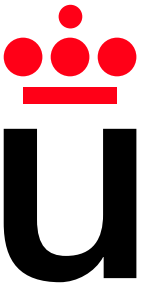


Universidad
Rey Juan Carlos

Escuela Técnica Superior
de Ingeniería Informática

Grado en Ingeniería de Computadores

Curso 2021-2022

Trabajo Fin de Grado

COMPARATIVA ENTRE LAS API DE SPARK EN SCALA Y PYTHON

Autor: Oscar Nydza Nicpoñ
Tutor: Juan Manuel Serrano Hidalgo

Agradecimientos

Quiero dedicar este trabajo a todo el mundo que ha confiado en mí a lo largo de los años y que me ha apoyado en los momento más difíciles. En especial a mi madre, que ha demostrado tener más paciencia que nadie que haya conocido.

Resumen

En este TFG vamos a ver cómo programar queries o peticiones a una fuente de datos local, ya sea en formato CSV o Parquet, usando Spark tanto en su API de Scala como de Python. Veremos los detalles que diferencian sintácticamente ambas API y cómo podríamos migrar peticiones de la API de Scala a la de Python. Tras esto, al ver que resulta relativamente arduo hacerlo a mano, desarrollaremos una serie de expresiones regulares con su respectiva expresión de sustitución que nos ayudará en el proceso de migración, y trataremos de automatizarlo a través de un *script*.

Veremos también los resultados de nuestras peticiones e intentaremos sacar conclusiones de ellas. Para ello usaremos la librería Plotly, que nos generará gráficos para ayudar visualmente a la tarea.

Tras esto pasaremos a comparar el rendimiento de ambas API, y veremos cómo el conjunto de datos utilizado no fue el más adecuado. Habiendo justificado el por qué, se realizará una última petición a un conjunto de datos que encaja mejor con la librería que estamos usando. Veremos el rendimiento en tareas necesarias para ejecutar las peticiones, tiempos de tarea y uso de memoria. Además, se comparará el rendimiento entre formatos de entrada: usaremos ficheros CSV, Parquet sin particionar y Parquet particionado para simular cierto paralelismo.

Todo el material desarrollado para este TFG está disponible en el repositorio disponible en el apéndice D.

Palabras clave:

- Spark
- Python
- Scala

Índice de contenidos

Índice de figuras

1
Introducción

Spark está definido como "un motor multilenguaje para ejecutar tareas de ingeniería de datos, ciencia de datos y *machine learning* en máquinas con un solo nodo o en *clusters*"[1]. En resumen, se utiliza en entornos de *big data*, ya que el paralelismo que ofrece otorga enormes ventajas que iremos descubriendo.

Mediante el uso de APIs disponibles en distintos lenguajes es posible programar peticiones a fuentes de datos de forma eficiente y relativamente sencilla, ya que tiene una sintaxis que resulta familiar si se ha trabajado lenguajes de consultas como SQL. De hecho, es posible utilizar únicamente este estilo de programación. Sin embargo, también proporciona otros métodos para realizar este procesado de datos mediante el uso de funciones como si de un lenguaje funcional se tratara.

A la hora de desplegar un entorno que use Spark como motor para el procesado de datos, nos puede surgir la duda de cuál de las distintas API podemos utilizar. A simple vista puede resultar complicado: dejando de lado la API de SQL, tenemos APIs disponibles en Scala, Python, Java y R. Sería interesante tener información sobre su rendimiento antes de tomar una decisión que puede costar mucho dinero a nivel de empresa.

Es por esto último que nos vamos a centrar en dos APIs: Scala Spark y PySpark. Se han elegido expresamente dado que una es "nativa" y otra es "no nativa", las dos categorías de APIs disponibles. Por un lado como nativas tenemos a la de Java y Scala y como no nativas a la de Python y R.

Este trabajo consistirá en intentar establecer una serie de comparativas y tratar de llegar a una conclusión lo más clara posible sobre cuál de estas dos es la más eficiente dentro del entorno limitado del que dispondremos.

[1] `https://spark.apache.org/`

2

Objetivos

El principal objetivo de este Trabajo de Fin de Grado realizar una comparativa entre las API de Spark de Scala y de Python. Al ser un objetivo amplio, vamos a especificar objetivos más pequeños y concretos:

Objetivo 1

Vamos a utilizar un conjunto de datos del dominio de la Fórmula 1 ya que tenemos varias preguntas que necesitan respuesta. Estas son:

- Cuál ha sido el piloto más consistente en la temporada 2012. Se calculará la diferencia entre el tiempo medio de todas las vueltas de cada piloto ese periodo de tiempo en concreto y la media de sus vueltas más rápidas.
- Cuál ha sido el mejor piloto de la historia calculando valores estadísticos como el total de carreras ganadas, el total de títulos, el número de vueltas lideradas, el número de primeras posiciones en clasificación, número de vueltas rápidas, etc. Todo ello relativo a su periodo de actividad.
- Cuál ha sido el fabricante más dominante en un periodo de tiempo concreto.
- Análisis de temporada por pilotos: se calcularán diversas medidas estadísticas para cada piloto. Por ejemplo, el total de podios, el porcentaje de carreras en las que se ha acabado en podio, la media de posiciones perdidas y ganadas por carrera, el número de vueltas lideradas, etc.
- Cuál ha sido la temporada más interesante para el espectador, teniendo en cuenta métricas como el número de adelantamientos, más cambios de líder en la clasificación general, etc.

Para responder a estas cuestiones tendremos que realizar una serie de queries utilizando ambas API, aprendiendo en el camino cómo programarlas y viendo las diferencias y similitudes entre ambas. Para ello usé [CZ18], [Spa] y [PyS].

Objetivo 2

Sería también útil encontrar una manera de poder visualizar estos resultados obtenidos de forma gráfica. Para ello usaremos la librería Plotly, que, como ya veremos, nos proporciona una manera sencilla de obtener gráficos a partir de datos.

Objetivo 3

Buscaremos la manera de migrar queries programadas en Scala Spark a PySpark. Esto nos permitirá ver mejor las diferencias entre las API a la hora de programarlas. Intentaremos buscar también la manera de automatizar este proceso.

Objetivo 4

Comparar el rendimiento de ambas API a nivel de tiempos de ejecución y uso de recursos. Utilizaremos la Spark UI para obtener estas métricas y veremos cuál se comporta mejor en un entorno lo más igualado posible.

2.1. Metodología de trabajo

En esta sección vamos a ver las herramientas que se van a usar, ya que se puede considerar un objetivo secundario el aprendizaje de las mismas.

Lo primero que tendremos que aprender será a programar funciones sencillas en Scala. Ya que es un lenguaje que no se enseñó en Ingeniería de Computadores durante los años en los que fui alumno, considero importante obtener un nivel básico de conocimientos sobre su uso antes de lanzarse de lleno a usar su API de Spark. Para aprender usé [OSVS19] y [Sca]

Lo mismo podríamos decir de Python, sin embargo este lenguaje sí se cursó en la asignatura Lenguajes de Programación y se obtuvo un nivel suficiente como usuario, así que no lo veo como un objetivo.

Tendremos que aprender también a utilizar cuadernos Jupyter para desarrollar código tanto en Scala como Python, ya que nos permitirá avanzar más rápidamente en las fases iniciales del proyecto al facilitar la depuración de las queries que se van a programar.

A la hora de migrar queries usaremos expresiones regulares de búsqueda y sustitución. En concreto para ello se usará tanto `sed` como Perl y recursos en línea para el desarrollo de nuestras expresiones. Usaremos dos estándares: POSIX ERE y PCRE2. Después se realizará un *script* que aplicará todas nuestras expresiones secuencialmente.

Para describir la fuente de datos usaremos DataGrip de JetBrains, que nos permitirá obtener diagramas Entidad-Relación de nuestro conjunto de datos de forma más sencilla.

También usaremos Git como sistema de control de versiones tanto para todo el código desarrollado como para esta misma memoria, que se va a escribir usando LaTeX. Como referencia usé [CS14]

Por último, será necesario aprender LaTeXpara desarrollar esta misma memoria.

3

Descripción Informática

En este capítulo se va a abordar la parte más técnica del proyecto.

En primer lugar veremos una descripción detallada de los datos de los que disponemos, proporcionando ejemplos y un diagrama Entidad-Relación en el que podremos ver cómo están relacionados los distintos ficheros de entrada.

Luego pasaremos a centrarnos en la programación de queries: primero en Scala Spark y luego en PySpark. Se describirá a grandes rasgos qué queremos conseguir y el por qué. Tras esto, nos centraremos en el proceso de construcción de queries desde un plano más técnico. En concreto, veremos cómo se programa una query en cada API, dejando el resto para los apéndices.

Por último veremos el proceso de migración de queries desde Scala a Python, centrándonos en las diferencias entre APIs y trataremos de facilitar lo máximo este proceso mediante el desarrollo de una serie de expresiones regulares y un *script* para automatizarlo.

3.1. Fuentes de datos

En esta sección vamos a ver en profundidad cómo es el conjunto de datos que se ha escogido para la realización de este proyecto. Veremos también varios formatos: CSV y Parquet. Nos centraremos en las diferencias entre ellos y pondremos el punto de mira en cómo podríamos cambiar del primero al segundo, en teoría más eficiente.

3.1.1. Formato CSV

Como se mencionó brevemente en el apartado de Objetivos, se ha utilizado un conjunto de datos de la Fórmula 1 que fue obtenido del enlace disponible en la bibliografía [Rao20]. Concretamente, este conjunto de datos tiene 13 tablas que proporcionan información sobre distintos aspectos de esta competición. Estas tablas son:

- `circuits`
- `constructor_results`
- `constructor_standings`
- `constructors`
- `driver_standings`
- `lap_times`
- `pit_stops`
- `qualifying`
- `races`
- `results`

- `seasons`
- `status`
- `drivers`

Todas estas tablas están interrelacionadas como se puede ver en el diagrama Entidad-Relación que se presenta a continuación:

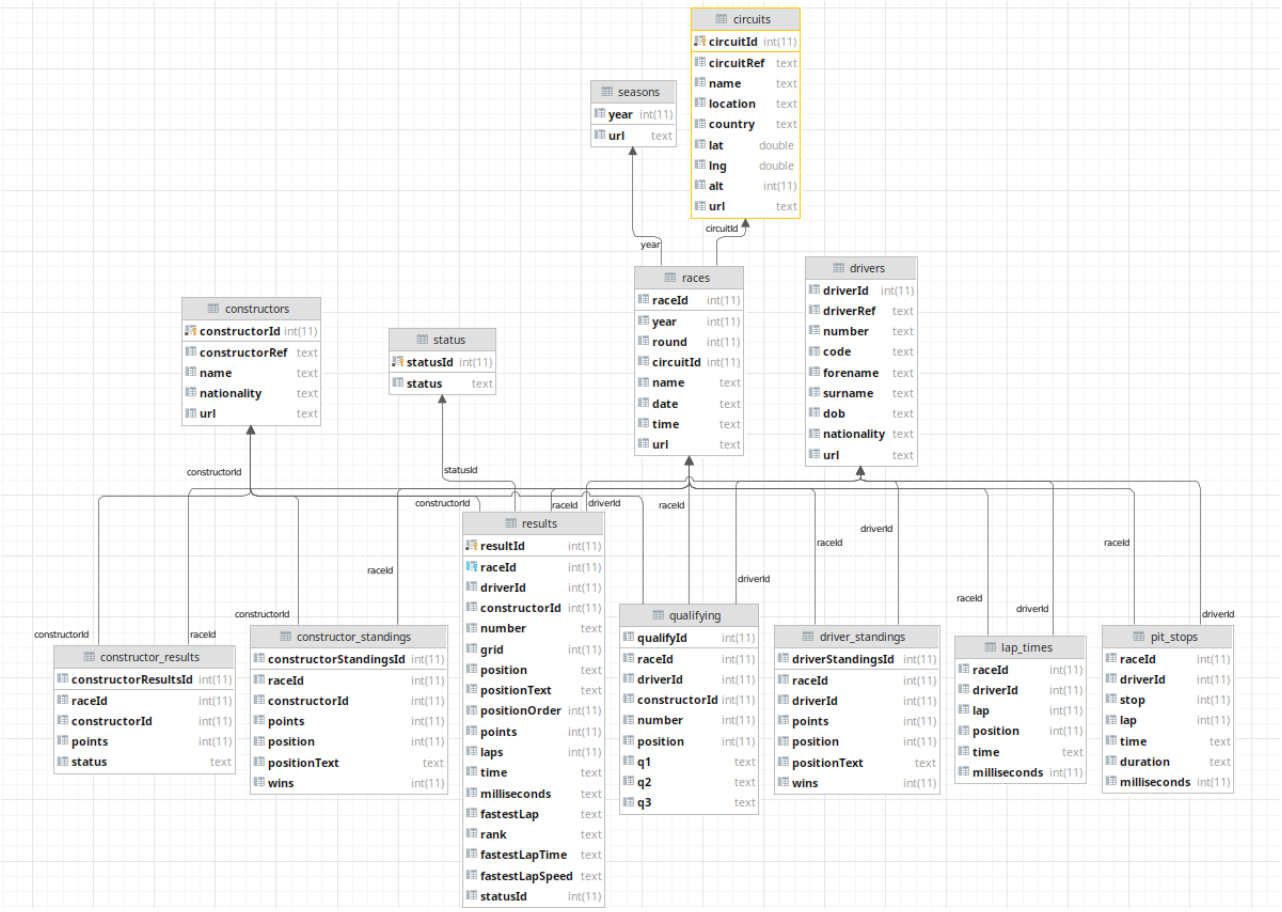


Figura 3.1: Diagrama Entidad-Relación

Tabla `circuits`

Esta tabla contiene información sobre todos los circuitos en los que se ha llevado a cabo un Gran Premio. Las columnas más interesantes son el nombre del circuito, una referencia textual y la localización.

circuitId	circuitRef	name	location	country	lat	lng	alt	url
1	albert_park	Albert Park Grand...	Melbourne	Australia	-37.8497	144.968	10	http://en.wikiped...
2	sepang	Sepang Internatio...	Kuala Lumpur	Malaysia	2.76083	101.738	18	http://en.wikiped...

Figura 3.2: Tabla circuits

Tabla `constructor_results`

Esta tabla nos proporciona información sobre los resultados de las carreras en base a los constructores.

constructorResultsId	raceId	constructorId	points	status
1	18	1	14	\N
2	18	2	8	\N

Figura 3.3: Tabla constructor_results

Tabla `constructor_standings`

Esta tabla contiene información sobre la clasificación de constructores. Como particularidad, tiene una entrada por carrera y constructor participante. Por tanto, podríamos ver cómo ha ido cambiando la clasificación de constructores a lo largo del campeonato.

Las columnas más interesantes son el identificador de la carrera, identificador del constructor, los puntos, la posición en la clasificación y las victorias hasta ese punto.

constructorStandingsId	raceId	constructorId	points	position	positionText	wins
1	18	1	14	1	1	1
2	18	2	8	3	3	0

Figura 3.4: Tabla constructor_standings

Tabla `constructors`

Esta tabla contiene información sobre los distintos constructores que han participado en algún campeonato mundial de Fórmula 1. Las columnas más interesantes son el id de constructor, la referencia, el nombre del constructor y la nacionalidad.

constructorId	constructorRef	name	nationality	url
1	mclaren	McLaren	British	http://en.wikiped...
2	bmw_sauber	BMW Sauber	German	http://en.wikiped...

Figura 3.5: Tabla constructors

Tabla `driver_standings`

Similar a la tabla de clasificación de constructores, pero para pilotos. Tenemos las mismas columnas, salvo que en lugar de tener un id de constructor, lo tenemos de piloto.

driverStandingsId	raceId	driverId	points	position	positionText	wins
1	18	1	10	1	1	1
2	18	2	8	2	2	0

Figura 3.6: Tabla driver_standings

Tabla `lap_times`

Esta tabla es una de las más interesantes, ya que nos da todos los tiempos de vuelta de todos los pilotos desde que hay registros. Esto es, desde parte de 1996 y 1997 al completo.

