lo convierte en una herramienta fundamental para este proyecto.

World of Warcraft, uno de los MMORPG más populares y longevos, desarrollado por Blizzard Entertainment, proporciona una rica fuente de datos a través de su API. Estos datos incluyen información detallada sobre personajes, objetos, monturas y mucho más, reflejando las interacciones y actividades de millones de jugadores. Analizar estos datos ofrece una oportunidad única para entender patrones de comportamiento, tendencias económicas y otros aspectos dentro del juego. Además, como ya mencioné en el otro proyecto, es un videojuego que tiene mucha relevancia para mi a nivel personal ya que llevo jugándolo muchos años y me apasiona disfrutar de todo el universo que ofrece.

También se explorará el despliegue del proyecto en AWS[4], aprovechando la escalabilidad y potencia de la nube para comparar el rendimiento y eficiencia del procesamiento de datos en un entorno distribuido.

En resumen, este proyecto no solo continúa con el trabajo iniciado en la primera fase, sino que también amplía el alcance hacia el análisis y procesamiento de Big Data, utilizando tecnologías avanzadas como Apache Spark y AWS. World of Warcraft, con su vasta y dinámica base de datos, es el escenario ideal para aplicar y demostrar las capacidades de estas tecnologías en el manejo y análisis de grandes volúmenes de datos en el mundo de los videojuegos.

2

Objetivos

El objetivo principal de esta segunda parte del proyecto es procesar y analizar los grandes volúmenes de datos obtenidos de la API pública de World of Warcraft. Utilizando Apache Spark[2], transformaremos estos datos para crear un dataset estructurado y optimizado, permitiendo realizar consultas y análisis detallados. Además, exploraremos el despliegue del proyecto en AWS[4] para aprovechar la escalabilidad y potencia de la nube. Este enfoque nos permitirá demostrar y mejorar nuestras habilidades en el manejo y análisis de Big Data en un entorno dinámico y complejo, ampliando así las capacidades desarrolladas en la primera fase del proyecto.

1. **Construcción del dataset**

   Para comenzar nos enfocaremos en la construcción del dataset. El objetivo es obtener un dataset lo suficientemente grande y completo que permita un análisis exhaustivo posterior. Este dataset debe ser estructurado y optimizado para que las operaciones sobre él sean rápidas y mantengan la integridad de la información.

2. **Diseño de queries**

   El siguiente paso será diseñar consultas eficientes y efectivas para extraer información relevante del dataset. El diseño adecuado de estas consultas es crucial para obtener datos valiosos y útiles sobre nuestro dataset.

3. **Visualización de queries**

   Una vez diseñadas las consultas, se procederá a visualizar los resultados obtenidos. La visualización de datos es esencial para interpretar y comunicar los resultados de manera clara y comprensible. Utilizaremos herramientas como Plotly[5] que nos permitan presentar los datos de forma intuitiva y accesible.

4. **Despliegue en AWS**

   Finalmente, exploraremos el despliegue del proyecto en AWS[4]. Aprovecharemos la escalabilidad y potencia de la nube para comparar el rendimiento y eficiencia del procesamiento de datos en un entorno distribuido gracias a los EMR. Este despliegue nos permitirá evaluar las capacidades de nuestro sistema en un contexto real.

2.1. Metodología

El núcleo de este proyecto es el uso de Apache Spark[2, 3] para el procesamiento y análisis de grandes volúmenes de datos. Spark es una plataforma de procesamiento de datos en clústeres open-source, diseñada para realizar operaciones distribuidas de manera eficiente. Spark permite el desarrollo en diferentes lenguajes de programación: Scala, Python, Java y R, de los cuales hemos mantenido el desarrollo en Scala[6], un lenguaje de programación de alto nivel que combina paradigmas funcionales y orientados a objetos.

De la misma manera que en el primer proyecto, el proceso de desarrollo se ha llevado a cabo en diferentes contextos. He mantenido el uso de dos IDEs (o entornos de desarrollo): **IntelliJ IDEA**[7] y **Visual Studio Code** (VS Code)[8]. Ambos IDEs son compatibles con Scala y Spark y ofrecen diversas funcionalidades y plugins para mejorar la eficiencia del desarrollo, como el resaltado de sintaxis, el autocompletado de código y la integración con **Git**[9].

Por otro lado, como hemos mencionado que el desarrollo tuvo lugar en diferentes contextos, también hay que mencionar que tuvo lugar en diferentes máquinas

con diferentes sistemas operativos. En concreto, este proyecto se realizó en dos sistemas operativos: **Windows**[10] y **MacOS**[11]. Esto demuestra que el proyecto es multiplataforma y puede ser desarrollado y desplegado en diversos entornos.

Para la ejecución del proyecto y la visualización de las queries se utilizó principalmente un **Jupyter Notebook**[12]. El **Jupyter Notebook** se ejecutó a través de un contenedor **Docker**[13] con el kernel de Scala de **Almond**[14], proporcionando un entorno interactivo para ejecutar y manejar la aplicación.

Finalmente para el despliegue de la aplicación en **AWS**[4] se compiló una aplicación con una funcionalidad idéntica a la del Jupyter Notebook en **Java**[15] utilizando **SBT**[16].

2.1.1. Librería y Dependencias

Este proyecto depende de varias librerías y módulos, entre ellos:

- **Spark SQL**[17]**:** Proporciona las herramientas necesarias para trabajar con datos estructurados utilizando el lenguaje SQL. Es fundamental para la manipulación de DataFrames y la ejecución de consultas.
- **Spark Implicits:** Ofrecen métodos adicionales para convertir datos en DataFrames y Datasets, simplificando las operaciones en Spark.
- **Plotly Scala**[5]**:** Utilizado para la visualización de datos en Scala. La librería `plotly-almond` permite crear gráficos interactivos y se integra bien con Jupyter Notebooks[12].

3

Descripción informática

Todo el código desarrollado ha sido almacenado en un repositorio personal de GitHub llamado **wow-api**[18] siguiendo las prácticas habituales de control de versiones. En este repositorio se pueden encontrar todos los ficheros que se mencionan a lo largo de este documento y además incluye un archivo **README.md** con información relevante del propio proyecto, su estructura, sus funcionalidades y su uso.

En esta sección se detallará el proceso informático detrás de la construcción, análisis y despliegue del proyecto. Comenzaremos con la creación del dataset, que implica la transformación y organización de los datos obtenidos de la API de World of Warcraft en un formato estructurado y optimizado para su análisis. A continuación, exploraremos el diseño de las consultas (queries) que permitirán extraer información valiosa del dataset, utilizando las capacidades avanzadas de Apache Spark.

Posteriormente, se abordará la visualización de los resultados obtenidos a partir de las consultas, empleando la librería Plotly Scala, que nos proporciona herramientas versátiles y personalizables para representar gráficamente los datos. Finalmente, el proyecto se desplegará en AWS, aprovechando la potencia y escalabilidad de la nube para realizar comparativas de rendimiento y garantizar que el sistema sea capaz de manejar grandes volúmenes de datos en un entorno distribuido.

3.1. Construcción del dataset

Para la construcción del dataset, primeramente quise obtener una lista de jugadores que perteneciesen al mismo servidor de juego en el que yo juego (**Sanguino**) y obtuve esa lista a través de la página **WoWProgress**[19]. Seguidamente desarrollé un script en **Bash** que iteraba sobre esa lista y llamaba a la aplicación de Scala que desarrollamos en la primera parte de este proyecto para recolectar la información correspondiente al perfil de cada uno de estos jugadores. Como resultado se generaba un archivo en formato **JSON** que recopilaba de manera ordenada todos los datos. Se puede ver un fragmento de este script en el código 3.1.

```
while IFS=$'\r\n' read -r character_name; do
    # Convert character name to lowercase and encode it
    ...

    # Fetch the character data
    result=$(java -jar "$JAR_PATH" character-info --rs=sanguino
    ↪  --cn="$character_name_encoded" --bt=xxxxx --ns=profile-eu
    ↪  --locale=es_ES)

    # Check if the result matches the undesired pattern, and if so,
    ↪  skip this iteration
    ...

    # If this is not the first entry, prepend a comma
    ...

    # Append the result to the output file
    echo "$result" >> $OUTPUT_FILE

done < sanguino-players.txt
```

Código 3.1: Fragmento de la lógica de iteración sobre la lista de nombres de jugadores para la creación del dataset.

Una vez obtenido el archivo **JSON**, decidí convertirlo a formato **parquet** puesto que es un formato mucho más óptimo para llevar a cabo operaciones de Big Data, tanto en su almacenamiento, lectura y escritura, eficiencia en las operaciones... Esto lo hice utilizando unas sencillas funciones en **Python** como se ve en el código 3.2.

```
# Load the JSON file
json_file_path = './aggregated_data.json'
df = pd.read_json(json_file_path)
```

```
# Convert to Parquet
parquet_file_path = './aggregated_data.parquet'
df.to_parquet(parquet_file_path, engine='pyarrow')
```

Código 3.2: Transformación del dataset de formato JSON a formato parquet.

Teniendo ya disponible el dataset en formato **parquet** podemos proceder a cargarlo como **DataFrame** en Spark y así poder realizar las consultas necesarias sobre éste.

```
val characters: DataFrame =
↪  spark.read.parquet("dataset/aggregated_data.parquet")
```

Código 3.3: Carga del dataset en formato parquet en la sesión de Spark utilizando DataFrames.

También decidí preparar la conversión al formato **Dataset** de Spark ya que, a pesar de que las siguientes queries las voy a llevar a cabo exclusivamente en formato **DataFrame**, podría ser interesante comparar más tarde las posibles diferencias en el rendimiento al ejecutar las queries en cada uno de estos formatos. Para ello, primeramente creamos la clase **Character** con la definición de tipos de cada uno de sus parámetros.

```
    object CharacterSchema {
          case class Link(href: String)
          case class Self(self: Link)
          case class NameType(name: String, `type`: String)
          case class IDName(id: Long, key: Link, name: String)
          ...
          case class Body(
              _links: Self,
              achievement_points: Long,
              achievements: Link,
              achievements_statistics: String,
              active_spec: IDName,
              appearance: Link,
              ...
          )
          case class Character(body: Body)
    }
```

Código 3.4: Definición de la clase Character.

Y lo finalizamos apoyándonos en los **Encoders** que permiten la conversión de objetos al formato de filas de Spark, donde finalmente generamos el Dataset a partir de la clase creada.

```
1    val ds: Dataset[Character] = characters.as[Character]
```

Código 3.5: Creación del Dataset en Spark.

3.2. Diseño de queries

3.2.1. Promedio de item level por cada clase

Esta consulta agrupa los personajes según su clase (como guerrero, mago, etc.) y calcula el promedio del nivel de objetos (item level) que llevan equipados. Es útil para entender cuál es la clase de personaje que generalmente tiene el mejor equipamiento.

Para resolver esta consulta debemos tomar dos datos o columnas de nuestro dataset: por un lado las clases de los personajes y por otro el promedio de item level de cada personaje, que no deja de ser el promedio aritmético del nivel de objeto de cada uno de los objetos que lleva equipado el personaje. Una vez que reunamos ambos datos procederemos a agruparlos por clase y haremos el promedio de todos los niveles de objeto recogidos pertenecientes a cada clase.

Esta consulta utilizando la API de DataFrames se expresaría de la siguiente manera:

```
1    characters.groupBy("body.character_class.name").agg(avg("body.ave␣
     ↪  rage_item_level"))
```

Código 3.6: Promedio de item level por cada clase con la API de DataFrames.

Siendo `characters` nuestro DataFrame, podemos ver como agrupamos primero los datos por la columna que identifica la clase del personaje en:

```
1    .groupBy("body.character_class.name")
```

y después añadimos la operación que calcula el promedio de los niveles de objeto para cada clase en:

```
    .agg(avg("body.average_item_level"))
```

usando la transformación **avg**.

3.2.2. Distribución de razas por facción

Esta consulta agrupa los personajes según su facción (Alianza o Horda) y su raza (como humano, orco, etc.), y cuenta cuántos personajes hay de cada combinación. Esto ayuda a visualizar la distribución de razas dentro de cada facción.

Para resolver esta consulta, debemos tomar dos columnas de nuestro dataset: la facción a la que pertenece el personaje y la raza del personaje. Agrupamos los datos por la facción y la raza, y contamos el número de personajes en cada grupo.

Esta consulta utilizando la API de DataFrames se expresaría de la siguiente manera:

```
    characters.groupBy("body.faction.name", "body.race.name").count()
```

Código 3.7: Distribución de razas por facción con la API de DataFrames.

Siendo `characters` nuestro DataFrame, agrupamos primero los datos por las columnas que identifican la facción y la raza del personaje en:

```
    .groupBy("body.faction.name", "body.race.name")
```

y después añadimos la operación que cuenta el número de personajes en cada grupo en `.count()`.

3.2.3. Top personajes por clase con más puntos de logros

Esta consulta es más compleja y pretende calcular las tres clases con el mayor promedio de nivel de objeto y a continuación, dentro de estas clases, encuentra los dos personajes con más puntos de logros. Esto es útil para identificar a los personajes más destacados en las clases más poderosas en términos de nivel de equipamiento.

Para resolver esta consulta, necesitamos realizar varias subconsultas y luego unirlas. Primero deberemos realizar una subconsulta idéntica a la primera 3.2.1,

donde obtendremos el promedio de nivel de objeto de cada clase, para después seleccionar las tres clases con mayor promedio. Después deberemos escoger los puntos de logros de todos los personajes dentro de cada clase y finalizaremos la ultima consulta escogiendo los dos personajes con más puntos de logros dentro de las tres clases que seleccionamos en la primera subconsulta.

Esta consulta utilizando Spark SQL se expresaría de la siguiente manera:

```
WITH ClassAvgItemLevel AS (
SELECT body.character_class.name AS class_name,
 ↪  AVG(body.average_item_level) AS avg_item_level
FROM characters_table
GROUP BY body.character_class.name
),
TopClasses AS (
SELECT class_name, avg_item_level
FROM ClassAvgItemLevel
ORDER BY avg_item_level DESC
LIMIT 3
),
ClassTopCharacters AS (
SELECT c.body.name AS character_name, c.body.character_class.name AS
 ↪  class_name,
c.body.achievement_points, c.body.average_item_level,
ROW_NUMBER() OVER (PARTITION BY c.body.character_class.name ORDER BY
 ↪  c.body.achievement_points DESC) AS rank
FROM characters_table c
JOIN TopClasses tc ON c.body.character_class.name = tc.class_name
)
SELECT character_name, class_name, achievement_points,
 ↪  average_item_level
FROM ClassTopCharacters
WHERE rank <= 2
ORDER BY achievement_points DESC
```

Código 3.8: Top personajes por clase con más puntos de logros en Spark SQL.

Siendo la tabla sobre la que haremos las consultas SQL `characters_table`, lo primero que deberemos hacer es reunir el promedio del nivel de objeto medio de cada clase, de la misma manera que hicimos en la primera consulta 3.2.1. Para ello creamos una tabla temporal llamada `ClassAvgItemLevel` con:

```
    WITH ClassAvgItemLevel AS (
```

y luego igual que la otra vez seleccionaremos las columnas de clase y nivel de objeto (haciendo el promedio de este último con `AVG`), como se ve en:

```
SELECT body.character_class.name AS class_name,
↪  AVG(body.average_item_level) AS avg_item_level
```

Nótese que hemos renombrado ambas columnas gracias a `AS`, y que finalmente hemos agrupado los resultados por clase con `GROUP BY`.
El siguiente paso será crear otra tabla temporal (usando `AS`) a partir de la primera con:

```
FROM ClassAvgItemLevel
```

A esta la llamaremos `TopClasses` y la ordenaremos de manera descendente en función del promedio de nivel de objeto, gracias a:

```
FROM ClassAvgItemLevel
```

Para finalizar con esta segunda subconsulta, tomaremos las tres filas con los valores más altos de la tabla usando `LIMIT 3`.
La tercera operación consistirá en asignar un valor numérico a cada personaje dentro de cada clase en función de los puntos de logros que tengan y lo almacenaremos en una tabla temporal llamada `ClassTopCharacters`. La asignación de estos valores, a la que llamaremos `rank`, la hacemos con `ROW_NUMBER()` y la división por clases la hacemos con `PARTITION BY`, como se puede ver en:

```
ROW_NUMBER() OVER (PARTITION BY c.body.character_class.name ORDER
↪  BY c.body.achievement_points DESC) AS rank
```

Para terminar esta tercera operación filtramos los resultados obtenidos obteniendo únicamente los personajes que aparecen en las tres clases que seleccionamos en `TopClasses` utilizando `JOIN` y tomando como referencia para el filtro la columna de clase.
Para finalizar todo el proceso solo nos queda hacer una última consulta a la tercera tabla temporal que generamos (`ClassTopCharacters`) y aplicamos un filtro con `WHERE rank <= 2` donde especificamos que tomaremos únicamente los dos mejores personajes de cada clase.

3.3. Visualización de queries

Para la visualización de consultas utilizaremos Plotly Scala[5], una potente librería que permite crear gráficos interactivos y altamente personalizables. Plotly Scala se integra perfectamente con Spark, facilitando la representación visual de los datos y proporcionando una amplia gama de opciones para personalizar gráficos y diagramas.

Como paso previo, configuraremos el entorno para poder visualizar los gráficos de manera correcta. Para ello, truncaremos la impresión de información a tres líneas para evitar hacer scroll en los outputs y ver directamente cada uno de los gráficos. Esto lo haremos con la sentencia 3.9.

```
repl.pprinter() = repl.pprinter().copy(defaultHeight = 3)
```

Código 3.9: Configuración previa de los outputs para una mejor visualización de los gráficos.

3.3.1. Distribución de personajes por nivel

Como primer acercamiento a Plotly vamos a explicar como funciona en la consulta más sencilla que realizamos: la distribución de todos los personajes en función de su nivel de personaje.

Una vez que realicemos la consulta, recogeremos los resultados en un array de filas usando `.collect()` en la variable `characterLevels` y entonces procederemos con las operaciones necesarias para configurar Plotly.

```
val levels = characterLevels.map(row => Option(row.get(0)).map(_.toString.toInt).getOrElse(0)).toSeq
val counts = characterLevels.map(row => Option(row.get(1)).map(_.toString.toLong).getOrElse(0L)).toSeq

val data = Seq(
    Bar(
        x = levels,
        y = counts,
        marker = Marker(
            line = Line(color = Color.RGBA(0, 0, 0, 1.0), width = 1)
        )
      )
    )

```

```
14      val layout = Layout(title = "Distribution of Character Levels",
15                          xaxis = Axis(title = "Level"),
16                          yaxis = Axis(title = "Count", `type` =
                             ↪  AxisType.Log)
17                         )
18
19      plot(data, layout)
```

Código 3.10: Distribución de personajes por nivel.

Lo primero será adaptar las dos variables que queremos incluir en nuestro gráfico. Para ello, mapearemos ambas en una secuencia(array ordenado) que interpreta de manera correcta el tipo de dato o devuelve un nulo en caso de error, como vemos aquí:

```
1       val levels = characterLevels.map(row =>
        ↪  Option(row.get(0)).map(_.toString.toInt).getOrElse(0)).toSeq
2       val counts = characterLevels.map(row =>
        ↪  Option(row.get(1)).map(_.toString.toLong).getOrElse(0L)).toSeq
```

Después definiremos el gráfico dentro de una variable llamada `data`, especificando el tipo de gráfico que en este caso será un gráfico de barras usando `Bar()`. En la definición del gráfico incluimos las variables que componen cada eje y el estilo, usando un borde de línea de color negro para mejorar la visualización y que declaramos de la siguiente manera:

```
1       marker = Marker(
2                   line = Line(color = Color.RGBA(0, 0, 0, 1.0), width =
                    ↪  1)
3               )
```

Antes de terminar podemos personalizar el **layout** o **leyenda**, asignando un texto a cada uno de los ejes y un título al gráfico. Además, debido a la distribución de los datos, en este caso decidí representar el eje vertical (número de personajes) en escala logarítmica usando `` `type` = AxisType.Log ``, pues la diferencia entre algunos valores era demasiado grande como para poder ver la información correctamente.