Las columnas más llamativas podrían ser el id de carrera, el de piloto, la vuelta en cuestión, la posición y el tiempo en milisegundos.

raceId	driverId	lap	position	time	milliseconds
841	20	1	1	1:38.109	98109
841	20	2	1	1:33.006	93006

Figura 3.7: Tabla lap_times

Tabla `pit_stops`

Esta tabla contiene información de las paradas en boxes. Las columnas más interesantes son los id de carrera y piloto, el índice de parada (si es la primera, segunda, etc), la vuelta en la que se hace y la duración en milisegundos.

raceId	driverId	stop	lap	time	duration	milliseconds
841	153	1	1	17:05:23	26.898	26898
841	30	1	1	17:05:52	25.021	25021

Figura 3.8: Tabla pit_stops

Tabla `qualifying`

Esta tabla nos da información sobre los resultados de todas las rondas de clasificación. La columnas más interesantes son la posición final y los tiempos en Q1, Q2 y Q3.

qualifyId	raceId	driverId	constructorId	number	position	q1	q2	q3
1	18	1	1	22	1	1:26.572	1:25.187	1:26.714
2	18	9	2	4	2	1:26.103	1:25.315	1:26.869

Figura 3.9: Tabla qualifying

Tabla `races`

Esta tabla contiene información sobre todas las carreras celebradas en la historia de la competición. Contiene columnas como el id del circuito, el nombre del Gran Premio, la fecha y el año en el que se celebró. Esta última quizá sea la más útil de todo el dataset, ya que es la única forma de filtrar las carreras o los resultados por temporada.

raceId	year	round	circuitId	name	date	time	url
1	2009	1	1	Australian Grand ...	2009-03-29	06:00:00	http://en.wikiped...
2	2009	2	2	Malaysian Grand Prix	2009-04-05	09:00:00	http://en.wikiped...

Figura 3.10: Tabla races

Tabla `results`

Esta tabla es similar a la de resultados por constructor, pero para pilotos. Es la tabla más completa de todas, ya que nos proporciona una entrada por piloto y carrera con información relevante de cómo se ha desarrollado la misma. Las columnas más interesantes pueden ser la posición de salida y la posición final, los puntos, las vueltas dadas, la vuelta más rápida, la velocidad más rápida y, en el caso de que haya habido algún incidente, el id del estado.

resultId	raceId	driverId	constructorId	number	grid	position	positionText	positionOrder	points	laps	time	milliseconds	fastestLap	rank	fastestLapTime	fastestLapSpeed	statusId
1	18	1	1	22	1	1	1	1	10	58	1:34:50.616	5690616	39	2	1:27.452	218.300	1
2	18	2	2	3	5	2	2	2	8	58	+5.478	5696094	41	3	1:27.739	217.586	1

Figura 3.11: Tabla results

Tabla `seasons`

Quizá se trate de la tabla menos útil, ya que solamente contiene una columna con el año y otra con una url a un artículo de Wikipedia para cada entrada.

year	url
2009	https://en.wikipe...
2008	https://en.wikipe...

Figura 3.12: Tabla seasons

Tabla `status`

Esta tabla nos da información sobre los estados en los que ha podido acabar la carrera un piloto determinado. Contiene un identificador y el estado en cuestión.

statusId	status
1	Finished
2	Disqualified

Figura 3.13: Tabla status

Tabla `drivers`

Contiene información sobre todos los pilotos que han competido a lo largo de la historia. En concreto la información más relevante puede ser el nombre y apellido, el código, la fecha de nacimiento y la nacionalidad.

driverId	driverRef	number	code	forename	surname	dob	nationality	url
1	hamilton	44	HAM	Lewis	Hamilton	1985-01-07	British	http://en.wikiped...
2	heidfeld	\N	HEI	Nick	Heidfeld	1977-05-10	German	http://en.wikiped...

Figura 3.14: Tabla drivers

Tabla `drivers constructor season`

Esta tabla no estaba originalmente en el conjunto de datos, pero resultó necesario establecer una relación entre cada piloto con su constructor en cada temporada. Principalmente se necesita para poder hacer comparativas entre pilotos del mismo equipo o bien de forma global o bien por temporadas.

Esta tabla se creó a partir de la tabla `races`, que contiene la temporada y la tabla `results`, que contiene tanto el constructor como el piloto. Se hizo la intersección de estas tablas mediante la columna identificadora de la carrera. El código es el siguiente:

```
val raceSeasonMap = spark.read.format("csv")
  .option("header", "true")
  .option("sep", ",")
  .load("../data/races.csv")
  .select("raceId", "year")

spark.read.format("csv")
  .option("header", "true")
  .option("sep", ",")
  .load("../data/results.csv")
  .join(raceSeasonMap, Seq("raceId"), "left")
  .select("year", "driverId", "constructorId")
  .dropDuplicates()
  .repartition(1)
  .write.format("csv")
  .option("header", "true")
  .save("../data/drivers_constr_season.csv")
```

Para escribir la tabla en disco, primero tenemos que utilizar `repartition` para que el resultado final quede en un solo archivo CSV. Después especificamos el formato y si queremos las cabeceras o no, y proporcionamos el directorio donde queremos que quede guardado.

Finalmente la tabla contiene información tal que:

```
+----+--------+-------------+
|year|driverId|constructorId|
+----+--------+-------------+
|2021|     846|            1|
|2021|     817|            1|
+----+--------+-------------+
```

Figura 3.15: Tabla auxiliar piloto-constructor-temporada

3.1.2. Formato Parquet

También utilizaremos el formato de ficheros Parquet como alternativa a CSV ya que nos puede otorgar beneficios a la hora de ejecutar nuestras queries, mejorando el rendimiento tanto en tiempos de ejecución como en tamaño de datos de entrada leídos.

Parquet es un formato que, además de los propios datos, almacena metadatos y realiza ciertos indexados y optimizaciones para que se ocupe el menor espacio posible en disco. Además, podemos particionar la tabla según ciertas columnas clave, permitiendo que Spark, tras hacer sus optimizaciones propias del proceso, sepa qué archivos tiene que leer y cuáles no. Un ejemplo de cómo se ve una fuente de datos de entrada en disco puede ser el siguiente:

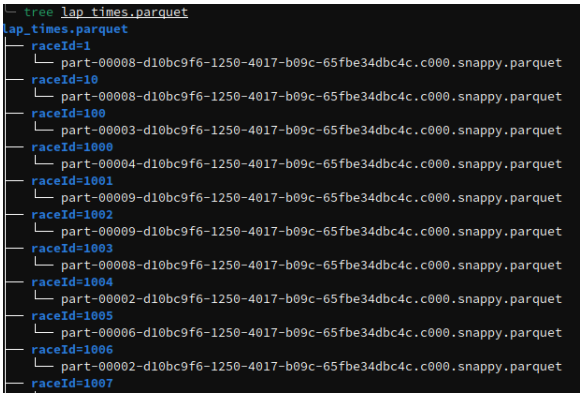


Figura 3.16: Archivos Parquet en disco

Podemos ver que la tabla en cuestión está particionada según el campo `raceId`, y que tenemos una subcarpeta por cada valor que toma. Dentro de cada subcarpeta está ya el archivo Parquet, que contiene los datos para ese `raceId`. Si tuviéramos más datos, tendríamos más archivos dentro de las subcarpetas.

Teóricamente leer datos que están en formato Parquet usando Spark debería ser más eficiente ya que no se lee la tabla completa, al contrario que con los CSV.

Más adelante se van a comparar tres formatos de entrada: CSV, Parquet particionado y Parquet sin particionar. Originalmente el conjunto de datos venía en formato CSV, así que no necesitamos realizar ningún procesamiento extra para conseguirlo. Sin embargo, los otros dos formatos van a necesitar un poco de trabajo adicional.

Convertir los tipos de entrada no conlleva mucho trabajo a nivel de código, pero puede resultar enrevesado ya que se necesita conocer el conjunto de datos y realizar un análisis previo al particionado, debido a que es posible realizar un particionado "malo" si no se tiene en cuenta cómo van a quedar los datos almacenados en disco.

Para llevar a cabo este particionado se ha desarrollado una función que dado un nombre de tabla y un nombre de columna realiza la conversión y escribe los datos en disco:

```
def toParquet(tableName: String, partitionCol: String): Unit = {
       val ini = spark.read.format("csv")
              .option("header", "true")
```

```
                .option("sep", ",")
                .load(s"../data/$tableName.csv")

        if (partitionCol == "noPartitionCol") {
                ini.write.mode("overwrite")
                        .parquet(s"../data/parquet/$tableName.parquet")
        } else {
                ini.repartition(col(partitionCol))
                        .write.mode("overwrite")
                        .partitionBy(partitionCol)
                        .parquet(s"../data/parquet/$tableName.parquet")
        }
}
```

El detalle que determina el particionado es, en el `else`, las operaciones `repartition` y `partitionBy`.

Por comodidad, se ha creado otra función distinta que dado un nombre de tabla lo escribe como Parquet sin particionar:

```
def toParquetNoPart(tableName: String): Unit = {
        spark.read.format("csv")
                .option("header", "true")
                .option("sep", ",")
                .load(s"../data/$tableName.csv")
                .write.mode("overwrite")
                .parquet(s"../data/parquetnopart/$tableName.parquet")
}
```

Con estas dos funciones definidas, una lista con los nombres de las tablas (`files`) y un mapa que establezca una relación entre nombre de tabla y columna de particionado (`partCol`), podemos convertir todas las tablas de entrada de golpe si ejecutamos lo siguiente:

```
files.foreach(x => toParquet(x, partCol(x)))
```

O, en el caso de que queramos parquet sin particionar:

```
files.foreach(x => toParquetNoPart(x))
```

Una vez tenemos todos los datos como deseamos solamente tendremos que cambiar las líneas en las que se leen los datos de entrada, de forma que si antes teníamos algo como:

```
spark.read.format("csv")
        .option("header", "true")
        .option("sep", ",")
        .load("tablePath")
```

Ahora tenemos algo como:

```
spark.read.parquet("tablePath")
```

3.2. Programación de queries en Scala Spark

En esta sección vamos a ver cómo se programan queries en la API de Scala, veremos principalmente la sintaxis y, al ser el primer apartado en el que se programe una query, se va a explicar detalladamente el proceso de creación de una query para Spark: nos detendremos en las funciones y veremos cómo usarlas para obtener los resultados buscados.

Como comenté anteriormente en forma de introducción a la sección, veremos un ejemplo y dejaremos el otro para el apéndice A.1.

Antes de entrar en materia cabe destacar que, debido a que vamos a usar distintas fuentes de entrada (CSV y Parquet), se va a abstraer la parte de la lectura de las fuentes de datos de la propia query, que se hará en una función que recibirá los `DataFrames` con los datos necesarios previos al procesado.

La función tendrá el siguiente aspecto:

```
def queryEnCuestion(entrada1: DataFrame, entrada2: DataFrame, ...): DataFrame = {
      ...
}
```

Y devolverá un `DataFrame` con los resultados obtenidos para luego poder visualizarlos, escribirlos o cualquier otra operación que veamos pertinente. El uso por tanto será el siguiente:

```
queryEnCuestion(entradaLeida1, entradaLeida2, ...)
```

Es por esto que en la descripción de las queries se va a dar por asumido que tenemos los datos de entrada ya leídos, salvo la primera vez que hagamos una lectura, en cuyo caso se explicará en detalle.

3.2.1. Piloto más consistente en la temporada 2012

En esta query intentaremos averiguar cuál ha sido el piloto más consistente en un periodo de tiempo en concreto, en este caso en la temporada 2012, aunque con mínimos cambios podríamos verlo para cualquier periodo que deseásemos. Ya que este término puede resultar ambiguo, en concreto intentaremos averiguar qué piloto tuvo una menor diferencia entre la media de sus vueltas rápidas y la media de todas las vueltas de todos los Grandes Premios de este periodo de tiempo. Esto nos resultará especialmente interesante teniendo en cuenta que cuanto menor sea esta diferencia, más consistente habrá sido el piloto respecto a su mejor resultado.

Dicho esto, la query tiene el siguiente aspecto:

```
def queryConsistencia(races:DataFrame, lap_times:DataFrame, drivers:DataFrame,
    results:DataFrame):DataFrame = {
      val driverWindow = Window.partitionBy("driverId")
      val seasonWindow = Window.partitionBy("year")


      val races_filtered = races
             .where(col("year") === 2012)

      val avg_lap_times = lap_times
             .withColumnRenamed("time", "lapTime")
             .join(races_filtered, Seq("raceId"), "right")
             .withColumn("milliseconds", col("milliseconds").cast(IntegerType))
             .withColumn("avgMs", avg(col("milliseconds")).over(driverWindow))
             .dropDuplicates("driverId")
             .select("driverId", "avgMs")

      val lapCount = lap_times
             .join(races_filtered, Seq("raceId"), "right")
             .withColumn("lapsPerDriver", count(col("lap")).over(driverWindow))

      val (distinctDrivers, allLaps) = lapCount
             .agg(
                    countDistinct("driverID"),
                    count(col("lap"))
             ).as[(BigInt, BigInt)]
             .collect()(0)

      val avgLapsThisPeriod = allLaps.toInt / distinctDrivers.toInt

      val experiencedDrivers = lapCount
             .where(col("lapsPerDriver") >= avgLapsThisPeriod)
             .select("driverId")
             .distinct()
             .as[String]
             .collect()

      results.join(races, Seq("raceId"), "right")
             .na.drop(Seq("fastestLapTime"))
             .withColumn("fastestLapTimeMs", lapTimeToMsUDF(col("fastestLapTime")))
             .withColumn("avgFastestLapMs", avg(col("fastestLapTimeMs")).over(driverWindow))
             .dropDuplicates("driverId")
             .join(avg_lap_times, Seq("driverId"), "left")
             .withColumn("diffLapTimes", abs('avgMs - 'avgFastestLapMs).cast(IntegerType))
             .withColumn("avgDiff", msToLapTimeUDF(col("diffLapTimes").cast(IntegerType)))
             .where(col("driverId").isInCollection(experiencedDrivers))
             .join(drivers, "driverId")
             .withColumn("driver", concat(col("forename"), lit(" "), col("surname")))
             .select("driver", "avgDiff")
             .orderBy("avgDiff")
}
```

Descripción detallada

A continuación entramos en el detalle de la implementación de esta query.

Primero de todo, queremos leer las fuentes de datos para luego pasar el `DataFrame` a la función ya explicada. Para ello, ejecutamos las siguientes líneas de código:

```
val races = spark.read.format("csv")
  .option("header", "true")
  .option("sep", ",")
  .load("data/races.csv")
```

En este caso leemos la tabla `races`. Como se puede observar, se utilizan un par de opciones de lectura. En nuestro caso, la fuente de datos contiene las cabeceras en la primera línea y cada dato está separado por una coma y por ello tenemos que especificarlo. Por último se proporciona el path relativo de la fuente de datos.

Si quisiéramos leer de un archivo Parquet, la sintaxis queda más simplificada:

```
val races = spark.read.parquet("path_to_races_parquet")
```

Tras esto, ya dentro de nuestra función, se hace el filtro según las temporadas que se quieran usar. Para ello, ya que el periodo sobre el que se quiere obtener datos viene dado como tipo entero y simboliza la temporada, tenemos que convertir la columna `year` a tipo entero, ya que por defecto, al no especificar el esquema a la hora de leer, Spark intenta adivinar los tipos de cada columna. Es posible que detecte esa columna como tipo entero, pero conviene asegurar haciendo la conversión de tipos. Después de esto, llevamos a cabo el filtro. Al final, para obtener este `DataFrame` que utilizaremos más adelante se llevan a cabo las siguientes operaciones:

```
val races_filtered = races
  .withColumn("year", col("year").cast(IntegerType))
  .where(col("year") === 2012)
```

De este trozo de código hay que comentar un par de aspectos. Primero, la conversión de tipos, que se hace al tipo `IntegerType`, y no a `Int`, como sería intuitivo hacer. Esto es porque Spark tiene una serie de tipos concretos para el tipo `Column`. Todos ellos se encuentran en el paquete `org.apache.spark.sql.types`, y es obligatorio su uso si se utiliza la función `cast`. También cabe destacar la función de DataFrame llamada `withColumn`, que se encuentra entre las más usadas, ya que permite añadir una columna al `DataFrame`. Crea una columna con el nombre que recibe como primer parámetro y con el valor que recibe en el segundo. En este caso, ya que la columna `year` ya existe, se sustituye la que había anteriormente con ese nombre.

El otro aspecto a comentar es el propio filtro. Se utiliza la función `where`, que cumple el mismo propósito que su equivalente en SQL. Como parámetro recibe una condición, que en nuestro caso querríamos que fuese que "la columna `year` sea igual al parámetro recibido, en este caso 2012". Para esto podemos hacer una comparativa de la columna en cuestión con el valor deseado como si de un `if` se tratara.