Por último, terminamos la representación llamando a la función que pinta el gráfico e incluyendo toda la información que hemos elaborado previamente con `plot(data, layout)`.

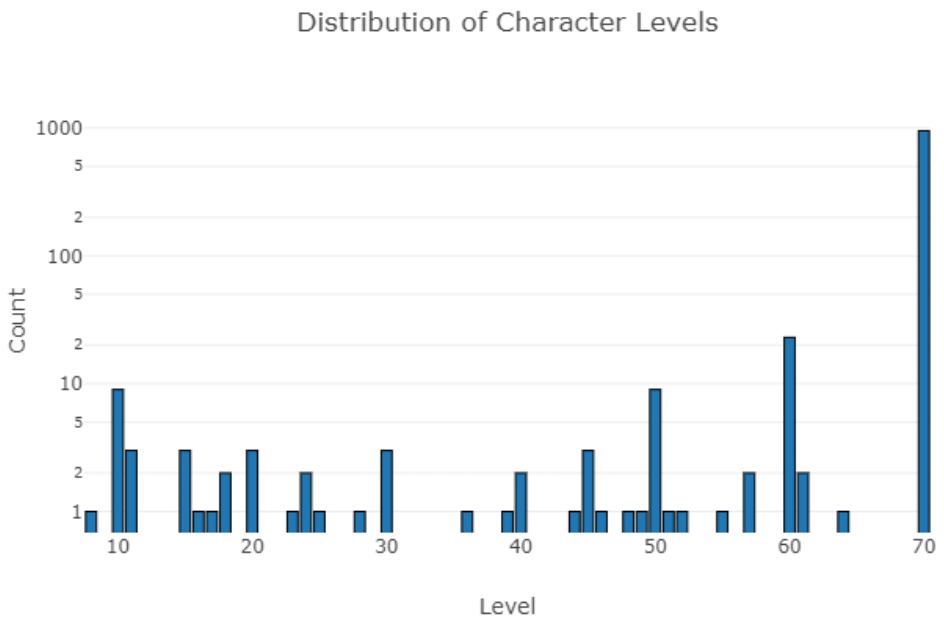


Figura 3.1: Distribución de personajes por nivel.

3.3.2. Promedio de item level por cada clase

Para el siguiente gráfico comenzaremos de la misma manera, recogiendo los resultados en un array de filas usando `.collect()` en una variable que ahora llamaremos `avgItemLevelPerClass`.

```
1  val classes = avgItemLevelPerClass.map(row =>
   ↪  Option(row.getString(0)).getOrElse("Unknown")).toSeq
2  val avgItemLevels = avgItemLevelPerClass.map(row =>
   ↪  Option(row.getDouble(1)).getOrElse(0.0)).toSeq
3
4  val classColors = Map(
5      "Caballero de la Muerte" -> Color.RGBA(196, 30, 58, 0.7),
6      "Cazador de demonios" -> Color.RGBA(163, 48, 201, 0.7),
7      "Druida" -> Color.RGBA(255, 124, 10, 0.7),
8      "Evocador" -> Color.RGBA(51, 147, 127, 0.7),
9      "Cazador" -> Color.RGBA(170, 211, 114, 0.7),
10     "Mago" -> Color.RGBA(63, 199, 235, 0.7),
11     "Monje" -> Color.RGBA(0, 255, 152, 0.7),
12     "Paladín" -> Color.RGBA(244, 140, 186, 0.7),
13     "Sacerdote" -> Color.RGBA(255, 255, 255, 0.7),
14     "Pícaro" -> Color.RGBA(255, 244, 104, 0.7),
15     "Chamán" -> Color.RGBA(0, 112, 221, 0.7),
16     "Brujo" -> Color.RGBA(135, 136, 238, 0.7),
```

```
        "Guerrero" -> Color.RGBA(198, 155, 109, 0.7)
    )

    val colors = classes.map(className =>
    ↪  classColors.getOrElse(className, Color.RGBA(128, 128, 128,
    ↪  0.7)))

    val data = Seq(
      Bar(
        x = classes,
        y = avgItemLevels,
        marker = Marker(
            color = colors,
            line = Line(color = Color.RGBA(0, 0, 0, 1.0), width = 1)
        )
      )
    )

    val layout = Layout(
      title = "Average Item Level per Character Class",
      xaxis = Axis(title = "Character Class"),
      yaxis = Axis(title = "Average Item Level")
    )

    plot(data, layout)
```

Código 3.11: Promedio de item level por cada clase.

La definición de este gráfico sigue una estructura idéntica al anterior, donde adaptamos primero las variables de los ejes, después definimos el gráfico y sus datos (también será un gráfico de barras verticales), definimos el **layout** y finalmente lo pintamos usando `plot(data, layout)`.

La diferencia principal que encontramos en esta ocasión es la inclusión del campo `color` dentro de la definición de estilo del propio gráfico. Este campo nos permite personalizar el color de las barras y para este he utilizado un mapa de colores que pintaba de un color determinado cada clase siguiendo los estándares oficiales de colores del videojuego que pude obtener en Wowpedia[20]. También por seguridad almacené este mapa en otra variable que pudiera devolver un color "nulo.en caso de no encontrar el nombre de columna apropiado, como se ve aquí:

```
    val colors = classes.map(className =>
    ↪  classColors.getOrElse(className, Color.RGBA(128, 128, 128,
    ↪  0.7)))
```

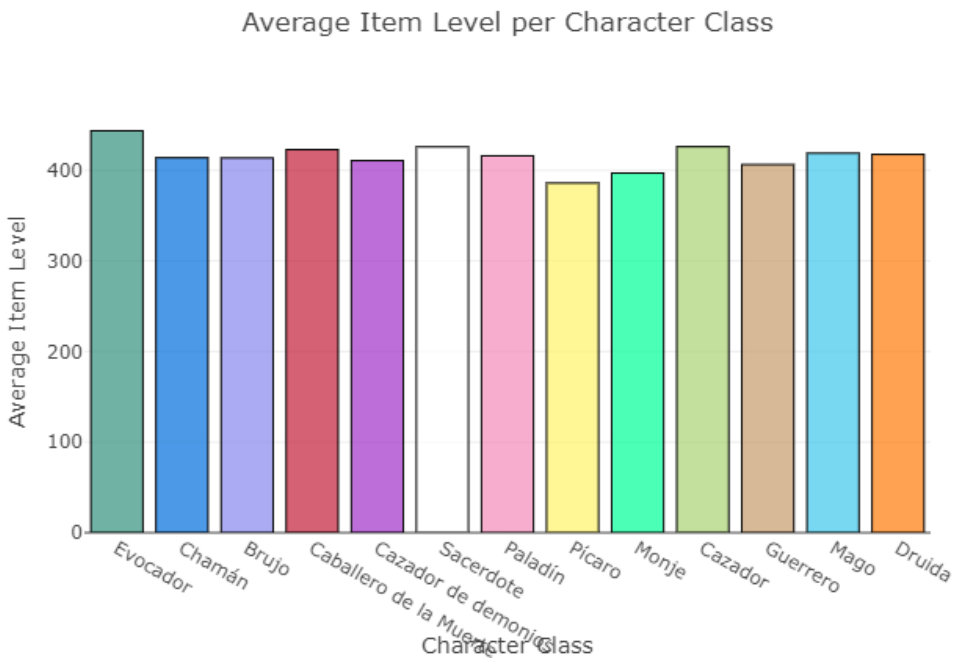


Figura 3.2: Promedio de item level por cada clase.

3.3.3. Distribución de razas por facción

Como en las ocasiones anteriores, recogeremos primero los resultados en un array de filas usando `.collect()` y lo almacenaremos en una variable llamada `factionRaceDistribution`.

```
val factions = factionRaceDistribution.map(row =>
↪  Option(row.getString(0)).getOrElse("Unknown")).toSeq
val races = factionRaceDistribution.map(row =>
↪  Option(row.getString(1)).getOrElse("Unknown")).toSeq
val counts = factionRaceDistribution.map(row =>
↪  Option(row.getLong(2)).getOrElse(0L)).toSeq

val colors = factions.map {
  case "Horda" => Color.RGBA(255, 0, 0, 0.7) // Red for Horde
  case "Alianza" => Color.RGBA(0, 0, 255, 0.7) // Blue for
  ↪  Alliance
  case _ => Color.RGBA(128, 128, 128, 0.7) // Grey for unknown
}

val data = Seq(
  Bar(
    x = races,
    y = counts,
```

```
        marker = Marker(
            color = colors,
            line = Line(color = Color.RGBA(0, 0, 0, 1.0), width = 1)
        )
      )
    )

    val layout = Layout(
      title = "Distribution of Characters by Faction and Race",
      xaxis = Axis(title = "Race"),
      yaxis = Axis(title = "Count"),
      barmode = BarMode.Stack
    )

    plot(data, layout)
```

Código 3.12: Distribución de razas por facción.

Dado que este también será un gráfico de barras verticales, mantenemos la misma estructura que anteriormente así que vamos a explicar las diferencias que encontramos.

Lo primero que destaca es que esta vez adaptaremos tres variables en lugar de dos, ya que dos de ellas nos servirán para representarlas en los ejes (`races` y `counts`), y usaremos una tercera para determinar el color de las barras: `factions`.

Lo siguiente que vemos es una nueva definición de colores, esta vez utilizando un `case` para pintar los colores que queremos y a su vez cubrir el caso nulo:

```
    val colors = factions.map {
      case "Horda" => Color.RGBA(255, 0, 0, 0.7) // Red for Horde
      case "Alianza" => Color.RGBA(0, 0, 255, 0.7) // Blue for
      ↪  Alliance
      case _ => Color.RGBA(128, 128, 128, 0.7) // Grey for unknown
    }
```

La última diferencia que se puede apreciar es que esta vez utilizaremos un gráfico de barras acumuladas o **Stack bars**, gracias a la sentencia `barmode = BarMode.Stack`. Esto lo haremos para evitar duplicar barras en los casos en los que una raza pueda pertenecer a diferentes facciones.

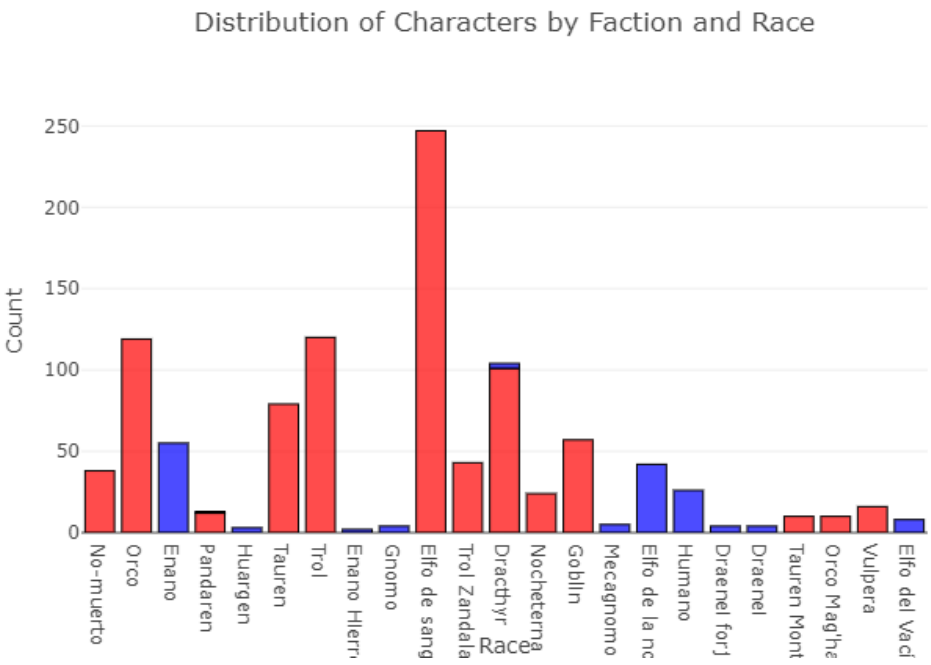


Figura 3.3: Distribución de razas por facción.

3.3.4. Top personajes por clase con más puntos de logros

De nuevo, volveremos a recoger los resultados en un array de filas usando `.collect()` y lo almacenaremos en una variable llamada `topCharacters`.

```
val groupedData = topCharacters.groupBy(row => row.getString(1))

val data = groupedData.map { case (className, rows) =>
  val names = rows.map(row =>
  ↪  Option(row.getString(0)).getOrElse("Unknown")).toSeq
  val points = rows.map(row =>
  ↪  Option(row.getLong(2)).getOrElse(0L)).toSeq
  val color = classColors.getOrElse(className, Color.RGBA(128,
  ↪  128, 128, 0.7))

  Bar(
    x = names,
    y = points,
    name = className,
    marker = Marker(
        color = color,
        line = Line(color = Color.RGBA(0, 0, 0, 1.0), width = 1)
    )
  )
```

```
17      }.toSeq
18
19      val layout = Layout(
20        title = "Top 2 Characters by Achievement Points within Top 3
          ↪  Classes by Average Item Level",
21        xaxis = Axis(title = "Character Name"),
22        yaxis = Axis(title = "Achievement Points"),
23        barmode = BarMode.Group,
24        width = 1000,
25        height = 600
26      )
27
28      plot(data, layout)
```

Código 3.13: Top personajes por clase con más puntos de logros.

Al igual que en el caso anterior, dado que la estructura principal es prácticamente idéntica vamos a explicar únicamente las diferencias. La principal es que vamos a usar un gráfico de barras agrupadas usando `barmode = BarMode.Group`, para lo cual necesitamos agrupar previamente los datos por clase:

```
1       val groupedData = topCharacters.groupBy(row => row.getString(1))
```

Con esto ya podremos pintar nuestro gráfico de barras agrupadas por clase correctamente, para el cual hemos desplazado la definición de las variables de los ejes dentro de la propia definición del gráfico. Además, reutilizamos el mapa de colores que utiliza los colores oficiales del videojuego. El último detalle a destacar es que hemos definido una altura (`height`) y anchura (`width`) específicos para este gráfico para mejorar su visualización.

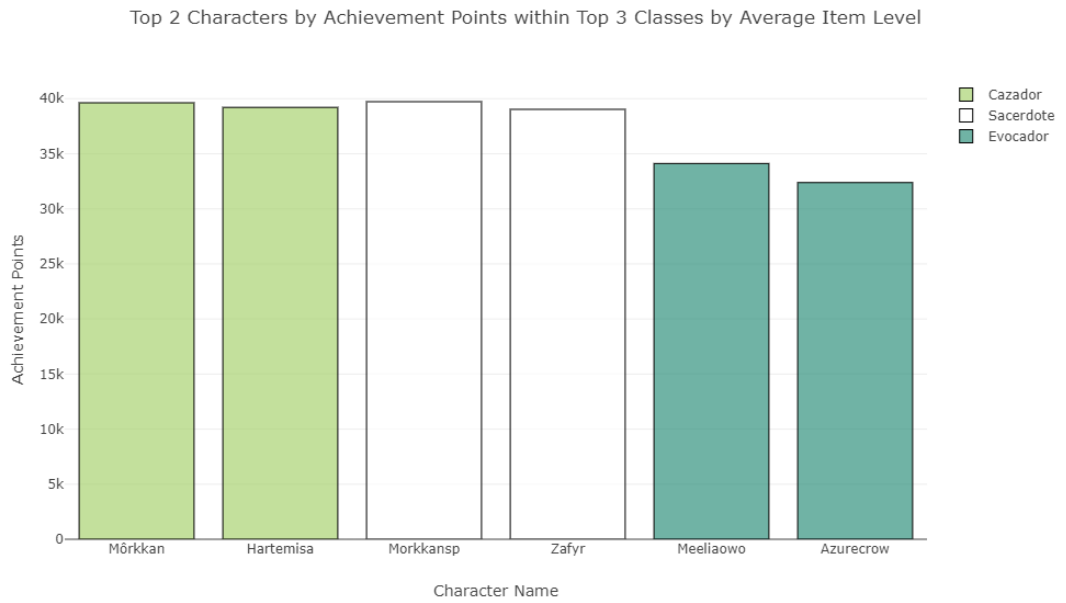


Figura 3.4: Top personajes por clase con más puntos de logros.

3.3.5. Top guilds con mayor cantidad de personajes al nivel 70 y su promedio de item level

Por última vez vamos a recoger los resultados en un array de filas usando `.collect()` y lo vamos a almacenar en la variable `level70Guilds`.

```
1    val guildNames = level70Guilds.map(row =>
     ↪  Option(row.getString(0)).getOrElse("Unknown")).toSeq
2    val level70Counts = level70Guilds.map(row =>
     ↪  Option(row.getLong(1)).getOrElse(0L).toDouble).toSeq
3    val avgItemLevels = level70Guilds.map(row =>
     ↪  Option(row.getDouble(2)).getOrElse(0.0)).toSeq
4
5    val barData: Trace = Bar(
6        x = guildNames,
7        y = level70Counts,
8        marker = Marker(
9          color = Color.RGBA(31, 119, 180, 0.7),
10         line = Line(color = Color.RGBA(0, 0, 0, 1.0), width = 1)
11       )
12     )
13
14   val lineDataWithSecondaryAxis: Trace = Scatter(
```

```
      guildNames,
      avgItemLevels,
      mode = ScatterMode(ScatterMode.Lines, ScatterMode.Markers),
      line = Line(color = Color.RGBA(255, 127, 14, 1.0)),
      yaxis = AxisReference.Y2
    )

    val layout = Layout(
      title = "Top 5 Guilds by Level 70 Character Count and Average
      ↪  Item Level",
      xaxis = Axis(title = "Guild Name"),
      yaxis = Axis(title = "Level 70 Count", side = Side.Left),
      yaxis2 = Axis(title = "Average Item Level", side = Side.Right,
      ↪  overlaying = AxisAnchor.Y)
    )

    val combinedData = Seq(barData, lineDataWithSecondaryAxis)

    plot(combinedData, layout)
```

Código 3.14: Top guilds con mayor cantidad de personajes al nivel 70 y su promedio de item level.

En esta ocasión la estructura pretende ser la misma pero hemos aplicado una serie de cambios suficientemente significativos, ya que nuestra intención es representar dos tipos de gráficos distintos al mismo tiempo. Para ello adaptaremos tres variables primero, dos para los ejes verticales y otra para el eje horizontal, que será compartida entre ambos gráficos (`guildNames`).

Lo siguiente que debemos hacer es definir las dos trazas o tipos de gráfico que vamos a dibujar. El primero de ellos será un grafico de barras exactamente igual a los que hemos definido previamente, mientras que el segundo se tratará de un grafico de dispersión.