Resumiendo, con estas pocas líneas de código hemos obtenido todas las ca-

rreras celebradas en el rango de temporadas que necesitamos. Más adelante se utilizará para filtrar los resultados de cada piloto y obtener solamente los que nos interesan. Merecía la pena pararse en este trocito de código ya que se repite en todas las queries en las que se requiere centrarse en un periodo concreto de tiempo.

Para realizar esta consulta vamos a necesitar varios `DataFrames` auxiliares además del recién explicado. En concreto, necesitaremos tener una cuenta de todas las vueltas que ha dado cada piloto en el periodo de tiempo establecido, además de la tabla `drivers` para completar la información final.

Para calcular todas las vueltas que ha dado cada piloto usaremos `lap_times`. Le tendremos que aplicar el filtro de temporadas utilizando lo obtenido anteriormente y, por último, se hará el conteo. Todo ello se puede hacer de la siguiente manera:

```
val lapCount = lap_times
  .join(races, Seq("raceId"), "right")
  .withColumn("lapsPerDriver", count(col("lap")).over(driverWindow))
```

Aquí vemos cómo se aplica el filtro de temporadas. Para ello hacemos la operación `join` con el DataFrame `races` obtenido anteriormente, sobre la columna `raceId` y de tipo `right`. En Spark SQL, existen varios tipos de intersecciones (join) que podemos realizar entre dos DataFrames:

- Inner Join.
- Full Outer Join.
- Left Outer Join
- Right Outer Join.
- Left Anti Join.
- Left Semi Join.

Todos ellos definidos de la misma manera que en el Álgebra de Conjuntos.

Para nuestro caso particular, utilizaremos un Right Outer Join, ya que nos queremos quedar con las vueltas de las carreras definidas en `races`.

Tras esto, queremos obtener las vueltas que ha dado cada piloto en ese periodo de tiempo. Para ello, tenemos que utilizar la función `count` sobre la columna `lap`. Sin embargo, nos topamos con que, si hiciéramos eso (aparte de que el compilador no nos dejaría), necesitamos definir una ventana sobre la que operar.

Las ventanas son una parte muy útil de Spark que nos permiten centrarnos en cierta información agrupada de la forma que necesitemos. En nuestro caso, necesitamos contar las vueltas que ha dado cada piloto sin tener en cuenta las del resto y para ello necesitamos definir una ventana nueva (en nuestro caso se podría llamar `driverWindow`) que particione los datos por piloto. Esto lo hacemos de la siguiente manera:

```
val driverWindow = Window.partitionBy("driverId")
```

Utilizando esta ventana, la operación `count` se llevará a cabo un conteo distinto por cada `driverId` que haya. Si particionásemos los datos según varias columnas, se llevaría a cabo la operación en cuestión según cada valor único de esas columnas en conjunto, es decir, si hay alguna variación en alguna de ellas, se toma como una operación distinta.

Este `DataFrame` lo vamos a utilizar para definir cuáles son los pilotos más experimentados de este periodo de tiempo, que diremos que son los que han dado más de la media de vueltas por piloto. Para calcular esto y partiendo del `DataFrame` recién obtenido necesitamos conseguir dos valores: el número total de vueltas dadas entre todos los pilotos y el número de pilotos que han competido en este periodo de tiempo. Lo haremos de la siguiente manera:

```
val (distinctDrivers, allLaps) = lapCount
  .agg(
    countDistinct("driverID"),
    count(col("lap"))
  ).as[(BigInt, BigInt)]
  .collect()(0)
```

Estos valores los obtendré en forma de tupla, en la que el valor de la izquierda será el número de pilotos y el de la derecha el número de vueltas. Cabe centrarse en la operación `agg`, que nos permite obtener un `DataFrame` cuyas columnas tendrán como valor el obtenido de las operaciones que definamos. En este caso, `countDistinct` que, como su nombre indica, cuenta los valores distintos de la columna `driverId` y `count`, que realiza un conteo de todas las entradas de la columna `lap`. Con `as` le definimos el tipo de datos que queremos obtener y con `collect`, obtenemos todos los valores del `DataFrame`. En este caso, como solo vamos a tener una entrada, y esta va a ser la única que necesitemos, hacemos un `collect()(0)`

Para calcular la media de vueltas por piloto en este periodo de tiempo, realizamos la siguiente operación:

```
val avgLapsThisPeriod = allLaps.toInt / distinctDrivers.toInt
```

Con esta métrica podremos definir cuáles son los pilotos más experimentados de la siguiente manera:

```
val experiencedDrivers = lapCount
  .where(col("lapsPerDriver") >= avgLapsThisPeriod)
  .select("driverId")
  .distinct()
  .as[String]
  .collect()
```

Con el `DataFrame` obtenido anteriormente, nos quedamos con los pilotos que tengan un número de vueltas superior o igual al índice calculado. Tras esto, nos quedamos solamente con los valores distintos la columna que indica el piloto y los obtenemos en forma de `List[String]` con las dos últimas operaciones para más adelante poder filtrar según ella.

Tras esto, querremos obtener la media de todas las vueltas que ha dado cada piloto. Para ello, usamos de nuevo la tabla `lap_times`, en la que tenemos una columna llamada `milliseconds` y filtramos las temporadas que nos interesan. Para asegurar, convertimos esta columna a tipo entero y hacemos la media usando la ventana que creamos antes. Eliminamos los pilotos duplicados y nos quedamos con dos columnas: identificador de piloto y la media obtenida. El código queda tal que:

```
val avgLapTimes = lap_times
  .withColumnRenamed("time", "lapTime")
  // filtro las vueltas de las carreras en el periodo de tiempo dado
  .join(races_filtered, Seq("raceId"), "right")
  .withColumn("milliseconds", col("milliseconds").cast(IntegerType))
  // media de tiempos de vuelta por piloto
  .withColumn("avgMs", avg(col("milliseconds")).over(driverWindow))
  .dropDuplicates("driverId")
  .select("driverId", "avgMs")
```

Finalmente, querríamos obtener un `DataFrame` que contenga dos columnas: el nombre del piloto y la diferencia ya mencionada anteriormente. Para ello, necesitamos usar la tabla `results` y dejar fuera las temporadas que no nos interesen. Esto lo haremos como ya hemos comentado antes.

Nos vamos a centrar en una de las columnas que tenemos: `fastestLapTime` que, como su nombre indica, nos da el tiempo de la vuelta más rápida de cada piloto en cada carrera. El problema es que nos lo proporciona en el formato MM:ss:mmm, donde MM son los minutos, ss los segundos y mmm los milisegundos. Necesitamos una forma de convertir esta columna a una unidad con la que podamos operar. Para este caso, lo mejor es convertir el tiempo a milisegundos.

Esta funcionalidad nos la proporcionan las UDFs (User-Defined Functions). La documentación de Spark las define como "rutinas programables por el usuario que actúan fila a fila". Haciendo uso de ellas, podemos convertir una función que realice esta conversión que queremos a una función que actúe de la misma manera para una columna, fila a fila.

En nuestro caso vamos a tener dos funciones de este estilo: una para convertir de ese formato a milisegundos y otra que actúe de forma inversa. El código es el siguiente:

```
val lapTimeToMs = (time: String) => {
  val regex = """([0-9]|[0-9][0-9]):([0-9][0-9])\.([0-9][0-9][0-9])""".r
  time match {
    case regex(min,sec,ms) => min.toInt * 60 * 1000 + sec.toInt * 1000 + ms.toInt
    case "\\N" => 180000
```

```
  }
}: Long
```

```
val msToLapTime = (time: Long) => {
  val mins = time / 60000
  val secs = (time - mins * 60000) / 1000
  val ms = time - mins * 60000 - secs * 1000

  val formattedSecs = if ((secs / 10).toInt == 0) "0" + secs else secs
  // if ms = 00x -> "0"+"0"+x . if ms = 0xx -> "0"+ms
  val formattedMs =
    if ((ms / 100).toInt == 0) "0" +
      (if ((ms / 10).toInt == 0) "0" + ms else ms)
    else ms
  mins + ":" + formattedSecs + "." + formattedMs
}: String
```

En la función `lapTimeToMs` convierto el formato de tiempo de vuelta a milisegundos. En este caso, lo hago con una expresión regular, de forma que extraigo los minutos, segundos y milisegundos de las posiciones correspondientes. Después, multiplico cada valor como corresponde y lo sumo. Es posible que, si el piloto no llegó a salir a pista, su tiempo de vuelta sea nulo, simbolizado por el string "\\N". En este caso, ha decidido usar 180000 milisegundos en su lugar, o 3 minutos. Se ha decidido usar esa cifra ya que es raro que una vuelta al circuito dure más de 2 minutos y de esta manera se "penalizará" al piloto que no haya acabado la vuelta.

De forma inversa, tenemos otra función llamada `msToLapTime` que, dado un valor en microsegundos, lo convierte al formato correcto. En este caso se hace la operación inversa. Se hallan los minutos, segundos y milisegundos para más adelante formatear el texto de forma que en el caso de que un piloto hiciese un tiempo de un minuto, tres segundos y tres milisegundos, quedase formateado como "1:03:003" en lugar de "1:3:3".

Tras esto hay que conseguir la UDF y registrarla, proceso que resulta sencillo con las siguientes instrucciones:

```
val lapTimeToMsUDF = udf(lapTimeToMs)
spark.udf.register("lapTimeToMs", lapTimeToMsUDF)
```

De esta manera podremos invocar la función `lapTimeToMsUDF`, le proporcionaremos una columna y nos de volverá otra ya procesada.

Una vez explicado esto, podemos continuar con el procesamiento del `DataFrame` final. Como comentamos, nos centramos en primera instancia en la columna `fastestLapTime`. Primero, debemos eliminar los valores nulos y después, todos los valores restantes los debemos convertir a milisegundos para poder operar con ellos. Esto lo podemos hacer de la siguiente manera:

```
results
  // filtro por temporada
```

```
  .join(races_filtered, Seq("raceId"), "right")
  .na.drop(Seq("fastestLapTime"))
  .withColumn("fastestLapTimeMs", lapTimeToMsUDF(col("fastestLapTime")))
```

Ya que este va a ser el `DataFrame` que devolvamos, podemos no guardarlo en ninguna variable y devolverlo directamente. Como viene siendo habitual, cogemos la tabla obtenida de los argumentos y filtramos las carreras. Después, con la función `na.drop`, eliminamos los valores nulos de la columna `fastestLapTime`. Si quisiéramos eliminar los valores nulos de varias columnas, bastaría con pasarle más nombres de columnas dentro de la lista que recibe.

Tras esto, para conseguir la columna con los milisegundos usamos `withColumn`, que recibe como nombre `fastestLapTimeMs` y como valor la conversión de la columna `fastestLapTime`, usando para ello la UDF que hemos definido.

Una vez hecho esto, aprovechamos la ventana que definimos anteriormente para hacer la media de las vueltas más rápidas de cada piloto tal que:

```
.withColumn("avgFastestLapMs", avg(col("fastestLapTimeMs")).over(driverWindow))
```

Ya que tendremos entradas de pilotos duplicadas, las eliminamos con la siguiente operación:

```
.dropDuplicates("driverId")
```

Hecho esto, necesitamos la media de todas las vueltas dadas por cada piloto, que tenemos guardadas en la variable `avgLapTimes`. Tendremos que hacer una intersección sobre la columna `driverId`, pero en este caso de tipo `left`, ya que queremos completar la información que ya tenemos.

Recordemos que nuestro objetivo es obtener la diferencia entre la media de vueltas rápidas y la media de todas las vueltas. El símbolo que tenga realmente no nos interesa, ya que resulta evidente que el piloto irá más rápido en las vueltas rápidas que en la media de vueltas, pero aún así utilizaremos el valor absoluto de esta resta para eliminar signos. Ya que esta diferencia está en milisegundos, también tendremos que convertirlos al formato de tiempo de vuelta utilizando la UDF que hemos comentado anteriormente.

El código para hacer todo esto que hemos comentado sería:

```
.join(avgLapTimes, Seq("driverId"), "left")
// saco el diferencial
.withColumn("diffLapTimes", abs(col("avgMs") - col("avgFastestLapMs")).cast(IntegerType))
// vuelvo a pasar a tiempo de vuelta
.withColumn("avgDiff", msToLapTimeUDF(col("diffLapTimes").cast(IntegerType)))
```

En principio podríamos decir que ya tenemos lo que queremos, pero en mi opinión, no es justo tener en cuenta a pilotos que por ejemplo han corrido una

sóla carrera, ya que no constituye una muestra significativa de la capacidad del piloto. Para solventar este problema podemos filtrar los pilotos no experimentados de la información que hemos obtenido utilizando la lista que llamamos `experiencedDrivers` de la siguiente manera:

```
.where(col("driverId").isInCollection(experiencedDrivers))
```

Una vez tenemos datos de todos los pilotos que nos interesan, pasamos a formatear la tabla que vamos a devolver. En concreto, sería interesante tener en una columna el nombre y apellido del piloto y en otra el diferencial calculado.

Para ello, tenemos que hacer otra intersección con la tabla `drivers` y concatenar el nombre y el apellido del piloto. Tras esto, nos quedamos con las columnas que nos interesan y ordenamos la tabla según el diferencial calculado de menor a mayor.

```
.join(drivers, "driverId")
.withColumn("driver", concat(col("forename"), lit(" "), col("surname")))
.select("driver", "avgDiff")
.orderBy("avgDiff")
```

3.3. Programación de queries en PySpark

En esta sección hablaremos de cómo se programan queries usando la API PySpark. Como muchos de los conceptos son muy similares o iguales a los ya comentados en la sección anterior, no entraremos tan en detalle, parándonos principalmente a explicar la funcionalidad nueva que nos vayamos encontrando.

Al igual que en la sección anterior, asumiremos que recibimos todos los `DataFrames` que necesitamos para comenzar la query, que también se hará en una función dedicada a ello. Esta función está definida como:

```
def queryEnConcreto(entrada1, entrada2, ...):
        ...
        return df_salida
```

Cuyo uso es exactamente el mismo al de su equivalente en Scala.

De igual forma que en la sección anterior, explicaremos detalladamente una query y dejaremos la otra para el apéndice A.2.

3.3.1. Mejor temporada para el espectador

En esta query vamos a intentar averiguar cuál ha sido la temporada más interesante desde el punto de vista del espectador. Para ello se van a calcular

tres métricas: el número de adelantamientos, el número de pilotos distintos que han liderado el campeonato y el número de pilotos distintos que han ganado una carrera en dicha temporada.

He considerado que son estos aspectos los que hacen que una temporada sea más interesante ya que cuanta más variedad haya en cuanto a ganadores de carreras y líderes del mundial, más cambian las circunstancias. En mi opinión resulta aburrido cuando a lo largo de una temporada solo gana un piloto o equipo por estar tan por encima del resto: cuanto más igualada esté la competición, más interesante resultará para el espectador. Por esto mismo se ha decidido también utilizar como métrica la cantidad de adelantamientos, aunque quizá esta esté sesgada en parte, ya que a lo largo de los años, y en especial desde la temporada 2014, los coches se han vuelto tan grandes que se hace complicada la lucha en pista y los adelantamientos. Sin embargo, buscamos la temporada más interesante y considero que cuantos más adelantamientos haya, mejor.

Al ser una cuestión un tanto subjetiva, otros podrían haber añadido métricas como número de accidentes, retiradas de la carrera y otras tantas, pero considero que por norma general todos podríamos ponernos de acuerdo en que estas tres métricas que se buscan estarían entre las que decidirían si una temporada es buena o no.

De esta query podemos destacar que no vamos a poder averiguar la mejor temporada de todos los tiempos. Esto es debido a que la tabla en la que se apoya principalmente la query, `lap_times`, solo contiene información a partir del año 1996.

Podemos ver el código completo de la query a continuación:

```
def averageRank(cols):
       return sum(cols) / len(cols)

def queryMejorTemporada(races, lap_times, driver_standings, results):

       seasonWindow = Window.partitionBy("year")
       driverRaceWindow = Window.partitionBy("driverId", "raceId")
       raceDriverLapWindow = Window.partitionBy("driverId", "raceId").orderBy("lap")

       averageRank = F.udf(averageRank, T.IntegerType())

       races = races\
              .select("raceId", "year")

       overtakes = lap_times\
              .withColumn("position", F.col("position").cast(T.IntegerType()))\
              .withColumn("lap", F.col("lap").cast(T.IntegerType()))\
              .join(races, "raceId")\
              .withColumn("positionNextLap", F.lead(F.col("position"),
                  1).over(raceDriverLapWindow))\
              .withColumn("positionsGainedLap", F.when(F.col("positionNextLap") <
                  F.col("position") , F.abs(F.col("position") -
                  F.col("positionNextLap"))).otherwise(0))\
              .groupBy("year")\
              .agg(F.sum(F.col("positionsGainedLap")).alias("positionsGainedSeason"))\
              .withColumn("rankPositionsGained",
```

```
                F.rank().over(Window.orderBy(F.col("positionsGainedSeason").desc())))

    leadersTroughoutSeason = driver_standings\
            .join(races, "raceId")\
            .where(F.col("position") == 1)\
            .dropDuplicates(["driverId", "position", "year"])\
            .groupBy("year")\
            .agg(F.approx_count_distinct(F.col("driverId")).alias("distinctLeaders"))\
            .withColumn("rankDistinctLeaders",
                F.rank().over(Window.orderBy(F.col("distinctLeaders").desc())))

    winnersTroughoutSeason = results\
            .join(races, "raceId")\
            .where(F.col("position") == 1)\
            .dropDuplicates(["driverId", "position", "year"])\
            .groupBy("year")\
            .agg(F.approx_count_distinct(F.col("driverId")).alias("distinctWinners"))\
            .withColumn("rankDistinctWinners",
                F.rank().over(Window.orderBy(F.col("distinctWinners").desc())))

    return overtakes\
            .join(leadersTroughoutSeason, "year", "inner")\
            .join(winnersTroughoutSeason, "year", "inner")\
            .withColumn("avgRank", averageRank(F.array(F.col("rankDistinctWinners"),
                F.col("rankDistinctLeaders"), F.col("rankPositionsGained"))))\
            .withColumn("overallRank", F.rank().over(Window.orderBy("avgRank")))\
            .sort("overallRank")
```

Descripción detallada

Ya que se van a utilizar las tablas `lap_times`, `driver_standings` y `results`, vamos a necesitar mapear cada `raceId`, presente en todas estas tablas, con la correspondiente temporada en la que se disputó la carrera. Para ello utilizaremos la tabla `races`, quedándonos solamente con las columnas `raceId` y `year`. El código es el siguiente:

```
races = races\
  .select("raceId", "year")
```

Solamente en este trozo pequeño de código se pueden ver algunas diferencias con la API de Scala. Principalmente se ve que se tiene que añadir el carácter \ al final de cada línea en la que se realiza una operación sobre el `DataFrame`. Iremos describiendo el resto de diferencias según vayan apareciendo.

También podemos aprovechar para crear las tres ventanas de particionado que vamos a usar. Estas son las siguientes:

```
seasonWindow = Window.partitionBy("year")
driverRaceWindow = Window.partitionBy("driverId", "raceId")
raceDriverLapWindow = Window.partitionBy("driverId", "raceId").orderBy("lap")
```

Una vez tenemos este `DataFrame` con una correspondencia directa entre carrera y temporada y las ventanas de particionado, podemos calcular el número

de adelantamientos. Para ello hemos de usar la tabla `lap_times`, que contiene información de todas las vueltas de cada piloto.

Después, viendo que tanto la columna `position` como `lap` son de tipo `String`, debemos pasarlas a entero para poder operar con ellas. Por norma general si quisiéramos comprobar una igualdad con ellas, como comprobar si estamos en la segunda vuelta, no tendríamos por qué hacer esta conversión de tipos, pero como vamos a ordenar la ventana de particionado por vuelta sí es necesario. Esto es porque dados los `String` "1", "2" y "19", el orden de menor a mayor sería "1", "19" y "2". La conversión de tipos la hacemos de la siguiente manera:

```
  .withColumn("position", F.col("position").cast(T.IntegerType()))\
  .withColumn("lap", F.col("lap").cast(T.IntegerType()))\
```

Aqui se pueden apreciar otra diferencia respecto a Scala. Por norma general, el código en PySpark suele ser mucho más explícito por la naturaleza del lenguaje. Python y Scala son opuestos en este aspecto.

Habiendo convertido los tipos de dichas columnas, debemos completar la información de nuestro DataFrame estableciendo una correlación entre carrera y temporada. Esto lo conseguimos interseccionándolo con el DataFrame que obtuvimos anteriormente de la siguiente manera:

```
.join(races, "raceId")\
```

Si no se especifica, por defecto Spark realiza una intersección de tipo "inner".

Lo siguiente que tenemos que obtener es la posición de cada piloto en la siguiente vuelta a la que se hace referencia en la fila actual. Para ello, necesitamos particionar por carrera y piloto y ordenar la ventana de datos por vuelta. En este caso utilizamos la función `lead`, que nos devuelve la columna que proporcionamos como parámetro, pero con las entradas desplazadas "hacia arriba" el número de entradas que se indique como parámetro. Es imprescindible que la ventana que utilicemos esté ordenada. En resumidas cuentas, tendríamos en la misma entrada la posición en esta vuelta y en la siguiente. Existe otra función llamada `lag` que tiene la misma funcionalidad, pero desplaza las entradas "hacia abajo". Para ambas funciones hay que tener en cuenta que siempre habrá una entrada de la columna desplazada que contenga un valor nulo, ya sea la primera o la última.

Teniendo la información de la siguiente vuelta, podemos ver el número de adelantamientos del piloto en esa vuelta. Para ello, si la posición en la siguiente vuelta es menor que en la actual se devuelve la diferencia y en otro caso se devuelve cero.

```
.withColumn("positionNextLap", F.lead(F.col("position"), 1).over(raceDriverLapWindow))\
.withColumn("positionsGainedLap", F.when(F.col("positionNextLap") < F.col("position") ,
    F.abs(F.col("position") - F.col("positionNextLap"))).otherwise(0))\
```

Tras esto, se pueden agrupar los datos según la temporada y se hace el sumatorio de los adelantamientos tal que:

```
.groupBy("year")\
.agg(F.sum(F.col("positionsGainedLap")).alias("positionsGainedSeason"))\
```

Por último, querríamos obtener la posición que ocuparía cada temporada si las ordenásemos de más adelantamientos a menos. Esto lo podemos conseguir con la función `rank`, que se utilizará sobre una ventana sin particionar y que esté ordenada únicamente por la columna que contiene el número de adelantamientos.

```
.withColumn("rankPositionsGained",
    F.rank().over(Window.orderBy(F.col("positionsGainedSeason").desc())))
```

La siguiente métrica que queremos calcular es el número de líderes distintos a lo largo de cada temporada. Para ello usamos la tabla `driver_standings` en lugar de `lap_times` y completamos la información de las temporadas al igual que antes. Tras esto, tendremos la clasificación al final de cada carrera, con una entrada por piloto, carrera y temporada. Como solo nos interesan los líderes, filtramos el `DataFrame` para quedarnos con las entradas donde `position` valga `1`

```
winnersTroughoutSeason = results\
  .join(races, "raceId")\
  .where(F.col("position") == 1)\
```

Como es bastante probable que un piloto lidere el campeonato en más de un punto a lo largo de la temporada, tenemos que deshacernos de las entradas duplicadas cada temporada:

```
.dropDuplicates(["driverId", "position", "year"])\
```

Tras esto, nuestro DataFrame contendrá solamente los distintos pilotos que han liderado el campeonato. Como lo que queremos es saber el conteo de estos pilotos para cada temporada, debemos agrupar los datos por temporada y utilizar la función `approx_count_distinct` sobre la columna `driverId`.

```
.groupBy("year")\
.agg(F.approx_count_distinct(F.col("driverId")).alias("distinctLeaders"))\
```

Con esto tenemos en nuestro DataFrame una entrada por cada año.

Por último, como para la métrica anterior, crearemos una columna que nos proporcione la clasificación de las temporadas en función a la métrica que acabamos de calcular:

```
.withColumn("rankDistinctLeaders",
    F.rank().over(Window.orderBy(F.col("distinctLeaders").desc())))
```

Para la última métrica que queremos calcular podemos reutilizar prácticamente entera la query anterior. La única diferencia será que se utilizará la tabla `results`. El código es el siguiente:

```
winnersTroughoutSeason = results\
  .join(races, "raceId")\
  .where(F.col("position") == 1)\
  .dropDuplicates(["driverId", "position", "year"])\
  .groupBy("year")\
  .agg(F.approx_count_distinct(F.col("driverId")).alias("distinctWinners"))\
  .withColumn("rankDistinctWinners",
      F.rank().over(Window.orderBy(F.col("distinctWinners").desc())))
```

Una vez calculadas las tres métricas que necesitamos, debemos encontrar una manera de establecer una clasificación global teniéndolas en cuenta. En este caso, se ha decidido crear una función que para cada entrada calcule la media de cada clasificación individual. De esta manera, si una temporada ha quedado primera en más adelantamientos, tercera en más líderes de la clasificación y segunda en más ganadores distintos, se haría la media de 1, 3 y 2.

Para ello usamos una función definida por el usuario, o UDF que recibirá como parámetro una lista y devolverá su media. El código es el siguiente:

```
def averageRank(cols):
  return sum(cols) / len(cols)

averageRank = F.udf(averageRank, T.DoubleType())
```

Es importante definir el tipo de datos de la salida de la función. En este caso, queremos devolver la media como entero, ya que no nos interesan los decimales. En caso de no definir tipo de salida, se devolverá un `StringType` que puede dar problemas a la hora de ordenar más adelante.

Para concluir, queremos interseccionar las tablas de todas las métricas según la columna `year`. En este caso podemos definir el join como inner, ya que al hacer la media no nos interesa que haya ningún nulo, en el caso extraño de que aparezca una temporada sin alguna métrica. Después aplicamos nuestra UDF a las columnas que queremos y preparamos el `DataFrame` para devolverlo y más adelante poder mostrarlo por pantalla. El código es el siguiente:

```
return overtakes\
  .join(leadersTroughoutSeason, "year", "inner")\
  .join(winnersTroughoutSeason, "year", "inner")\
  .withColumn("avgRank", averageRank(F.array(F.col("rankDistinctWinners"),
      F.col("rankDistinctLeaders"), F.col("rankPositionsGained"))))\
  .withColumn("overallRank", F.rank().over(Window.orderBy("avgRank")))\
  .drop("rankDistinctWinners", "rankDistinctLeaders", "rankPositionsGained", "avgRank")\
  .sort("overallRank")
```

3.4. Migración de queries

En esta sección me centraré en describir el proceso de migración de una query escrita en Scala Spark a una query en PySpark. Para ello utilizaré como ejemplo la query descrita en la sección A.1.1, pero sin tener en cuenta la abstracción que separa las lecturas de la propia query, es decir, se leerán las tablas y directamente se operará sobre ellas.

Los cambios en los que nos vamos a centrar van a ser los siguientes:

- Imports.
- Añadir tipos de variable.
- Indentado inesperado.
- Operadores.
- Ventanas.
- Separaciones y comentarios.
- Tipos de Spark.
- Listas y mapas.

Imports

Para migrar una query, lo primero es traducir los `imports`. En este caso vamos a necesitar las funciones de Spark SQL, los tipos propios de Spark y las ventanas. En Python, al ser un lenguaje muy explícito, es recomendable importar las funciones que vayamos a necesitar con la sintaxis `import paquete as nombre` si vamos a necesitar el paquete completo. En nuestro caso, esto es absolutamente necesario, ya que si importásemos la función `abs` del paquete `pyspark.sql.functions`, podría confundirse con la función `abs` que el lenguaje trae. Como este caso hay varias, entre ellos `max`, `min` o `avg`, nuestros `imports` tendrán que tener la siguiente forma:

```
from pyspark.sql import functions as F
from pyspark.sql import types as T
from pyspark.sql import Window
```

Estamos ignorando que necesitamos importar también `SparkSession` para crear la propia sesión de Spark.

Tipos de variable

Una vez hemos importado todo lo que necesitamos y tenemos creada la `SparkSession`, que se hace de manera muy parecida a la de Scala, podemos pasar a migrar la query para obtener nuestro primer `DataFrame`.

Partimos del siguiente código, en el que se carga directamente un `DataFrame` y se realizan dos operaciones sobre él:

```
val races = spark.read.format("csv")
```

```
      .option("header", "true")
      .option("sep", ",")
      .load("../data/races.csv")
      .select("raceId", "year")
      .where(col("year") === 2021)
```

Lo primero que veremos al intentar ejecutar este trozo de código es que en Python no necesitamos indicar el tipo de variable, como es el caso de Scala u otros lenguajes como Java o C. En lugar de eso, directamente asignamos un valor al nombre de variable que deseemos. Por ello, nos tendremos que deshacer de todos `val` y `var` que haya en nuestro código. En caso de tener funciones declaradas como `var` en nuestro código, debemos pasar su definición para que use `def` en su lugar.

Indentado inesperado

Lo siguiente que nos dirá el intérprete de Python al intentar ejecutar el código habiendo eliminado lo que acabamos de comentar es que ha habido un indentado inesperado. Python en concreto es un lenguaje estricto con las indentaciones, ya que indican al programa el scope del código indentado. Para solucionar este error, necesitamos incorporar al final de cada línea el carácter \, lo cual hace que el intérprete vea un trozo de código de varias líneas como una sola. Este carácter tendrá que estar presente en todas las líneas menos la última.

Con estos cambios incorporados, nuestra query tiene la siguiente pinta:

```
races = spark.read.format("csv")\
      .option("header", "true")\
      .option("sep", ",")\
      .load("../data/races.csv")\
      .select("raceId", "year")\
      .where(col("year") === 2021)
```

Operadores

Si ejecutamos esto, nos daremos cuenta de que el operador `===` no es válido. Para realizar comparaciones de este tipo, tenemos que usar el operador `==`. Este error nos puede adelantar que tendremos que revisar todos los operadores que tengamos en el código. Otro tipo de operadores que cambian son los lógicos. Por ejemplo, las operaciones `&&` (and) y `||` (or) se denotan con `&` y `|` respectivamente.

De nuevo, una vez hagamos el cambio en el operador de igualdad y ejecutemos, veremos que nos encontramos con otro error. Esta vez se trata de un error distinto. En este caso nos dice que la función `col` no está definida. Esto es porque necesitamos añadir el paquete del que proviene delante del nombre de la función. En este caso, como importamos las funciones con el nombre `F`, tendremos que cambiarlo por `F.`.

Si añadimos el prefijo a la función `col`, veremos que el código ya ejecuta

correctamente. Finalmente el código migrado quedaría tal que:

```
races = spark.read.format("csv")\
       .option("header", "true")\
       .option("sep", ",")\
       .load("../data/races.csv")\
       .select("raceId", "year")\
       .where(F.col("year") == 2021)
```

Ventanas

El siguiente paso sería migrar las ventanas, lo cual no nos llevará ningún problema ya que al quitar el `var` o `val` que precede al nombre de la ventana tendríamos todo el trabajo hecho.

Separaciones y comentarios

Es posible que nos encontremos con queries como la siguiente, que añaden líneas vacías posiblemente como forma de ordenar el código y separar las distintas partes que lo componen, o incluso añadir comentarios entre líneas para complementar el código:

```
val driverStats = spark.read.format("csv")
       .option("header", "true")
       .option("sep", ",")
       .load("../data/lap_times.csv")

       .withColumn("position", col("position").cast(IntegerType))
       .withColumn("lap", col("lap").cast(IntegerType))
       // filtramos las carreras
       .join(races, "raceId")
```

Esto en Python no es posible hacerlo por el mismo motivo por el que tenemos que añadir el carácter \ al final de cada línea: Internamente interpretará todo este código como una sola línea. Para solucionar esto tenemos dos opciones, ambas correctas.