Para este segundo gráfico deberemos definir las variables de los ejes y el estilo de la misma manera, pero añadiremos dos nuevos campos: un campo `mode` donde especificaremos si queremos que se representen solo puntos, líneas que los unan o ambos (como en nuestro caso); y un campo `yaxis` , donde especificaremos que va a utilizar un segundo eje vertical como referencia.

```
    val lineDataWithSecondaryAxis: Trace = Scatter(
      guildNames,
      avgItemLevels,
      mode = ScatterMode(ScatterMode.Lines, ScatterMode.Markers),
      line = Line(color = Color.RGBA(255, 127, 14, 1.0)),
```

```
      yaxis = AxisReference.Y2
    )
```

En la definición del **layout** también encontramos diferencias, pues tenemos que explicar como representar ambos ejes verticales. Para ello, utilizamos el campo `side` para explicar en qué lado deberá ir la leyenda de cada gráfico y el campo `overlaying` para expresar que ambos ejes verticales se superponen:

```
    yaxis = Axis(title = "Level 70 Count", side = Side.Left),
    yaxis2 = Axis(title = "Average Item Level", side = Side.Right,
    ↪  overlaying = AxisAnchor.Y)
```

Por último, agrupamos ambas trazas en una única secuencia y llamamos a la función que las pinta utilizando esta combinación como data.

```
    val combinedData = Seq(barData, lineDataWithSecondaryAxis)
    plot(combinedData, layout)
```

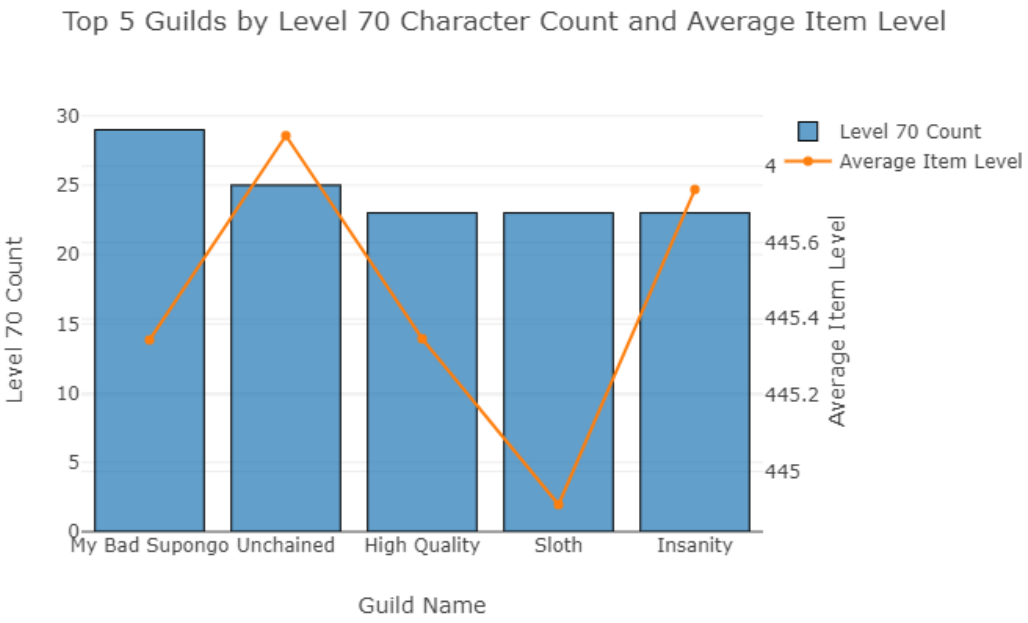


Figura 3.5: Top guilds con mayor cantidad de personajes al nivel 70 y su promedio de item level.

3.4. Despliegue en AWS

El despliegue en AWS ha utilizado como referencia el repositorio **spark-intro**[21], que ha servido de guía a la hora de desarrollar los pasos necesarios para el correcto funcionamiento del despliegue.

Como entorno de desarrollo he utilizado **AWS Academy Learner Lab**, que es un entorno práctico de aprendizaje para estudiantes en la nube ofrecido por AWS.

Hemos comenzado creando un bucket en el servicio **S3** de AWS, que es el servicio de almacenamiento Cloud que ofrece Amazon. Tanto este bucket como todas las operaciones en AWS se pueden realizaar directamente en la consola, pero en mi caso las he llevado a cabo utilizando el **CLI** como se ve en el código 3.15.

```
aws s3 mb s3://rvelasco-wow --region us-east-1 --endpoint-url
↪  https://s3.us-east-1.amazonaws.com
```

Código 3.15: Creación del bucket en S3 utilizando el CLI de AWS.

En este bucket hemos cargado el dataset en dos formatos diferentes: **json** y **parquet**, y la propia aplicación compilada en formato **jar**. Para ello hemos vuelto a utilizar el **CLI** y hemos dividido el bucket en dos carpetas para repartir los archivos que hemos mencionado anteriormente:

```
aws s3 cp --recursive data/aggregated_data.parquet
↪  s3://rvelasco-wow/data/aggregated_data.parquet
```

Código 3.16: Carga del dataset en formate parquet en el bucket.

Y para terminar usamos la librería **Config** para indicar en un **application.conf** de dónde debe leer estos archivos la aplicación.

```
1     val config: Config = ConfigFactory.load()
2     val inputJsonPath = config.getString("input-json-path")
3     val inputParquetPath = config.getString("input-parquet-path")
4
5     input-json-path="s3://rvelasco-wow/data/aggregated_data.json"
6     input-parquet-path="s3://rvelasco-wow/data/aggregated_data.parque
      ↪  t"
```

Código 3.17: Configuración para leer archivos en AWS.

El siguiente paso es crear un cluster en EMR, que es el servicio de clulstering de AWS para análisis de BigData. Esto lo hacemos directamente desde la consola de AWS utilizando la configuración por defecto que nos recomienda la guía de AWS Academy Learner Lab.

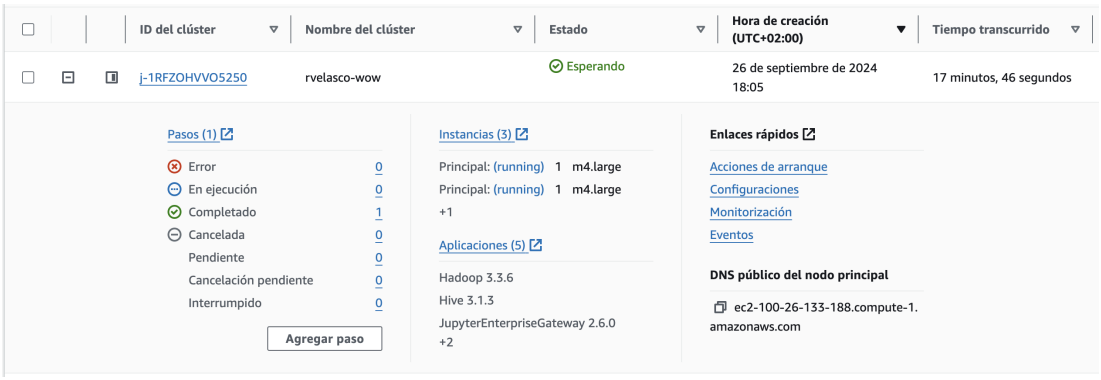


Figura 3.6: Detalles del cluster de EMR.

Por último, para poder ejecutar nuestra aplicación dentro del cluster tenemos que crear un paso o step que llame al archivo .jar que hemos subido en nuestro bucket de S3, como se ve en la figura 3.7.

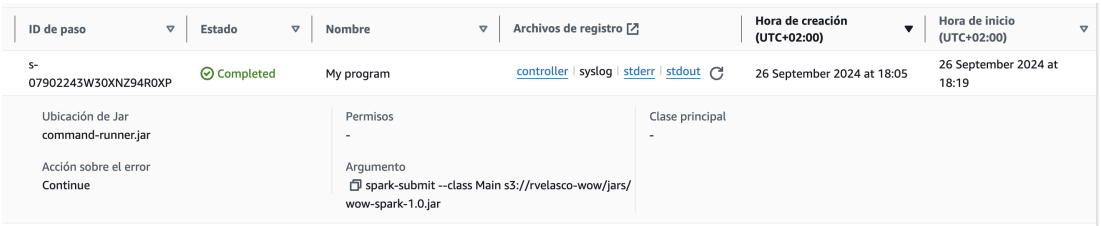


Figura 3.7: Detalles de paso del cluster.

La creación de este paso la llevamos a cabo también con un comando del CLI como el que se muestra en el código 3.18, donde indicamos la ubicación del ejecutable en S3 junto con el id del cluster y una serie de argumentos de configuración.

```
aws emr add-steps \
  --cluster-id $CLUSTER_ID \
  --steps Type=Spark,Name="My program",ActionOnFailure=CONTINUE,Args=
  ↪  [--class,Main,s3://rvelasco-wow/jars/wow-spark-1.0.jar]
  ↪  \
  --profile default
```

Código 3.18: Creación del paso dentro del cluster para la ejecución de la aplicación.

Una vez se ha completado la ejecución podemos acceder a una serie de logs donde podemos ver el output de la ejecución y también la salida **stderr** que podemos utilizar para debugear la aplicación en caso de que hubiera algún tipo de error. También disponemos de una herramienta intrínseca en los cluster de EMR llamada **Historial de Servidor de Spark**, en la que podemos analizar con mucho más detalle los resultados de la ejecución.

Entre los detalles que podemos ver es una lista de las queries lanzadas junto con los jobs en los que ha dividido su ejecución AWS y el tiempo que ha tomado cada una de ellas, como se ve en la imagen 3.8. Es importante mencionar que para poder identificar cada una de las queries y los jobs asociados he utilizado **spark.sparkContext.setJobDescription**, que permite personalizar la ejecución de un determinado job o tarea de Spark añadiéndole un String como identificador.

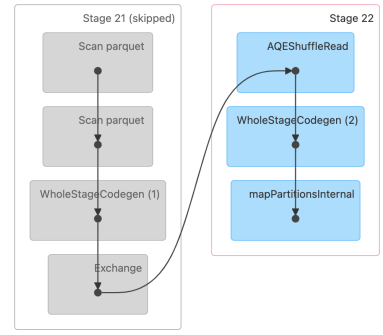
ID ▾	Description	Submitted	Duration	Job IDs
17	Top 5 guilds by level 70 characters (DS) +details	2024/09/26 16:48:21	0.5 s	[34][35]
16	Ranked characters by achievement points (DS) +details	2024/09/26 16:48:19	2 s	[30][31][32][33]
15	Top 5 guilds by level 70 characters (DF) +details	2024/09/26 16:48:17	2 s	[28][29]
14	Ranked characters by achievement points (DF) +details	2024/09/26 16:48:13	4 s	[24][25][26][27]
13	Characters with 'Represión' spec (DS) +details	2024/09/26 16:48:10	2 s	[23]
12	Amount of characters per race (DS) +details	2024/09/26 16:48:09	0.4 s	[21][22]
11	Top 10 by achievement points (DS) +details	2024/09/26 16:48:09	0.3 s	[20]
10	Top 5 by equipped item level (DS) +details	2024/09/26 16:48:08	0.3 s	[19]
9	Filter by level 70 and get average item level (DS) +details	2024/09/26 16:48:08	0.4 s	[17][18]

Figura 3.8: Queries ejecutadas en el step del cluster.

También podemos ver una lista de cada uno de los jobs lanzados (imagen 3.9) con el tiempo que han tardado en completarse y la query asociada, además de poder indagar aún más en cada uno de estos jobs (imagen 3.10), que incluye los diagramas de análisis y la cantidad de datos necesitados para leer y escribir sobre el DataFrame.

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
35	Top 5 guilds by level 70 characters (DS) show at Main.scala:96	2024/09/26 16:48:22	59 ms	1/1 (1 skipped)	1/1 (1 skipped)
34	Top 5 guilds by level 70 characters (DS) show at Main.scala:96	2024/09/26 16:48:21	0.2 s	1/1	1/1
33	Ranked characters by achievement points (DS) show at Main.scala:96	2024/09/26 16:48:21	62 ms	1/1 (1 skipped)	1/1 (1 skipped)
32	Ranked characters by achievement points (DS) show at Main.scala:96	2024/09/26 16:48:20	0.6 s	1/1	1/1
31	Ranked characters by achievement points (DS) $anonfun$withThreadLocalCaptured$1 at FutureTask.java:264	2024/09/26 16:48:20	61 ms	1/1 (1 skipped)	1/1 (1 skipped)
30	Ranked characters by achievement points (DS) show at Main.scala:96	2024/09/26 16:48:20	0.2 s	1/1	1/1
29	Top 5 guilds by level 70 characters (DF) show at Main.scala:96	2024/09/26 16:48:18	0.3 s	1/1 (1 skipped)	1/1 (1 skipped)
28	Top 5 guilds by level 70 characters (DF) show at Main.scala:96	2024/09/26 16:48:17	0.4 s	1/1	1/1
27	Ranked characters by achievement points (DF) show at Main.scala:96	2024/09/26 16:48:16	0.4 s	1/1 (1 skipped)	1/1 (1 skipped)

Figura 3.9: Jobs lanzados en el step del cluster.



▾ Completed Stages (1)

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Stage Id ▾	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
22	Average item level by character class (DS) show at Main.scala:96 +details	2024/09/26 16:48:08	30 ms	1/1			969.0 B	

Figura 3.10: Detalles de un job lanzado en el step del cluster.

4

Experimentos / validación

4.1. Resultados de las queries

4.1.1. Distribución de personajes por nivel

Si nos fijamos en la distribución de los personajes en función de su nivel, podemos ver que el gráfico sigue una distribución que se podría ajustar a una parábola cóncava. Esto se puede explicar porque generalmente la mayor concentración de personajes siempre es a nivel máximo, pero también muchos jugadores crean nuevos personajes y los van abandonando a medida que juegan, de ahí que veamos muchos a nivel bajo y muchísimos más a nivel alto.

También se puede observar que hay picos importantes en cada decena (20, 30, 50, 60..). Muy probablemente se deba al funcionamiento dentro del propio videjuego. La historia del videojuego está dividido en expansiones o capítulos que generalmente abarcan una decena de niveles. Es por esto que podemos ver estos picos en cada decena, porque los jugadores hayan completado un determinado capítulo con su personaje y no hayan continuado con el capítulo siguiente.

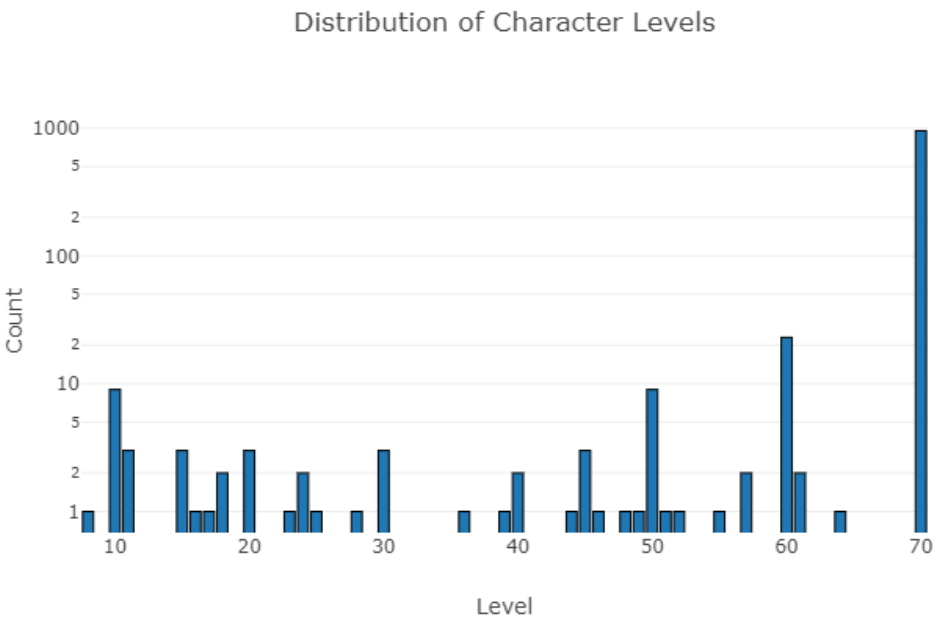


Figura 4.1: Distribución de personajes por nivel.

4.1.2. Promedio de item level por cada clase

En este gráfico podemos ver que, generalmente, todas las diferentes clases se juegan por igual ya que se encuentran en valores muy similares. El hecho de que destaquen algunas puede ser por moda o porque en ese momento las características y estadísticas de esa clase en concreto estén ligeramente superior o inferior al resto.

Por ejemplo, vemos como los Evocadores están ligeramente por encima del resto de clases, ya que es una clase nueva y hay muchísima más gente jugándola en comparación a las demás. También vemos como los Sacerdotes, Cazadores y Caballeros de la Muerte están un poco por encima, pues estas son las clases que tienen una ligera ventaja en estadísticas actualmente y hace que la gente las juegue más, lo que les permite llegar más lejos y conseguir mayor nivel de objeto. En cambio, tanto Pícaros como Monjes están ligeramente por debajo de las demás, pues es justamente el caso opuesto a las tres que hemos mencionado anteriormente: sus estadísticas actuales son ligeramente inferiores y por tanto se juegan menos y son menos capaces de alcanzar niveles de objeto más altos.

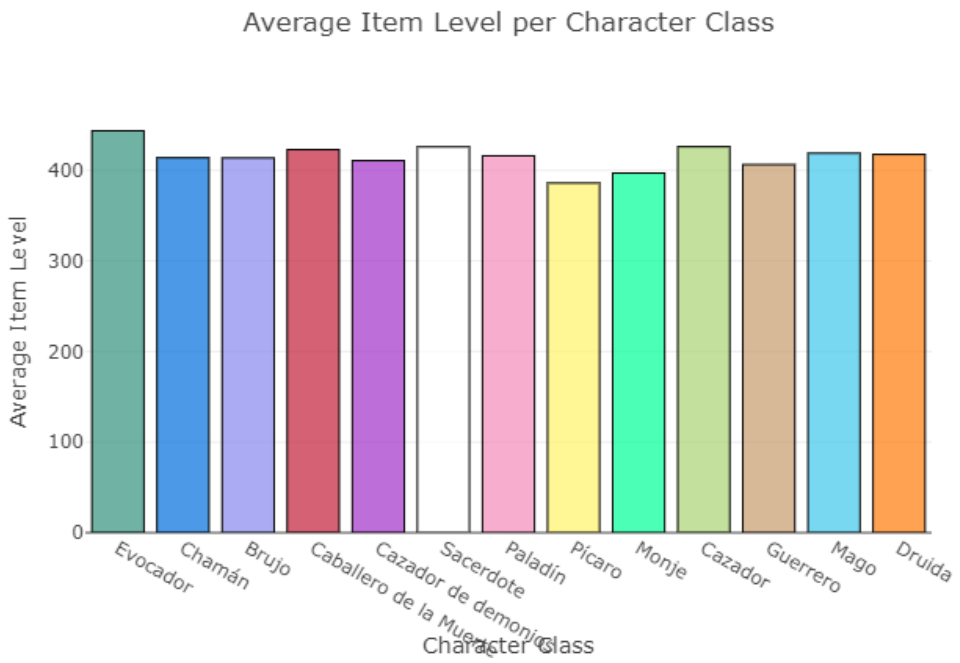


Figura 4.2: Promedio de item level por cada clase.

4.1.3. Distribución de razas por facción

En esta ocasión nos encontramos con un gráfico muy interesante para el que hay que dar algo más de contexto. Por un lado querría mencionar que todos los datos están extraídos del reino en el que yo juego: Sanguino. El videojuego World of Warcraft, al ser online, distribuye la carga de jugadores en diferentes reinos o servidores y cada jugador elige en cuál quiere jugar en base a la cercanía del servidor (Europa, Asia, América del Sur...) y el idioma oficial de ese reino (Español, Inglés, Alemán...). Por otro lado los personajes del videojuego se dividen en dos facciones principales: Alianza y Horda, facciones que están enfrentadas y permite a los jugadores combatir entre ellos. Por último cada personaje pertenece a una raza en concreto que se elige en el momento de su creación, y cada raza pertenece exclusivamente a una de las dos facciones mencionadas. La única excepción serían los Pandaren y los Dracthyr, que son razas que pueden elegir combatir por una u otra facción. Con todo esto podemos observar en el gráfico las razas que están asignadas a la Horda en rojo y las que están asignadas a la Alianza en azul, con las dos excepciones donde vemos las dos barras de colores acumuladas una encima de la otra.

La razón por la que este gráfico es tan interesante es porque los reinos no tienen una facción asignada, puedes crear un personaje de la facción que desees en todos ellos. Sin embargo, hay una tendencia de los jugadores que prefieren determinada facción por jugar en el mismo reino. Esto provoca que haya reinos

de la Alianza y reinos de la Horda, pero son preferencias de los jugadores porque en ninguna parte se exige que se deba escoger una determinada facción para un determinado reino. En mi caso, el reino en el que juego se declina claramente por la Horda, habiendo una superioridad de personajes de esta facción muy evidente. Además, en las dos razas que son multi-facción podemos ver como la barra de personajes de la Alianza es muy pequeña o prácticamente imperceptible en el caso de los Pandaren.