La primera sería deshacernos de los espacios y juntar todo el código, teniendo también que eliminar los comentarios. La segunda sería partir la query en varias partes como se muestra a continuación:

```
driverStats = spark.read.format("csv")\
       .option("header", "true")\
       .option("sep", ",")\
       .load("../data/lap_times.csv")

driverStats = driverStats\
       .withColumn("position", F.col("position").cast(T.IntegerType()))
       .withColumn("lap", F.col("lap").cast(T.IntegerType()))

# filtramos las carreras
driverStats = driverStats\
       .join(races, "raceId")
```

Tipos de Spark

Otro detalle que tenemos que comentar son los tipos propios de Spark. Los tipos en sí no difieren de una librería a otra, pero sí cómo se usan en el código. En Scala nos encontramos una versión más sencilla y menos "abultada" o explícita, mientras que en Python tenemos que añadir el prefijo `T.` y los paréntesis tras el propio tipo. En este caso, el prefijo no es necesario, ya que por defecto Python no tiene ningún tipo o función que comparta nombre con los tipos de Spark y por ende podríamos importar todo el contenido del paquete `types`, pero se considera buena práctica importar como lo hemos hecho para separar los "namespaces".

Por otro lado, es muy probable que nos encontremos con operaciones como la siguiente:

```
val results = spark.read.format("csv")
       .option("header", "true")
       .option("sep", ",")
       .load("../data/results.csv")

       .withColumn("position", col("position").cast(IntegerType))
       .withColumn("grid", col("grid").cast(IntegerType))
       .withColumn("points", col("points").cast(IntegerType))

       .join(races, "raceId")
       .join(driverStats, Seq("raceId", "driverId"), "left")
       .join(drivers, "driverId")
```

Listas y mapas

Donde podemos ver que se utilizan listas, en este caso para indicar las columnas sobre las que hacer la intersección. En este caso, la traducción resulta sencilla. Tendríamos que coger los elementos de la lista y envolverlos entre corchetes (`[]`), eliminando en este caso el `Seq` y sustituyendo los paréntesis por ellos. Es posible que veamos otros tipos como `List`, pero la idea se mantiene.

Por otro lado esto nos puede surgir el tener que traducir un Mapa. En este caso usaremos los diccionarios de Python, en el que sustituiremos los caracteres `->` que separan la clave del valor por `:`. También tendremos que eliminar de nuevo el `Map` y cambiar los paréntesis que lo envuelven por llaves del estilo

Funcionalidad con implementaciones distintas

En Spark existe una función llamada `lit` que dado un valor de entrada de cualquier tipo nos devuelve una cuyas entradas contienen únicamente ese valor. En Scala Spark también existe una función llamada `typedLit`, que permite hacer lo mismo para tipos complejos como listas o mapas. Resultan útiles en el caso de que queramos aplicar un mapa a una columna, ya que le pasaríamos una columna de entrada como parámetro y devolvería otra con todos los valores mapeados.

El problema a la hora de migrar código a PySpark es que esta función no existe, y para mapear una columna hay que hacerlo de una manera más enrevesada.

Mientras que el código para Scala Spark se vería así:

```
val mySparkMap = typedLit(Map(...))

...
.withColumn("mapeada", mySparkMap(col("input")))
```

El código para implementar la misma funcionalidad sin usar UDFs sería el siguiente:

```
from pyspark.sql.functions import col, create_map, lit
from itertools import chain

myPythonDict = {...}

mapping_expr = create_map([lit(x) for x in chain(*myPythonDict.items())])

...
.withColumn("value", mapping_expr.getItem(col("clave")))
```

Como se puede observar, hay que hacer algún paso extra, pero más importante que eso es que dependemos de otro paquete externo a Spark. Además, el uso del `mapping_expr` cambia ligeramente entre las versiones 2 y 3 de Spark, y por lo tanto es otra cosa más que tener en cuenta a la hora de migrar una versión a otra.

Funciones concretas con declaraciones distintas

Se ha observado que algunas funciones de `DataFrame` que están declaradas de una manera en Scala Spark no lo están de la misma forma en PySpark. Un ejemplo de ello es la función `na.replace`, que en el caso de la API de Scala puede recibir como primer argumento un valor por el que serán sustituidos los nulos y como segundo argumento una lista de columnas a tener en cuenta.

En la API de Python estos dos argumentos se invierten: primero recibimos las columnas a tener en cuenta y después el valor que sustituirá los nulos.

Visto este caso podemos llegar a preguntarnos qué otras funciones siguen este patrón, es decir, que en una API tienen una declaración y en la otra siguen una distinta. Esto es una pregunta difícil de responder, ya que existen numerosas funciones aplicables a un `DataFrame`, y para comprobarlo tendríamos que dedicar mucho tiempo y esfuerzo, más aún si se hace manualmente.

Además, también existe el caso de que tengamos funciones en una API que no existan en la otra, como es el caso de `replace`, que está en PySpark pero no en Scala Spark.

Con todos estos cambios mencionados podrían cubrirse la mayoría de queries sencillas que nos encontremos.

3.4.1. Expresiones regulares para facilitar la migración

En esta sección vamos a hablar de una herramienta que puede facilitar mucho la migración de queries entre APIs. Estoy hablando de ciertas expresiones regulares que potencialmente nos ahorrarían una cantidad considerable de tiempo en el caso de que tuviéramos que migrar queries muy grandes. A pesar de que nos pueden facilitar la experiencia para labores repetitivas como añadir el prefijo `F.` a cada función de columna, hay ciertos aspectos que seguirán requiriendo intervención manual, como los que comentaremos más adelante.

Se tratarán los siguientes casos:

- Sustituciones directas.
- Estandarización de las invocaciones a `col`.
- Añadir `F.` a las funciones de columna.
- Corregir las invocaciones a los tipos de Spark.
- Añadir paréntesis a condiciones compuestas.
- Eliminación de `val` y `var`.
- Añadir el carácter `\` al final de cada línea.
- Traducción de listas, secuencias y arrays.
- Traducción de Mapas.
- Otros ajustes menores.

Al no disponer de todo el espacio que quisiera, se van a explicar las dos más interesantes y dejaremos el resto para el apéndice B.1. Todas ellas se desarrollarán con el comando `sed` en mente, que utiliza el estándar POSIX ERE, salvo dos de ellas. Estas las veremos en sus respectivos apartados.

Añadir `F.` a las funciones de columna

Como hemos visto ya, a la hora de importar las funciones de Spark en un entorno de Python es necesario usar un nombre concreto ya que algunas de estas tienen la misma denominación que otras que ya trae incorporadas Python, como podría ser el caso de `abs`. En este caso asumiremos que vamos a importar estas funciones de la siguiente manera:

```
from pyspark.sql import functions as F
```

Con esto en mente, nuestro objetivo sería añadir dicho prefijo a todas las funciones que se hallen en nuestro código. Como no he encontrado ningún patrón por el que se rijan estas invocaciones como en otros casos, lo más sencillo será conseguir los nombres de todas las funciones disponibles y buscarlas con el operador OR en nuestra expresión regular. Esto lo podemos hacer de la siguiente manera desde una sesión de Python habiendo ya importado las funciones como se ha mostrado anteriormente:

```
regex = "(" + "|".join(dir(F)[30:]) + ")\("
```

La función `dir` lista los contenidos de un paquete en este caso, y necesitamos descartar los 30 primeros elementos ya que no se trata de funciones como tal. Entendido esto, esta expresión regular la podemos imprimir por pantalla y guardarla para aplicar cuando la necesitemos fuera de Python.

En nuestro caso, lo que detecte esta expresión regular lo sustituiremos por `F.\1(`.

Adición del carácter \ al final de cada línea

La siguiente tarea será añadir el carácter `\` al final de cada línea de la query. Para esto utilizaremos dos expresiones regulares. La primera de ellas detectará como línea de query aquella que tenga un cierre de paréntesis y una nueva línea, y la segunda actuará cuando tengamos una asignación de un `DataFrame` a otro seguido de las operaciones pertinentes.

Para ilustrar este caso, pongamos que tenemos la siguiente query, donde ya hemos aplicado algunas de nuestras expresiones regulares:

```
test = data
       .withColumn(...)
       .filter(...)
```

Si aplicamos la expresión regular `(\))$`, que nos detecta el cierre de paréntesis y el fin de línea, y sustituimos por `\1\\`, la query quedaría tal que:

```
test = data
       .withColumn(...)\
       .filter(...)\
```

Sin embargo, esto seguirá lanzando un error, ya que también deberíamos añadir dicho carácter en la primera línea. Para ello detectamos cuando tengamos una asignación con `([a-zA-Z0-9_]*) = ([a-zA-Z0-9_]*)$` , podemos sustituir por `\1 = \2\\`, de forma que la query ya quedaría como:

```
test = data\
       .withColumn(...)\
       .filter(...)\
```

Esta última expresión detecta cualquier variable asignada a otra, de forma que si nuestro código contiene asignaciones normales u operaciones que no utilicen Spark también se verán afectadas.

Con el conjunto de sustituciones directas y expresiones regulares desarrolladas

tanto aquí como en el apéndice ya tenemos cubierta una gran parte del código que nos podamos encontrar a la hora de crear queries básicas. Cierto es que dentro de todas las opciones que nos ofrece Spark se cubren casos muy limitados, pero si se avanzase en el tema de las expresiones regulares quizá fuese posible cubrir la gran mayoría del código que se puede desarrollar en Scala Spark. A continuación se muestran algunas de las limitaciones nos encontraríamos, es decir, código que no se puede traducir directamente y que requeriría una parte de interpretación, análisis de la funcionalidad y replicado de la misma.

Limitaciones de las expresiones regulares

Pese a que gran parte de la traducción o migración se puede hacer mediante expresiones regulares, hay ciertos aspectos en los que nos podemos encontrar ciertas limitaciones. Vamos a comentar dos ejemplos:

Separación de código por comentarios

Como se comentó anteriormente, es posible intercalar comentarios y líneas en blanco en Scala sin que de error. De hecho, puede ser útil a la hora de aclarar ciertos aspectos del código, pero en Python esto no está permitido. Todos los comentarios tienen que ir antes o después de la query de Spark. Por ello, sería buena idea detectar estos casos en los que tenemos código con comentarios intercalados y separarlo como propusimos anteriormente:

```
driverStats = spark.read.format("csv")\
       .option("header", "true")\
       .option("sep", ",")\
       .load("../data/lap_times.csv")

# filtramos las carreras
driverStats = driverStats\
       .join(races, "raceId")
```

Es decir, buscamos detectar comentarios y "partir" la query en dos. Para ello, necesitamos guardar el nombre del `DataFrame`, detectar el código hasta un comentario y el comentario en sí, y estos dos N veces. Esto lo podemos hacer con la siguiente expresión:

```
([^ ]*) = ([^\#]*)([^\n]*)([^\#]*)
```

Sin embargo, esto solo cubre el código hasta el segundo comentario, sin incluir. Para cubrir más casos necesitaríamos o bien copiar varias veces los dos últimos grupos de captura o los dos primeros. La tercera opción sería cubrir uno de estas dos opciones que menciono con un `*`, para indicar que se repite N veces, pero entonces nos topamos con el siguiente problema: ¿Hasta dónde vamos a llegar? Tenemos que recordar que para sustituir necesitamos indicar los grupos de captura y el orden en el que los queremos. Si tenemos una query muy larga

con muchos comentarios intercalados, nuestra expresión regular detectará cada uno de esos casos, y tendremos que hacer una expresión regular de sustitución individual al caso. En resumen: no se puede estandarizar usando el estándar que estamos utilizando, ya que podemos tener N comentarios intercalados.

Funciones definidas por el usuario

Como es de suponer, estas expresiones regulares no pueden sustituir las UDFs que haya programado el usuario en Scala, ya que se trataría de una migración de lenguajes más que de APIs. Para migrar estas funciones tendríamos que hacerlo a mano, teniendo en cuenta la entrada, la salida y que proporcione la misma funcionalidad exacta que su contraparte.

A la hora de realizar la migración de varias de las queries programadas hubo que hacer el cambio a mano de las UDFs que se utilizaron. Sin embargo, no resulta una tarea tan repetitiva como el resto de la migración, ya que se trata de un cambio de funciones individuales y requiere más concentración por parte de quien se encarga de ello.

Es por esto que no resultó tan arduo como el resto de la migración, y de hecho fue una manera de descansar y aprender más sobre ambos lenguajes fuera de Spark.

3.4.2. Automatización de la migración

Vamos a ver ahora cómo podríamos automatizar el proceso. Hasta el momento tendríamos que haber aplicado estas expresiones regulares manualmente utilizando alguna herramienta como Visual Studio Code, que nos permite utilizar estas expresiones regulares con sustitución haciendo cambios mínimos a las desarrolladas, ya que usa un estándar distinto. Sin embargo, sigue resultando una tarea ciertamente repetitiva, especialmente si tenemos varias queries que migrar. Por ello se ha desarrollado un *script* que se encargará de aplicarlas secuencialmente por nosotros. Los detalles más importantes sobre este *script* los comentaremos a continuación, mientras que otros aspectos a destacar se han explicado en el apéndice B.2.

Consiste principalmente en el uso del comando `sed`, que nos permite aplicar expresiones regulares a *streams* de texto o archivos. Esta segunda opción es la que nos interesaría. Podemos ejecutar en nuestra *shell* un comando como el que se muestra a continuación para sustituir todas las apariciones de la palabra "null" por "None", su homólogo en Python:

```
sed -E 's/null/None/g' <filename>
```

Podemos ver que se ejecuta con el *flag* `-E`, que habilita las expresiones regulares extendidas y que el formato del comando interno de `sed` sigue el formato `s/<busca>/<sustituye>/g`. En este caso usamos `s` para sustituir y `g` al final para aplicar la expresión de forma global, para que se aplique en todo el archivo y no solo en la primera aparición de nuestro patrón.

Con esto visto podemos dar el siguiente paso: utilizar los *script* propios de `sed`. Estos nos permiten definir todas las operaciones que queremos realizar de forma secuencial en un archivo. Para usarlos debemos usar el *flag* `-f` y proporcionar a continuación el archivo con nuestras operaciones.

Por último, podemos proporcionar a `sed` el archivo sobre el que operar y nos devolverá un *stream* de texto que podemos redirigir a un archivo. Un ejemplo de uso de este mandato con todas las opciones ya definidas podría ser el siguiente:

```
sed -E -f sedscript.sed inputfile >outputfile
```

Si en el *script* añadimos todas las expresiones desarrolladas y sus respectivas sustituciones en el formato de operación de `sed`, podremos automatizar ya gran parte del proceso.

Sin embargo, existen tres excepciones a esto: la declaración de mapas y listas y la gestión de librerías importadas.

Las primeras nos resultan un problema debido a que pueden declararse en varias líneas. Por ello, utilizar las expresiones regulares extendidas de `sed` no resultará suficiente para estos casos, ya que no es posible diferenciar el carácter de retorno de carro. Es por esto que, en estos dos casos concretos se va a utilizar `perl`, como ya se describió en sus respectivos apartados. Si primero nos encargamos de estos casos y luego redirigimos la entrada al mandato `sed` que hemos explicado sí podremos podremos tratar de todos los casos que hemos descrito.

La gestión de librerías importadas resultan no triviales. En este caso, en lugar de sustituir usando expresiones regulares, vamos a añadir las librerías de PySpark a mano y a eliminar las de Scala. Para añadir las nuevas podemos usar la siguiente instrucción a nuestro *script* de `sed`:

```
1i <texto a añadir>
```

Esto nos añadirá en la primera línea lo que deseemos. Nos gustaría añadir todas las dependencias que podamos encontrar en nuestro código. Para simplificar, importaremos la `SparkSession`, `Window` para hacer operaciones de ventana, los tipos de Spark y las funciones. Estos dos nos interesa que reciban el nombre de `T` y `F`, así que añadiéndolos nosotros podemos asegurar de que al ejecutar el código resultante no dará error al no encontrar las funciones o tipos si se han importado con otros nombres.

De forma similar, `sed` nos proporciona una forma de eliminar líneas que sigan

un patrón definido por una expresión regular. En nuestro caso nos interesa deshacernos de todas las librerías de Spark, que siguen el formato `org.apache.spark.<paquete>`. Al ser un patrón común, no es difícil encontrar una expresión que los detecte. También nos interesaría deshacernos de los implícitos de Spark, que siguen un patrón tal que `<nombre del SparkSession>.<paquete>`. Como no podemos adivinar qué nombre le ha dado el usuario a su `SparkSession`, asumiremos que lo habrá llamado "`spark`".

Para eliminarlos por lo tanto añadiremos a nuestro *script* de `sed` las siguientes líneas:

```
/import org\.apache\.spark\.*/d
```

```
/import spark\.*/d
```

Nótese que con el `d` final le estamos diciendo que elimine la línea.