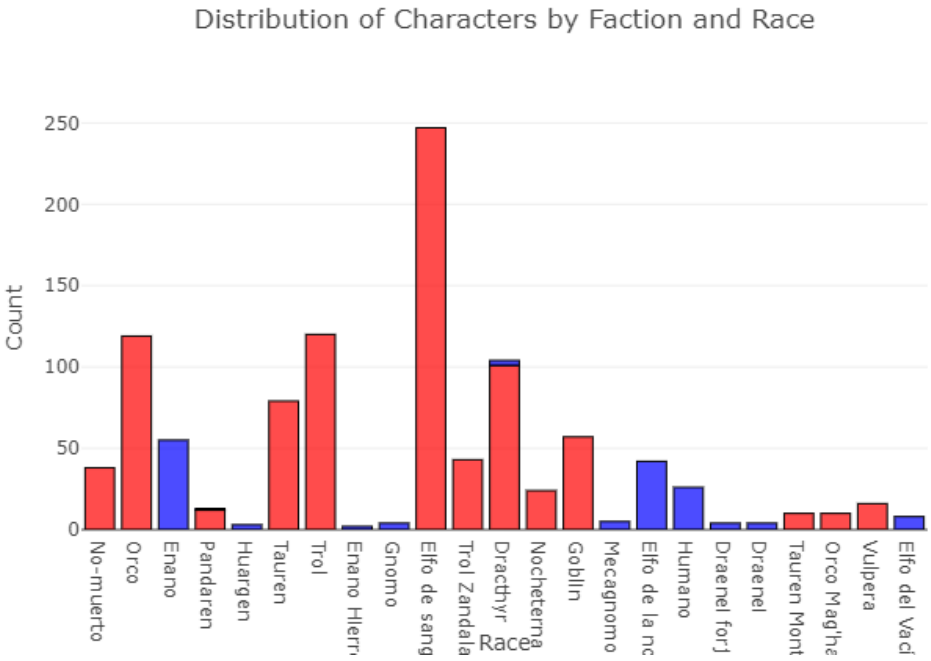


Figura 4.3: Distribución de razas por facción.

4.1.4. Top personajes por clase con más puntos de logros

Este gráfico, si lo relacionamos con el segundo 4.1.2, podemos llegar a las mismas conclusiones y reafirmarnos. Aquí vemos las tres clases con mayor media de nivel de objeto que son las que mencionamos en el otro gráfico: Sacerdote y Cazador, por tener mejor estadísticas, y Evocador, por ser la clase de moda al ser la más nueva.

Sin embargo, cuando escogemos a los dos personajes con mayor cantidad de puntos de logros vemos una diferencia muy evidente entre los Cazadores y los Sacerdotes frente a los Evocadores. Esto se debe a que los puntos de logros se van consiguiendo con el transcurso del tiempo y a lo largo del desarrollo del personaje, cuando completa determinadas actividades y proezas. Por otro lado, tener un nivel de objeto alto va relacionado con ser bastante activo dentro del juego en los últimos meses. Es por eso que estas tres clases tienen un nivel de objeto similar, porque esos personajes serán muy activos últimamente. Pero, como los Evocadores tienen poca trayectoria dentro del juego por ser tan nuevos, vemos una diferencia muy grande en puntos de logros frente a los personajes de las otras dos clases, que probablemente tengan muchísimo más tiempo de juego.

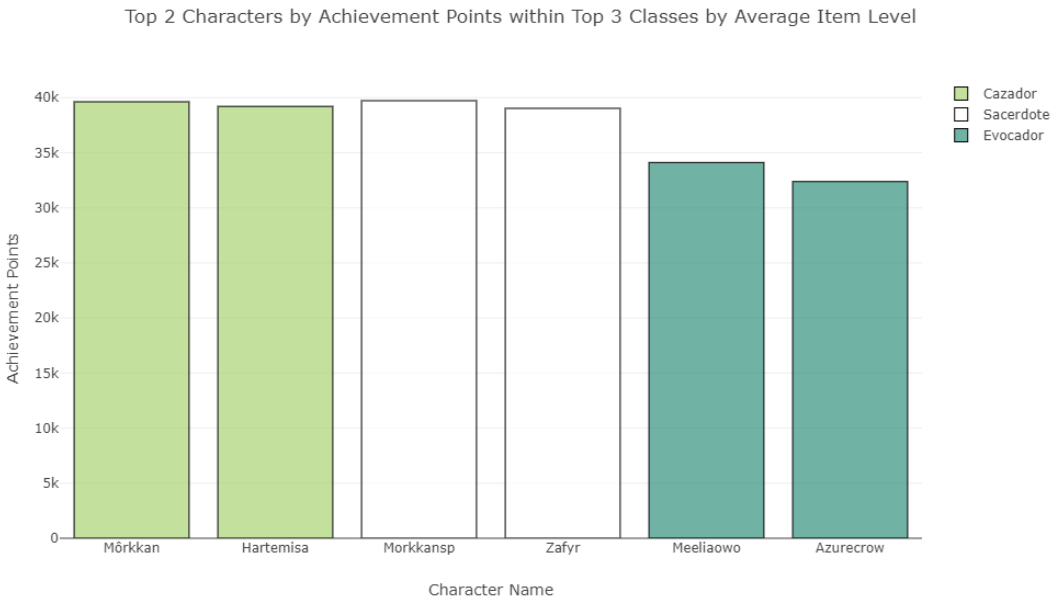


Figura 4.4: Top personajes por clase con más puntos de logros.

4.1.5. Top guilds con mayor cantidad de personajes al nivel 70 y su promedio de item level

Las Hermandades o Guilds en World of Warcraft son asociaciones de jugadores dentro del propio juego, donde la gente se junta bajo una misma bandera para completar actividades y jugar juntos. Hay Hermandades con todo tipo de propósitos: desde hermandades "hardcore"que pretenden completar las actividades más difíciles del juego y por ello han de tener el máximo nivel de objeto posible; Hermandades más casual donde se hace todo tipo de contenido; y por último Hermandades de rol, donde la gente interpreta a sus personajes con el propósito de contar una historia y no tanto completar actividades como las otras.

En este gráfico podemos ver como las cinco Hermandades que observamos podrían considerarse "hardcore", ya que todas ellas tienen un nivel de objeto bastante similar y bastante alto. También podemos ver diferencias entre Hermandades como **My Bad Supongo** y **Unchained**, donde a pesar de que la primera tiene más miembros al nivel 70, el promedio de nivel de objeto de la segunda es bastante más alto, lo que nos puede hacer pensar que estos segundos son más selectivos a la hora de escoger a sus participantes.

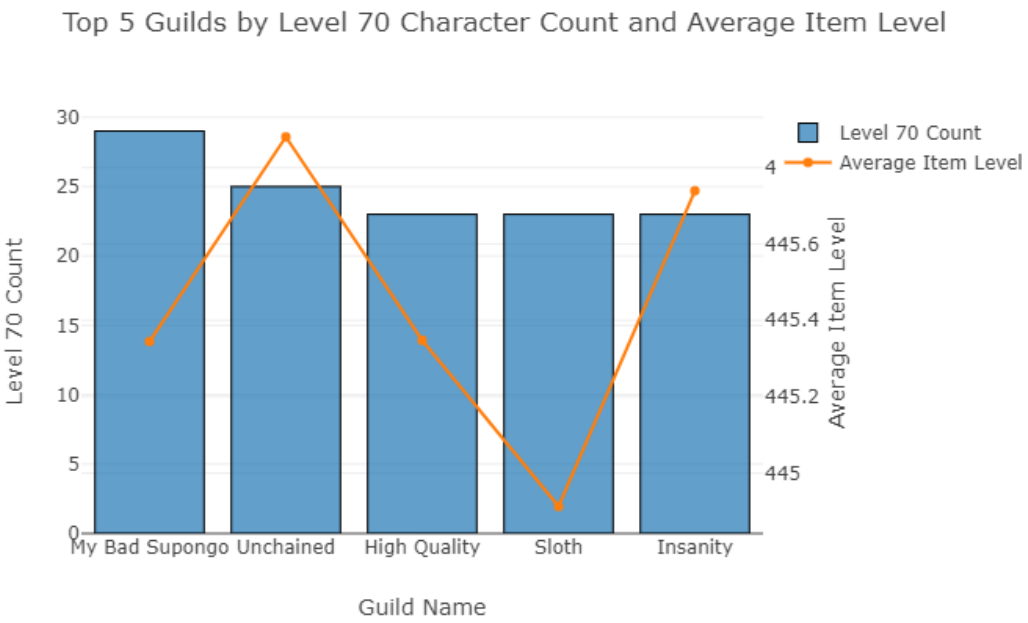


Figura 4.5: Top guilds con mayor cantidad de personajes al nivel 70 y su promedio de item level.

4.2. Rendimiento

Para medir el rendimiento de la ejecución de las queries he dividido la ejecución en dos etapas: una ejecución en una máquina local y una en el cluster de EMR que creamos en AWS. Ambas ejecuciones recogen las mismas métricas: tiempos de carga del dataset tanto en formato **json** como en formato **parquet** y tiempos de ejecución de todas las queries tanto en formato **Dataset** como en formato **Dataframe**.

Comenzando con la ejecución en local, se utilizó un MacBook Pro equipado con 16GB de memoria RAM y el procesador Apple M1 de arquitectura ARM, que cuenta con 8 núcleos, divididos en 4 núcleos de alto rendimiento y 4 núcleos de eficiencia energética. Esta arquitectura híbrida permite optimizar el uso de recursos según las demandas de procesamiento, maximizando el rendimiento en tareas intensivas y reduciendo el consumo energético en operaciones más ligeras.

En cuanto a los resultados, vemos una primera diferencia muy grande a la hora de cargar el dataset, ya que la carga del dataset en formato **json** tardó un total de **5560 ms** mientras que la carga en formato **parquet** tardó sólo **856 ms**. Donde se ve que para un mismo tamaño de dataset es muchísimo más eficiente utilizar el formato **parquet**.

Respecto a la ejecución de las queries se ve también que el formato **Dataset** es mucho más eficiente que el formato **Dataframe**, donde hemos visto como los tiempos se reducen a la mitad en muchos casos. Aquí tenemos algunos ejemplos:

- **Promedio de item level por cada clase**:
  - **Dataframe**: 1106 ms
  - **Dataset**: 511 ms
- **Distribución de razas por facción**:
  - **Dataframe**: 774 ms
  - **Dataset**: 321 ms
- **Top personajes por clase con más puntos de logros**:
  - **Dataframe**: 1864 ms
  - **Dataset**: 792 ms

En cuanto a la ejecución en el clúster de EMR, se configuró con tres instancias de tipo **m4.large**, una para el nodo maestro (principal), otra para el nodo principal (central) y una más como nodo de tarea. Cada una de estas instancias cuenta con 2 CPUs virtuales y 8 GB de memoria. El nodo maestro se encarga de

gestionar el estado del cluster y coordinar las tareas, mientras que el nodo principal (central) realiza la mayor parte del procesamiento de datos distribuido y almacenamiento intermedio. El nodo de tarea se utiliza específicamente para manejar tareas adicionales y cargas de trabajo específicas, aumentando la capacidad de procesamiento en momentos críticos.

En los resultados vemos una mejora significativa en los tiempos de carga y ejecución. Al igual que en la ejecución local, la carga del dataset en formato **parquet** es mucho más rápida que en formato **json**, con una diferencia notable. La carga en **json** tomó **2261 ms**, mucho más rápida que la carga en local, mientras que la carga en **parquet** fue ligeramente más rápida que en local, tomando tan solo **748 ms**, aunque se mantiene más rápida que en formato **json**.

En cuanto a la ejecución en el cluster de EMR vemos mejoras en los tiempos de ejecución de las queries con respecto a la máquina local. Aquí tenemos algunos ejemplos y una tabla comparativa:

- **Promedio de item level por cada clase**:
  - **Dataframe**: 388 ms
  - **Dataset**: 124 ms
- **Distribución de razas por facción**:
  - **Dataframe**: 192 ms
  - **Dataset**: 160 ms
- **Top personajes por clase con más puntos de logros**:
  - **Dataframe**: 203 ms
  - **Dataset**: 153 ms

Query	Local (Dataframe)	EMR (Dataframe)	Local (Dataset)	EMR (Dataset)
Promedio de item level por clase	1106 ms	388 ms	511 ms	124 ms
Diferencia	718 ms		387 ms	
Distribución de razas por facción	774 ms	192 ms	321 ms	160 ms
Diferencia	582 ms		161 ms	
Top personajes con más logros	1864 ms	203 ms	792 ms	153 ms
Diferencia	1661 ms		639 ms	

Tabla 4.1: Comparación de tiempos entre ejecución local y en EMR (AWS)

Como podemos observar, hay una mejora bastante considerable a la hora de lanzar queries con el formato **Dataframe** dentro del cluster, consiguiendo diferencias de hasta **1661 ms** en una de las queries. Por otro lado, el formato

Dataset ha unos rendimientos ligeramente mejores, pero no es tan grande la diferencia. Esto se debe principalmente a dos factores clave: la optimización interna de Spark y el proceso de serialización.

Por un lado Spark aplica optimizaciones más avanzadas en los **Dataframes** lo que hace que su ejecución sea más eficiente, especialmente en un entorno como EMR, donde puede aprovechar mejor los recursos del clúster.

Por otro lado, los **Datasets** requieren más trabajo de serialización y deserialización de los datos. Este proceso de serialización introduce una sobrecarga en un entorno distribuido, lo que hace que los tiempos sean más elevados en EMR, pues en local este coste es menor.

5

Conclusiones

En lo referente a esta segunda fase del proyecto estoy muy contento con los resultados obtenidos y el proceso para llegar a ellos. Partiendo de que la temática me resulta muy atractiva, el análisis llevado a cabo ha sido suficientemente claro y comprensible como para alcanzar unos resultados muy interesantes. Además, gracias a Plotly ha sido más fácil entender los resultados a la hora de analizarlos. Finalmente el despliegue en AWS ha servido como un gran acercamiento a los servicios cloud distribuidos y han ejemplificado perfectamente sus ventajas frente a entornos locales.

Si analizamos los objetivos específicos que planteamos al inicio del proyecto, podemos hacer los siguientes apuntes:

- **Construcción del dataset**: La creación del dataset fue uno de los primeros pasos críticos y se llevó a cabo con éxito. Gracias a la API pública de World of Warcraft y la ayuda de la página **WoWProgress**, logramos recolectar una gran cantidad de datos relevantes de mi reino. Además, la transformación de los datos a formato **parquet** optimizó significativamente el rendimiento para su procesamiento en Spark, permitiendo tiempos de lectura y escritura más eficientes.
- **Diseño de queries**: El diseño de consultas para extraer información relevante supuso el punto principal de todo el proyecto. Las consultas implementadas, como el promedio de item level por clase o la distribución de razas por facción, dieron lugar a resultados muy valiosos e interesantes a la hora de analizar el funcionamiento del juego y sobretodo el comportamien-

to de los jugadores. También el hecho de poder explorar Spark SQL para compararlo con las funciones de Spark ha sido muy entretenido y un gran aprendizaje.

- **Visualización de queries**: La visualización de los resultados de las consultas utilizando **Plotly Scala** ha sido sin duda una clave fundamental para interpretar los datos de manera intuitiva. Los gráficos generados facilitaron el análisis de los datos, mostrando patrones claros sobre los personajes y las clases dentro del juego, además de los propios jugadores.

- **Despliegue en AWS**: La migración del proyecto a un entorno distribuido en la nube utilizando AWS y su servicio de **EMR** fue la parte más interesante de todo el proyecto, además de que ser un tema muy interesante y una gran tendencia en el mundo de Big Data actual, significó un paso importante para demostrar la escalabilidad y potencia del sistema. A pesar de tenber que enfrentar algunas dificultades en la configuración inicial del clúster, el rendimiento fue notablemente superior en comparación con la ejecución en local. Este despliegue nos permitió comprobar la capacidad del sistema para manejar grandes volúmenes de datos y obtener métricas de rendimiento útiles para evaluar la eficiencia del procesamiento en entornos distribuidos.

Finalmente, la utilización de Apache Spark en este proyecto ha demostrado ser una herramienta potente y flexible para el manejo de Big Data. Combinado con AWS, hemos sido capaces de escalar el procesamiento de los datos de World of Warcraft, lo que abre la puerta a análisis mucho más profundos y a la posibilidad de realizar estudios más complejos en el futuro. Sin embargo, existen áreas de mejora que explico en el apartado 5.1 para perfeccionar y ampliar el alcance del proyecto.

5.1. Líneas futuras

- **Dataset**: Incrementar el tamaño del dataset puede hacer que este proyecto sea más interesante. Desde aumentar la cantidad de personajes que aparecen en el dataset hasta tomar datos de otros reinos o incluso otras regiones puede llegar a dar mucha información relevante que complete la que ya hemos conseguido.

- **Consultas**: Podemos añadir más consultas si queremos expandir el análisis y explorar otras áreas del dataset. También podemos hacer consultas algo más complejas que podrían añadir detalles a las ya existentes que completen la información que queremos analizar.

- **Despliegue AWS**: Sería interesante ver las diferencias en rendimiento que pudiera haber si desplegasemos la aplicación en otros Cloud diferentes a AWS, como pueda ser Microsoft Azure o Google Cloud.

Bibliografía

[1] Blizzard Entertainment, "World of warcraft api documentation," 2024. [Online]. Available: https://develop.battle.net/documentation/

[2] Apache Software Foundation, "Apache spark," 2024. [Online]. Available: https://spark.apache.org/

[3] B. Chambers and M. Zaharia, *Spark: The Definitive Guide: Big Data Processing Made Simple*, 1st ed. Sebastopol, CA: O'Reilly Media, 2018.

[4] Amazon Web Services, "Aws documentation," 2024. [Online]. Available: https://docs.aws.amazon.com/

[5] Plotly, "Plotly scala," 2024. [Online]. Available: https://plotly.com/scala/

[6] Scala, "Scala," 2024. [Online]. Available: https://www.scala-lang.org/

[7] JetBrains, "Intellij idea," 2024. [Online]. Available: https://www.jetbrains.com/idea/

[8] Microsoft, "Visual studio code," 2024. [Online]. Available: https://code.visualstudio.com/

[9] Software Freedom Conservancy, "Git," 2024. [Online]. Available: https://git-scm.com/

[10] Microsoft, "Windows," 2024. [Online]. Available: https://support.microsoft.com/

[11] Apple, "Macos," 2024. [Online]. Available: https://support.apple.com/macos

[12] Project Jupyter, "Jupyter," 2024. [Online]. Available: https://jupyter.org/documentation

[13] Docker Inc., "Docker," 2024. [Online]. Available: https://docs.docker.com/

[14] Almond, "Almond," 2024. [Online]. Available: https://almond.sh/

[15] Oracle, "Java," 2024. [Online]. Available: https://www.java.com/es/

[16] Scala, "sbt," 2024. [Online]. Available: https://www.scala-sbt.org/

[17] Apache Software Foundation, "Spark sql," 2024. [Online]. Available: https://spark.apache.org/sql/

[18] Raúl Velasco Rubio, "wow-api," 2024. [Online]. Available: https://github.com/raulvr2/wow-api

[19] WOWPROGRESS LLC, "Wowprogress," 2024. [Online]. Available: https://www.wowprogress.com/gearscore/eu/connected-sanguino

[20] Wowpedia, "Wowpedia," 2024. [Online]. Available: https://wowpedia.fandom.com/wiki/Class_colors

[21] Juan Manuel Serrano, "spark-intro," 2020. [Online]. Available: https://github.com/jserranohidalgo/spark-intro