Por último deberíamos tener también en cuenta que el `SparkSession` se declara de forma ligeramente diferente. En Scala primero haríamos:

```
val spark = SparkSession
      .builder()
      ...
```

Mientras que en PySpark hacemos:

```
spark = SparkSession.builder\
      ...
```

Para arreglar esto lo más fácil es eliminar la línea que contiene el `.builder()` y sustituir `SparkSession` por `SparkSession.builder`.

Habiendo visto esto, lo más cómodo es hacer un *script* de *shell* que se encargue de recoger el archivo de entrada, aplicar todas estas operaciones y escribir el resultado en otro archivo del mismo nombre pero con extensión `.py`. Además de esto, ya que es posible que en la última línea del *script* de Python tengamos un `\`, tenemos que asegurarnos de dejar una línea extra. Esto lo hacemos con:

```
printf "\n\n»>outputfile
```

Siendo este el último paso que realizará nuestro *script*, podemos ver que, si le proporcionamos un archivo de entrada con código de Scala Spark nos creará otro con el equivalente en PySpark y el mismo nombre.

3.5. Uso de Plotly

Esta librería nos permite mostrar datos gráficamente con una configuración sencilla. Por ejemplo para mostrar por pantalla un gráfico de barras que represente la métrica `positionDelta` de cada piloto, resultante de la query descrita en el apéndice A.1.1, haríamos lo siguiente:

```
val driverCodes = results
       .sort("positionDelta")
       .select("code")
       .as[String]
       .collect()
       .toList

val positionDelta = results
       .sort("positionDelta")
       .select("positionDelta")
       .as[Double]
       .collect()
       .toList

val data = Seq(
       Bar(
              driverCodes,
              positionDelta
       )
)

val layout = Layout(
       barmode = BarMode.Group
)

plot(data, layout)
```

Como se puede observar, necesitamos recoger ambos ejes de la gráfica como listas para luego hacer la configuración de la gráfica en sí.

Ese código resulta en una gráfica que mostrará en el eje horizontal los distintos pilotos y en el vertical el valor de su `positionDelta`.

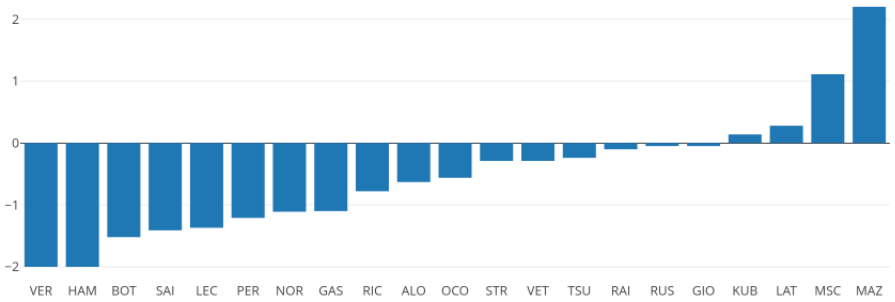


Figura 3.17: Diferencia entre posición de salida y al final en la temporada 2021 por piloto

4

Experimentos / Validación

4.1. Respuestas a las consultas

En este apartado se proporcionarán las respuestas obtenidas a las consultas descritas en los apartados anteriores. Veremos tanto respuestas obtenidas directamente con Spark en formato de texto como gráficas resultantes de la utilización de la librería Plotly. Nos vamos a centrar en los resultados de la query "Análisis de temporada por piloto". El resto de resultados se podrán consultar en el apéndice C.

A lo largo de esta sección van a aparecer en numerosas ocasiones los códigos de piloto en lugar de sus nombres completos. Esto es por mantener las figuras con un tamaño razonable. A continuación se proporciona una leyenda que establece una correspondencia entre código de piloto y su nombre completo:

code	fullName
ALO	Fernando Alonso
BOT	Valtteri Bottas
GAS	Pierre Gasly
GIO	Antonio Giovinazzi
HAM	Lewis Hamilton
KUB	Robert Kubica
LAT	Nicholas Latifi
LEC	Charles Leclerc
MAZ	Nikita Mazepin
MSC	Mick Schumacher
NOR	Lando Norris
OCO	Esteban Ocon
PER	Sergio Pérez
RAI	Kimi Räikkönen
RIC	Daniel Ricciardo
RUS	George Russell
SAI	Carlos Sainz
STR	Lance Stroll
TSU	Yuki Tsunoda
VER	Max Verstappen
VET	Sebastian Vettel

Figura 4.1: Leyenda de pilotos

4.1.1. Análisis de temporada por piloto

En esta query se pretendía realizar un análisis sencillo de cierta temporada poniendo el foco en los pilotos. En concreto, buscábamos calcular las siguientes métricas:

- Puntos conseguidos en el campeonato.
- Media de puntos.
- Porcentaje de puntos respecto al piloto que más obtuvo.
- Total de podios.
- Porcentaje de carreras acabadas en podio.
- Diferencia media entre la posición de salida y la obtenida al cruzar la meta.
- Media de posiciones perdidas.
- Media de posiciones ganadas.
- Total de posiciones ganadas y perdidas.

- Vueltas lideradas.
- Porcentaje de vueltas lideradas.

En este caso, como en el `DataFrame` final tenemos muchas métricas, no nos interesa mostrarlos en forma de texto como en el caso anterior, sino que para facilitar el análisis, sería más conveniente mostrar las métricas de forma gráfica.

A continuación se presenta un gráfico en el que podemos ver en el eje horizontal los distintos pilotos y en el vertical el valor de su `positionDelta`. El código utilizado para obtenerlo se ha descrito en el apartado 3.5.

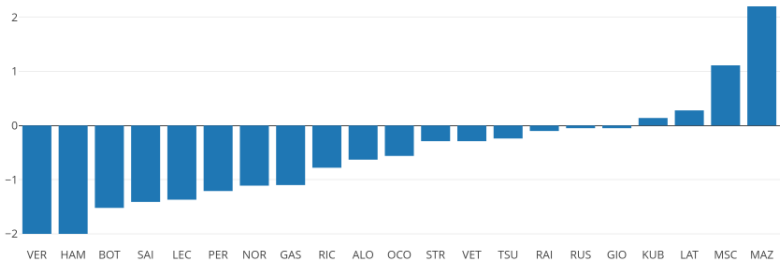


Figura 4.2: Diferencia entre posición de salida y al final en la temporada 2021 por piloto

Podemos llegar a la conclusión de que, cuanto más adelante clasifiques, más posiciones es probable que pierdas. Probablemente esto sea por que hay menos gente para adelantar, mientras que si clasificas al final de la parrilla será más fácil hacer adelantamientos y más difícil ser adelantado al tener menos gente detrás.

Otra métrica que resulta interesante es la media de posiciones perdidas, que será en parte análoga a la recién mostrada. Los resultados son los siguientes:

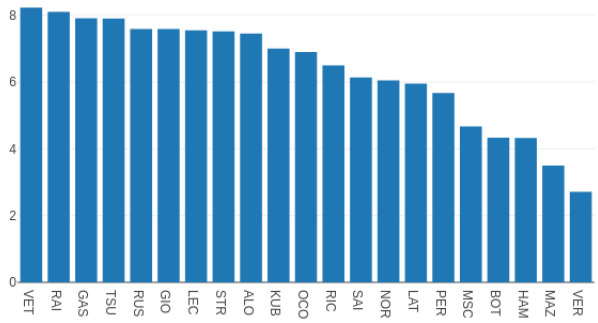


Figura 4.3: Media de posiciones perdidas en la temporada 2021 por piloto

Podemos llegar a la conclusión de que, de media, pierden menos posiciones los que están más al frente y al final de la parrilla. También podemos llegar a la conclusión de que esta temporada fue una con mucha diferencia entre el grupo de constructores que competían por el título, el grupo medio y el grupo más bajo, ya que vemos que los pilotos que de media acabaron al frente y al final solían perder menos posiciones que los pilotos del grupo medio.

La posición media en la que terminaron los pilotos en esta temporada la podemos ver a continuación:

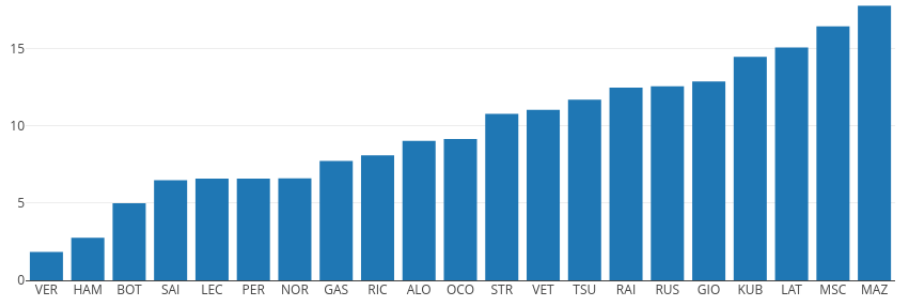


Figura 4.4: Media de posiciones finales en la temporada 2021 por piloto

Otra métrica interesante es el porcentaje de vueltas lideradas a lo largo de la temporada. Podemos obtener el siguiente gráfico:

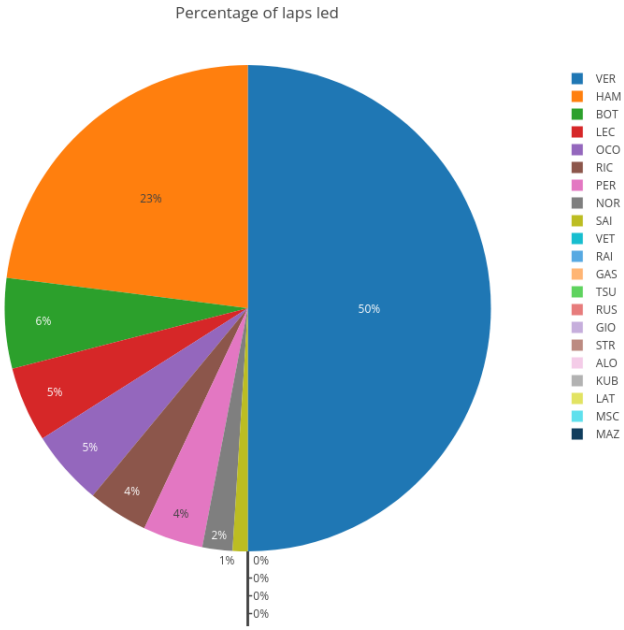


Figura 4.5: Porcentaje de vueltas lideradas en la temporada 2021 por piloto

Se observa que la mitad de las vueltas han sido lideradas por Max Verstappen, mientras que su competidor por el título, Lewis Hamilton, lideró un 23 % de las vueltas. Aquí podemos ver de nuevo representada la diferencia de los dos contendientes al título respecto al resto de pilotos. Sin embargo, también podemos concluir que no es necesario liderar la mayoría de vueltas para competir por el título, ya que a pesar de lo que se muestra en esta métrica, tanto Max como Lewis llegaron igualados en puntos a la última carrera de la temporada.

Podemos achacar que Max no dominase la temporada en puntos a que quizá a Lewis le costase adelantarle, ya que, como vimos antes, de media terminó por detrás de su contrincante. Se podría explicar también si Max hubiese tenido más problemas de fiabilidad y, a pesar de haber liderado la carrera, hubiese tenido que abandonar más carreras que Lewis.

4.2. Comparativas de rendimiento

En este apartado se va a realizar una comparativa de rendimiento entre ambas API. Para ello se van a comparar distintas métricas obtenidas de la Spark UI. A grandes rasgos se van a comparar el número de Jobs, Stages y Tasks de cada proceso, así como su Task Time, el total leído de disco y el total leído y escrito en *shuffles*.

Además, se va a comparar el rendimiento de Spark al usar ficheros de entrada en formato CSV, parquet sin particionar y parquet particionado.

A la hora de hacer comparativas de este estilo, es importante igualar lo máximo posible las condiciones y el entorno del que dependen las mediciones. En nuestro caso, es importante que ambas sesiones de Spark cuenten con los mismos recursos y que el código sea lo más parecido posible, sin ninguna optimización de por medio. Por ejemplo, si para una de las API usamos una UDF, pero en la otra no es necesario porque existen mecanismos para evitarla, entonces debemos en ambos casos usar la UDF.

Lo primero que necesitamos para ello es crear un objeto `SparkSession`. Esto se hace de la siguiente manera:

```
val spark: SparkSession = SparkSession
       .builder()
       .config("spark.sql.shuffle.partitions", 4)
       .master("local[4]")
       .getOrCreate()
```

En nuestro caso con estas opciones es suficiente, ya que estamos dedicando 4 núcleos de nuestra máquina local y 4 particiones de shuffle para las tareas que vayamos a realizar. Sin embargo, existen otras opciones que podríamos añadir si fuese necesario, como un nombre para la aplicación con `.appName("Nombre")`.

De esta manera, si quisiéramos configurar algún parámetro, lo haríamos añadiendo más modificaciones tal que:

```
val spark: SparkSession = SparkSession
       .builder()
       .config("spark.some.config.option", "some-value")
       .config("spark.some.config.option", "some-value")
       ...
       .master("local[N]")
       .getOrCreate()
```

Una vez tenemos el `SparkSession` creado correctamente, podemos usarlo para leer y escribir datos en distintos formatos, como CSV o Parquet. Además, nos permitirá crear `DataFrames` a partir distintos de tipos de datos, como Listas o Tuplas.

En los ejemplos anteriores hemos visto cómo se crearía el `SparkContext` para la API de Scala, pero en el caso de usar PySpark, lo crearíamos de la siguiente manera:

```
spark = SparkSession.builder\
       .config("spark.sql.shuffle.partitions", 4)\
       .master("local[4]")\
       .getOrCreate()
```

Ya que nuestras queries son sobre conjuntos de datos pequeños, conviene reducir el número de particiones de shuffle, ya que por defecto hay 200. Spark está pensado para volúmenes grandes de datos, y es por eso que tiene un valor tan alto. El *shuffle* es la manera que tiene Spark de redistribuir datos a los distintos ejecutores e incluso entre máquinas. Como en nuestro caso estamos en un entorno local con 4 cores no hacen falta tantas particiones, y se ha optado por usar solamente 4.

Los procesos en sí serán los mismos descritos en apartados anteriores y, en el caso de los que no se hayan descrito para un determinado lenguaje, se habrá obtenido utilizando las técnicas mencionadas en el apartado 3.4. La única diferencia respecto a los descritos es que al final se ejecutará un `collect` sobre el `DataFrame` que contiene los resultados, lo cual hará que se ejecute el plan.

Como se ha mencionado ya brevemente, se recogerán las distintas métricas de los procesos de la Spark UI. La Spark UI es un servicio web que se crea por cada proceso de Spark que está ejecutando en el sistema. Por defecto se abre en el puerto 4040 en el momento en el que tenemos el `SparkContext` creado, y nos proporciona información sobre el propio proceso: número de etapas, tareas, etc.

En nuestro caso vamos a recolectar la siguiente información de cada proceso: trabajos completados, etapas completadas y saltadas, total de tareas completadas, tiempo de tarea, tamaño de la entrada y tamaños de datos leídos y escritos en operaciones de *shuffle*. Esta información la recogeremos de las pestañas “Jobs”,

"Tasks" y "Executors".

Comparativa entre APIs

Explicado todo esto la comparativa queda tal que:

		Scala							
		Completed Jobs	Completed Stages	Skipped Stages	Completed Tasks	Task Time	Input	Shuffle Read	Shuffle Write
Consistencia	CSV	21	21	9	30	4s	47,1MiB	320KiB	320KiB
	Parquet particionado	25	25	9	1015	8s	24,1MiB	320KiB	320KiB
	Parquet no part	21	21	9	30	4s	3,2MiB	320KiB	320KiB
Mejor Piloto	CSV	29	29	43	29	4s	6,1MiB	2,4MiB	2,4MiB
	Parquet particionado	33	33	43	4390	22s	51,7MiB	2,2MiB	2,2MiB
	Parquet no part	29	29	43	29	3s	624KiB	2,1MiB	2,1MiB
Mejor Temporada	CSV	19	19	24	25	6s	17,6MiB	4,2MiB	4,2MiB
	Parquet particionado	22	22	24	2693	18s	34,5MiB	4,3MiB	4,3MiB
	Parquet no part	19	19	24	25	6s	847,2KiB	4,4MiB	4,4MiB
Dominio de constructores	CSV	8	8	2	8	0,8s	571,3KiB	2,5KiB	2,5KiB
	Parquet particionado	9	9	2	1005	6s	116,6KiB	2,8KiB	2,8KiB
	Parquet no part	8	8	2	8	1s	106,9KiB	2,5KiB	2,5KiB
Análisis de temporada	CSV	17	17	25	20	3s	16,9MiB	302,1KiB	302,1KiB
	Parquet particionado	19	19	25	1556	8s	828,1KiB	295,9KiB	2965,1KiB
	Parquet no part	17	17	25	20	2s	764,5KiB	356,7KiB	356,7KiB
Policía	CSV	4	4	2	27	36s	2,7GiB	8,2MiB	8,2MiB
	Parquet particionado	4	4	2	13	17s	55MiB	8,5MiB	8,5MiB
	Parquet no part	4	4	2	13	27s	124,4MiB	8,2MiB	8,2MiB

		Python							
		Completed Jobs	Completed Stages	Skipped Stages	Completed Tasks	Task Time	Input	Shuffle Read	Shuffle Write
Consistencia	CSV	21	21	9	30	4s	47,1MiB	317,8KiB	313,5KiB
	Parquet particionado	23	23	9	1015	9s	24,1MiB	317,8KiB	313,5KiB
	Parquet no part	21	21	9	30	4s	3,2MiB	317,8KiB	313,5KiB
Mejor Piloto	CSV	30	30	46	30	3s	6,3MiB	2,1MiB	2,1MiB
	Parquet particionado	34	34	46	4391	23s	51,7MiB	2,2MiB	2,2MiB
	Parquet no part	30	30	46	30	3s	624,4KiB	2,1MiB	2,1MiB
Mejor Temporada	CSV	42	42	118	48	4s	17,6MiB	4,2MiB	4,2MiB
	Parquet particionado	45	45	118	2716	17s	34,5MiB	4,3MiB	4,3MiB
	Parquet no part	42	42	118	48	4s	847,2KiB	4,2MiB	4,2MiB
Dominio de constructores	CSV	10	10	5	10	0,6s	571,3KiB	3,2KiB	3,2KiB
	Parquet particionado	11	11	5	1007	5s	116,6KiB	3,7KiB	3,1KiB
	Parquet no part	10	10	5	10	0,9s	106,9KiB	3,9KiB	3,6KiB
Análisis de temporada	CSV	19	19	36	22	2s	16,9MiB	304,4KiB	302,4KiB
	Parquet particionado	21	21	36	1558	8s	828,1KiB	298,2KiB	296,2KiB
	Parquet no part	19	19	36	22	2s	764,5KiB	360,9KiB	357,8KiB
Policía	CSV	4	4	2	27	36s	2,7GiB	8,2MiB	8,2MiB
	Parquet particionado	4	4	2	13	17s	55MiB	8,5MiB	8,5MiB
	Parquet no part	4	4	2	13	27s	124,4MiB	8,2MiB	8,2MiB

Figura 4.6: Comparativa entre APIs

Como los resultados son bastante similares para todas las queries, vamos a centrarnos en una para explicar lo que ha sucedido. Más adelante explicaré el motivo de tener una query extra llamada "Policía".

		Análisis de Temporada							
		Completed Jobs	Completed Stages	Skipped Stages	Completed Tasks	Task Time	Input	Shuffle Read	Shuffle Write
CSV	Scala	17	17	25	20	3s	16,9MiB	302,1KiB	302,1KiB
	Python	19	19	36	22	2s	16,9MiB	304,4KiB	302,4KiB
Parquet particionado	Scala	19	19	25	1556	8s	828,1KiB	295,9KiB	2965,1KiB
	Python	21	21	36	1558	8s	828,1KiB	298,2KiB	296,2KiB
Parquet no part	Scala	17	17	25	20	2s	764,5KiB	356,7KiB	356,7KiB
	Python	19	19	36	22	2s	764,5KiB	360,9KiB	357,8KiB

Figura 4.7: Rendimiento de la query de análisis de temporada

Podemos empezar centrándonos en los tiempos de tarea (Task Time). Se ob-

serva que los tiempos son muy bajos, de menos de 10 segundos para todas las mediciones. Esto puede significar varias cosas: que la máquina sobre la que se ejecutan las queries es demasiado potente o que el proceso dura muy poco, ya sea porque hace pocas operaciones o porque el conjunto de datos es muy pequeño. También podemos observar que la diferencia entre los tiempos de ejecución de ambas API es muy pequeña, siendo a veces inexistente.

Si miramos la parte del uso de memoria, podemos ver que de nuevo la diferencia entre API es prácticamente nula, a veces en el orden de Kibibytes. La diferencia más grande que podemos apreciar es a nivel de Trabajos, Etapas y Tareas, pero de nuevo la diferencia es muy baja. Esto se hace más evidente si nos fijamos en todas las queries en conjunto, donde el hecho de que la API de Scala genere menos Trabajos, Etapas y Tareas es algo constante salvo en la última query, y quizá podríamos llegar a la conclusión de que la API de Scala gestiona mejor esta parte del proceso, mientras que a veces su contraparte resulta ligeramente más eficiente en cuanto a tiempos de ejecución.

Se puede observar también que ocurre algo singular con la cantidad de trabajos y etapas completadas: normalmente el número es igual en ambos. Resulta curioso debido a que cada vez que se hace un *shuffle* se crea una etapa nueva. Estos números tendrían sentido si no se produjera ningún *shuffle*, sin embargo, podemos ver que sí se hace cierto *shuffle* si nos fijamos en la cantidad de datos escritos y leídos en estas operaciones.

Para hallar la respuesta tendremos que fijarnos en otra de las métricas obtenidas: las etapas saltadas. Si nos fijamos en la query del Mejor Piloto con datos en formato Parquet particionado, veremos que tenemos 33 Jobs completados, 33 tareas completadas y 43 saltadas. La clave está en estas últimas: si Spark detecta que hay etapas que ya ha realizado, se las salta. Esto es posible debido a que el resultado de cada etapa completada se guarda en caché precisamente para evitar repetir operaciones. Podemos ver esto más detalladamente en las figuras 4.8 y 4.9.

Podemos ver cómo en el Job 8 se hacen las operaciones “Scan parquet”, “WholeStageCodegen” y “Exchange”, todas ellas en la etapa 8. La última de ellas es un *shuffle*. Como comentamos antes, el Job acaba con uno de ellos.

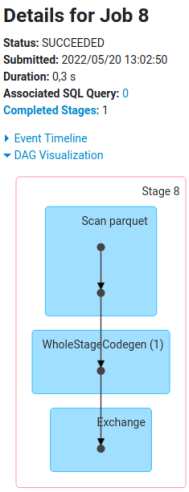


Figura 4.8: Grafo de operaciones para un Job inicial

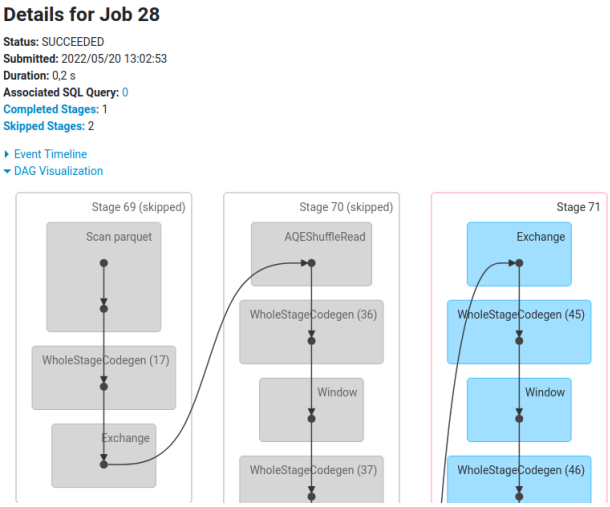


Figura 4.9: Grafo de operaciones para un Job al final

En las figuras vemos cómo en uno de los Jobs finales tenemos 3 etapas. En concreto es importante fijarse en la 69, que resulta ser la misma que la vista en la figura 4.8. Es por esto que el resultado ya está en caché y no es necesario volver a ejecutarla, lo cual provoca que se salte. Lo mismo pasa con la etapa 70, pero no con la 71, que es nueva y por tanto se ejecuta como sería natural.

Como he comentado antes brevemente, es posible que el conjunto de datos escogido para realizar las peticiones sea muy pequeño. Spark está pensado para manejar cantidades muy grandes de datos, en el orden de Gigabytes o más y, sin embargo, el conjunto de datos que escogí para realizar este trabajo es muy pequeño en comparación, en torno a los 20 Megabytes. Esto puede provocar que todo el procesamiento se haga en un solo núcleo y en una sola máquina, y esto

no es para lo que fue diseñado.

Es por esto último que decidí hacer una query más con un conjunto de datos un poco más grande. Este consiste en un solo archivo en formato CSV de unos 1,4 Gigabytes sobre delitos cometidos en el estado de Nueva York. La query realiza una lectura de este archivo, un filtrado sencillo, una intersección y una escritura final, y busca obtener una correspondencia entre delitos cometidos en Manhattan y el Bronx en 2015 y los delitos cometidos en Queens y Brooklyn dos horas después.

El código de la query es la siguiente:

```
val entiretable = spark.read.format("csv")
       .option("header", "true")
       .option("sep", ",")
       .load("../data/police/NYPD_Complaint_Data_Historic.csv")
       .withColumn("CMPLNT_FR_DATETIME", to_timestamp(concat(col("CMPLNT_FR_DT"), lit(" "),
            col("CMPLNT_FR_TM")), "MM/dd/yyyy HH:mm:ss"))
       .withColumn("COMPLETE_LOC", concat(col("LOC_OF_OCCUR_DESC"), lit(" "),
            col("PREM_TYP_DESC")))

val crimesManh = entiretable
       .withColumn("CMPLNT_FR_DATETIME_2H_AFTER", col("CMPLNT_FR_DATETIME") + expr("INTERVAL 2
            HOURS"))
       .where((col("BORO_NM") === "MANHATTAN" || col("BORO_NM") === "BRONX") &&
            year(col("CMPLNT_FR_DATETIME")) === 2015)
       .withColumnRenamed("CMPLNT_FR_DATETIME", "CMPLNT_FR_DATETIME_MANH")
       .withColumnRenamed("CMPLNT_FR_DATETIME_2H_AFTER", "CMPLNT_FR_DATETIME_2H_AFTER_MANH")
       .withColumnRenamed("CMPLNT_FR_DT", "CMPLNT_FR_DT_MANH")
       .withColumnRenamed("CMPLNT_NUM", "CMPLNT_NUM_MANH")
       .select("CMPLNT_FR_DATETIME_MANH", "CMPLNT_FR_DATETIME_2H_AFTER_MANH",
            "CMPLNT_FR_DT_MANH", "CMPLNT_NUM_MANH")

val crimesQue = entiretable
       .withColumn("CMPLNT_FR_DATETIME_2H_AFTER_QUE", col("CMPLNT_FR_DATETIME") +
            expr("INTERVAL 2 HOURS"))
       .where((col("BORO_NM") === "QUEENS" || col("BORO_NM") === "BROOKLYN") &&
            year(col("CMPLNT_FR_DATETIME")) === 2015)
       .withColumnRenamed("CMPLNT_FR_DATETIME", "CMPLNT_FR_DATETIME_QUE")
       .withColumnRenamed("CMPLNT_FR_DATETIME_2H_AFTER", "CMPLNT_FR_DATETIME_2H_AFTER_QUE")
       .withColumnRenamed("CMPLNT_FR_DT", "CMPLNT_FR_DT_QUE")
       .withColumnRenamed("CMPLNT_NUM", "CMPLNT_NUM_QUE")
       .select("CMPLNT_FR_DATETIME_QUE", "CMPLNT_FR_DATETIME_2H_AFTER_QUE", "CMPLNT_FR_DT_QUE",
            "CMPLNT_NUM_QUE")

val results = crimesManh
       .join(crimesQue, crimesManh.col("CMPLNT_FR_DT_MANH") ===
            crimesQue.col("CMPLNT_FR_DT_QUE"), "fullouter")
       .where(col("CMPLNT_FR_DATETIME_QUE") <= col("CMPLNT_FR_DATETIME_2H_AFTER_MANH") &&
col("CMPLNT_FR_DATETIME_QUE") >= col("CMPLNT_FR_DATETIME_MANH"))
       .select("CMPLNT_FR_DATETIME_MANH", "CMPLNT_NUM_MANH", "CMPLNT_FR_DATETIME_QUE",
            "CMPLNT_NUM_QUE")
```

Simplemente ejecutando este proceso podemos ver una diferencia notable con respecto a las otras queries, y quizá, al ser más grande el conjunto de datos, sea más representativo de la realidad. Podemos ver las métricas recogidas sobre este proceso en la siguiente imagen:

		Query Policía							
		Completed Jobs	Completed Stages	Skipped Stages	Completed Tasks	Task Time	Input	Shuffle Read	Shuffle Write
CSV	Scala	4	4	2	27	36s	2,7GiB	8,2MiB	8,2MiB
	Python	4	4	2	27	36s	2,7GiB	8,2MiB	8,2MiB
Parquet particionado	Scala	4	4	2	13	17s	55MiB	8,5MiB	8,5MiB
	Python	4	4	2	13	17s	55MiB	8,5MiB	8,5MiB
Parquet no part	Scala	4	4	2	13	27s	124,4MiB	8,2MiB	8,2MiB
	Python	4	4	2	13	27s	124,4MiB	8,2MiB	8,2MiB

Figura 4.10: Rendimiento de la query extra

Podemos deducir entonces que no existe una diferencia notable entre las API con los tamaños de datos que estamos manejando. Lo ideal sería realizar una query sobre un conjunto de datos que ronde las decenas o centenares de Gigabytes con el máximo paralelismo posible tanto a nivel de las propias máquinas (muchos núcleos y memoria dedicadas) como a nivel de nodos (varias máquinas conectadas entre sí).

Cabe destacar aquí que ambas API son interfaces de Apache Spark y, por lo tanto, dados dos `SparkContext` creados con los mismos parámetros y opciones que tengan asignados los mismos recursos, el rendimiento en teoría tendría que ser el mismo. Esto sería así salvo en un caso: que comparemos una API no nativa (como Python o R) a una nativa (Scala o Java) y utilicemos UDFs. En este caso al usar una API no nativa el modo en el que se aplica una UDF es diferente.

Cuando usamos una API nativa, el código de la UDF compila directamente para la máquina virtual de Java (JVM a partir de ahora) y no es necesario llevar a cabo más operaciones: todo se queda en la JVM. Sin embargo, al usar una API no nativa, la UDF no puede compilar en la JVM y por lo tanto se lleva fuera de la misma para aplicar la función. Por ello es un proceso costoso en el que tenemos que pasaremos de tener los datos dentro de la JVM, para luego serializarlos fuera de ella, aplicarla función y deserializar para volver a tenerlos dentro de la JVM. Como las UDF aplican funciones fila a fila cuanto más grande sea el conjunto de datos sobre el que se apliquen este tipo de funciones, mayor será el sobrecoste de usar una API no-nativa. Este proceso se denomina Serialización/Deserialización[1].

Una manera que tenemos para comprobar cómo afecta el paralelismo en los procesos de Spark resulta relativamente sencillo de implementar: podemos particionar la entrada y observar qué métricas nos arroja la Spark UI. Paralelizar la entrada en teoría debería permitir que Spark dedicase más tareas a la lectura y por lo tanto más recursos. En la siguiente sección veremos los detalles de cómo afecta esto al rendimiento.

Comparativa entre tipos de fuentes de datos

Hemos visto ya cómo afecta la propia API al rendimiento de las queries. Ahora vamos a intentar simular un paralelismo extra haciendo que en lugar de

[1] `https://medium.com/quantumblack/spark-udf-deep-insights-in-performance-f0a95a4d8c62`

leer únicamente de un archivo lea la misma tabla, pero particionada de forma que tenga que, teóricamente, utilizar más tareas y recursos para leer.

Como he comentado brevemente antes, hay que tener especial cuidado a la hora de decidir qué columnas se usan para particionar. Es posible incluso que no nos merezca la pena particionar los datos. Vamos a ver un ejemplo de un "buen particionado" y uno "malo".

Para ver el ejemplo del "mal" particionado, vamos a referir de nuevo a la figura 4.7. Si en lugar de fijarnos en la comparativa entre APIs nos centramos en las diferencias entre formatos de entrada, podemos ver algo llamativo: al leer de un Parquet particionado se tarda más del doble que leyendo de un CSV y cuatro veces más que leyendo el Parquet sin particionar.

En las queries sobre el conjunto de datos de la Fórmula 1 se decidió particionar la mayoría de tablas según el campo `raceId`, principalmente porque lo primero que se hace en muchas de ellas es filtrar por ese campo. Esto por norma general es buena práctica, ya que al particionar los datos según ese campo a la hora de llevar a cabo el procesado Spark tendrá un filtrado menos que hacer, pero las métricas obtenidas nos dicen otra cosa, ya que se esperaría que la query tardase menos y, lo más importante de todo, que leyese menos datos de disco.

Lo primero no se cumple y lo segundo se cumple en parte: lee mucho menos que al usar un CSV pero no se nota una diferencia notable entre particionar o no. Empecemos mirando el por qué de los tiempos de ejecución tan altos.

7 (a3ad8bf1-1b65-4aa9-919c-0432bbcfe51f)	broadcast exchange (runId a3ad8bf1-1b65-4aa9-919c-0432bbcfe51f) $anonfun$withThreadLocalCaptured$1 at FutureTask.java:266	2022/05/08 19:23:39	0,3 s	1/1	1/1
6 (74f2c3ab-f781-40d9-a17a-fe045d6fae3b)	broadcast exchange (runId 74f2c3ab-f781-40d9-a17a-fe045d6fae3b) $anonfun$withThreadLocalCaptured$1 at FutureTask.java:266	2022/05/08 19:23:39	0,3 s	1/1	1/1
5	parquet at cmd7.sc:1 parquet at cmd7.sc:1	2022/05/08 19:23:38	18 ms	1/1	1/1
4	Listing leaf files and directories for 1057 paths: file:/home/oscar/TFG/Spark-TFG/data/parquet/results.parquet/raceId=265, ... parquet at cmd7.sc:1	2022/05/08 19:23:37	0,8 s	1/1	1057/1057
3	parquet at cmd6.sc:1 parquet at cmd6.sc:1	2022/05/08 19:23:37	21 ms	1/1	1/1
2	Listing leaf files and directories for 476 paths: file:/home/oscar/TFG/Spark-TFG/data/parquet/lap_times.parquet/raceId=891, ... parquet at cmd6.sc:1	2022/05/08 19:23:36	0,5 s	1/1	476/476
1	parquet at cmd4.sc:1 parquet at cmd4.sc:1	2022/05/08 19:23:36	23 ms	1/1	1/1
0	parquet at cmd3.sc:1 parquet at cmd3.sc:1	2022/05/08 19:23:34	0,4 s	1/1	1/1

Figura 4.11: Tareas al leer Parquet particionado

Como podemos observar en la figura anterior, que está obtenida de la pestaña "Jobs"de la Spark UI de un proceso que utiliza Parquet particionado, se pierde mucho tiempo listando el directorio de lectura. Además de eso, se crea una tarea para cada elemento del directorio en el que se encuentra la tabla (en este caso, 1057).

El otro problema que hemos visto es que no obtenemos una ganancia sustancial particionando si nos fijamos en la cantidad de datos leídos. Esto está relacionado con lo que acabamos de ver.

Como comentamos anteriormente, el conjunto de datos usado originalmente era demasiado pequeño para Spark, y por tanto llegamos a la conclusión de que no sacábamos todo el rendimiento que se podría sacar de esta API. Otro problema surgido de esto es que al particionar los ficheros Parquet creamos muchos archivos con muy pocos datos dentro. Por poner un ejemplo, la tabla `constructor_standings` particionada por `raceId` tendrá N subcarpetas dependiendo del número de valores que tome el campo clave, y cada una de las particiones tendrá muy pocos datos en ella, en torno a 10 entradas de media, ya que es la cantidad de constructores que participan en una carrera.

Esto último saca a relucir otro problema: que al particionar una tabla tan pequeña es posible que los metadatos añadidos al utilizar Parquet como formato de entrada ocupen más que la propia información que contiene la tabla. La ya mencionada tabla `constructor_standings` en formato CSV pesa en torno a 300KB. Usando Parquet sin particionar pesa en torno a 109KB y al particionar pesa exactamente 6MB.

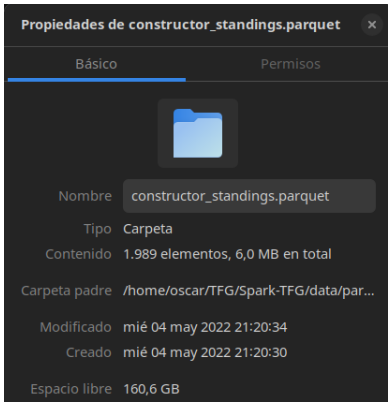


Figura 4.12: Tamaño Parquet particionado

Por lo tanto, podemos llegar a la conclusión de que este superávit de tiempo de ejecución puede estar producido por el hecho de tener que leer tantos archivos. Si los archivos tuvieran más información y la query llevase a cabo un procesamiento más pesado lo más probable es que no se llegase a notar. Quizá leer un archivo grande no sea mucho más lento que leer uno pequeño y por tanto en el cómputo total fuese asumible. Sin embargo, al hacer operaciones a tan pequeña escala se notan mucho los tiempos gastados en listar directorios y leer particiones tan pequeñas.

Vamos a intentar mejorar el rendimiento de alguna query cambiando de modelo de particionado. En lugar de particionar gran parte de las tablas según `raceId`,

vamos a particionar solamente `lap_times`, y lo vamos a hacer según la columna `driverId`. El motivo de elegir esta combinación es el siguiente: la tabla en cuestión compone el 75 % del conjunto de datos (15MB de los 20MB que tenemos aproximadamente), y `driverId` tiene únicamente 135 entradas. Hay que mencionar que la tabla `drivers` tiene más de 800 entradas, pero como `lap_times` sólo tiene datos a partir del año 1996, la cantidad de pilotos se reduce drásticamente.

Veamos ahora los resultados obtenidos y comparemos cara a cara con su contraparte "mal" particionada:

		Mejor Temporada							
	Particionado	Completed Jobs	Completed Stages	Skipped Stages	Completed Tasks	Task Time	Input	Shuffle Read	Shuffle Write
Parquet particionado	raceId	22	22	24	2693	18s	34,5MiB	4,3MiB	4,3MiB
	driverId	20	20	24	161	7s	5,3MiB	4,1MiB	4,1MiB

Figura 4.13: Comparativa de particionado

Se puede observar una ganancia en todos los aspectos. La cantidad de Jobs es menor, hay menos etapas y, ante todo, la cantidad de tareas completadas se ha reducido drásticamente, al igual que el tiempo de ejecución, reducido a la mitad, y la cantidad de datos leídos, que es 6,5 veces menor.

Podemos llegar a la conclusión entonces de que el cómo particionemos los datos afecta directamente al rendimiento de nuestras peticiones.

Si centramos nuestra atención a la figura 4.10, que utiliza un conjunto de datos mayor, veremos que leer de Parquet de nuevo siempre es mejor que leer un CSV, y por lo tanto concluimos que particionar si merece la pena.

Al tener un conjunto de datos más grande, leer de CSV resulta muy pesado ya que Spark tiene que leer línea a línea, columna por columna, inferir esquemas, etc. Al usar Parquet gran parte de este preprocesado se vuelve innecesario: el tipo de las columnas lo tenemos en los metadatos y solo necesitamos leer lo que realmente vamos a usar gracias a las optimizaciones de Spark.

Tanto las ganancias en Task Time como en datos leídos de disco son muy grandes. Entre leer datos de un CSV y leer de un fichero Parquet particionado el Task Time se reduce aproximadamente a la mitad, y leemos 44 veces menos datos de disco.

Cabe destacar que esta tabla se ha particionado según un campo que toma una cantidad muy limitada de valores, y por lo tanto las particiones siguen siendo relativamente grandes, al contrario que en el particionado anterior.

De este apartado podemos, por tanto, sacar varias conclusiones rápidas:

- No existe una diferencia real entre APIs, especialmente cuanto más grande es el conjunto de datos siempre que no usemos UDFs a menudo.
- Spark no funciona bien con pocos datos. Cuanto más, mejor.

- Cuanto más paralelismo, mejor.
- Para particionar correctamente una tabla es necesario conocer el contexto y no hacer las particiones demasiado pequeñas.

En el siguiente capítulo elaboraré más estas conclusiones.

5

Conclusiones y trabajos futuros

En este trabajo hemos visto cómo se usa la API de Spark tanto en Scala como Python y hemos usado ambas para resolver cuestiones relacionadas con el dominio de la Fórmula 1. Tras obtener los resultados, estos han sido analizados con ayuda de la librería Plotly, que nos ayudó a generar gráficos sencillos.

También hemos visto cómo podemos migrar consultas de una API a otra, en este caso de Scala Spark a PySpark, ayudándonos de una serie de expresiones regulares y aplicándolas de forma secuencial con un *script* que automatiza esta tarea.

Habiendo visto ya cómo se programa en ambas APIs, hemos pasado a analizar su rendimiento, y nos hemos encontrado con que el conjunto de datos que hemos usado para las consultas es demasiado pequeño y no utiliza todo el potencial que nos brinda Spark. Hemos visto también cómo el campo que utilicemos para particionar una tabla de entrada puede afectar al rendimiento.

Tras esto, hemos utilizado un conjunto de datos más grande y hemos visto que los datos obtenidos ya sí que resultaban más coherentes, aunque todavía no llegamos a utilizar todos los recursos de estas APIs.

Spark es una librería que permite acceder a fuentes de datos y realizar procesamiento tanto en paralelo en la misma máquina como de forma distribuida entre distintos nodos, y cuanto más grande sea la entrada y los datos que se manejan, más beneficio se saca a esta API. Quizá al tratarse de un conjunto de datos tan pequeño hubiese resultado más eficiente usar una librería como Pandas, que aunque no realice tantas optimizaciones como Spark resulta menos pesado y hubiese resultado en procesos más eficientes.

Sin embargo, para entornos en los que se manejan cantidades muy grandes de datos creo que Spark sería la mejor opción, ya que las opciones que proporciona a la hora de paralelizar los procesos y distribuir el trabajo entre varios nodos resulta muy interesante si se tiene la infraestructura necesaria para darle uso.

Como hemos visto, la diferencia de rendimiento entre ambas API resulta prácticamente nula cuando se usan tamaños de entrada en el orden de Gigabytes, y sospecho que cuanto más grande sea el tamaño de entrada la diferencia será aún menor si es posible. Un punto clave en el que puede estar la diferencia entre ellas puede ser el uso de UDF ya que, aunque en las queries en las que las hemos usado no hemos notado diferencias notables, probablemente con una carga más grande de datos seguramente veamos que la balanza se inclina más a la API de Scala.

Como en rendimiento ambas API son idénticas en las comparaciones realizadas, entonces para llegar a una conclusión sobre cuál es mejor tenemos que comparar la experiencia de programar queries en Python y Scala. Esto roza el plano subjetivo, así que cada persona puede tener una opinión distinta al respecto, pero personalmente creo que la API de Scala es mejor en este aspecto.

En mi opinión la experiencia de programar queries de Spark en Scala resulta más natural que en su contraparte ya que la API fue diseñada inicialmente para ella, y por tanto tenía que cumplir con la filosofía del lenguaje. Si nos fijamos en otras librerías del estilo de Pandas, que fue diseñado con Python en mente, veremos que hacer queries resulta muy parecido a operar con listas o matrices. Veremos bucles y condiciones mientras que en Spark (y Scala) nos centramos en las transformaciones que debemos aplicar al conjunto de datos para obtener los resultados deseados al final.

En resumidas cuentas, programar queries usando PySpark puede resultar un choque de paradigmas de programación y quizá a alguien acostumbrado a programar en Python le resulte más intuitivo y natural usar una librería como Pandas.

Además, usando esta API en Scala disponemos de herramientas extra como las clases implícitas, que nos permiten extender la funcionalidad de clases como `DataFrame`, resultando así en un código más legible y limpio.

Hablando de esto último, en mi opinión el código queda mucho más limpio si usamos la API de Scala. Al migrar queries de ella a PySpark hemos visto que es un proceso en el que se añaden caracteres y, en general, ruido al código. Hacer el camino inverso por tanto será un proceso de limpieza.

Por estos motivos acababa usando Scala Spark en tareas secundarias. El código quedaba más limpio y me resultaba más rápido de desarrollar al no tener que preocuparme de añadir esos caracteres extra y centrándome en qué transformaciones quería hacer.

Por esto último, en mi opinión, la API de Scala es mejor que la de Python. Quizá no hayamos visto que se ahorre mucho tiempo de ejecución, pero definitivamente ahorramos en tiempo de desarrollo.

En general he disfrutado del desarrollo de este trabajo, a pesar de que ciertas etapas han podido resultar tediosas. En concreto las partes que más he disfrutado han sido la de la migración de queries y la comparativa final. La primera me ha permitido familiarizarme con la API de Python, ya que aunque el orden de este trabajo no lo refleje, se programaron todas las queries en Scala primero y una vez se vio que funcionaba y arrojaba los datos que se buscaban se migraron mediante el uso de las expresiones regulares. Esto también me permitió familiarizarme con ellas, ya que a pesar de haberlas visto a lo largo de la carrera nunca me fueron necesarias para el desarrollo de ninguna práctica, así que no pude afianzar las ideas.

La parte de la automatización del proceso de migración también ha sido bastante útil ya que me ha permitido aprender más acerca de los distintos estándares de expresiones regulares que se han utilizado y cómo se podrían aplicar todas de forma automática mediante un *script* de *shell*.

Por otro lado la comparativa me permitió indagar un poco más en cómo funciona Spark por dentro, mirar cómo afectan factores que van más allá de la programación de queries al rendimiento: ya sea el particionado de los datos y su formato o la cantidad de recursos de los que se dispone.

La parte más tediosa fue la inicial. Familiarizarme con Spark, aprender los conceptos básicos de Scala o informarme sobre el dominio del el que iban a tratar las cuestiones planteadas como queries. También se me hizo complicado y largo el proceso de programación de las queries, pero principalmente achaco esto último a factores externos a la realzación del propio proyecto.

Finalmente, me gustaría comentar que he aprendido mucho con la realización de este proyecto y he adquirido conocimientos que seguro que más adelante me serán muy útiles en el ámbito profesional.

5.1. Trabajos futuros

A raíz de todo lo visto a lo largo de esta memoria quedan varios asuntos pendientes que se podrían desarrollar en trabajos futuros.

Respecto a la comparativa, queda pendiente comparar ambas API con un conjunto de datos más grande, idealmente en las decenas o cientos de Gigabytes y varios nodos que utilicen todo el potencial de Spark. Hemos visto que los resultados se estabilizan cuando aumentamos el tamaño de los datos de entrada. Por ello y por lo que ya hemos comentado anteriormente, lo justo sería realizar una comparativa parecida a la ya realizada pero con un contexto en el que Spark funcione de forma óptima y en un entorno lo más parecido posible a los que se tuvieron en mente a la hora de su diseño.

Por otro lado podríamos intentar automatizar la migración de queries "importando" el Job de una instancia de Spark a otra. Esto puede ser posible porque, como ya hemos comentado, PySpark es una interfaz de Spark al mismo nivel que la de Scala, lo cual implica que los planes que genere una API deberían funcionar en otra. Requeriría más investigación por mi parte, pero proporcionaría una precisión perfecta al descartar el factor humano que siempre existe al realizar una migración manual.

Bibliografía

[CS14] Scott Chacon y Ben Straub. *Pro Git, Second Edition*. Apress, 2014.

[CZ18] Bill Chambers y Matei Zaharia. *Spark: The Definitive Guide*. O'Reilly Media, Inc., 2018.

[OSVS19] Martin Odersky, Lex Spoon, Bill Venners y Frank Sommers. *Programming in Scala, Fourth Edition*. artima, 2019.

[PyS] Pyspark api. `https://spark.apache.org/docs/latest/api/python/`.

[Rao20] Rohan Rao. Formula 1 world championship (1950 - 2022). 2020. URL `https://www.kaggle.com/rohanrao/formula-1-world-championship-1950-2020`.

[Sca] Scala api. `https://www.scala-lang.org/api/2.13.3/index.html`.

[Spa] Scala spark api. `https://spark.apache.org/docs/latest/api/scala/`.

Apéndices

A

Programación de queries

A.1. Programación de queries en Scala Spark

A.1.1. Análisis de temporada por piloto

Esta query consiste en obtener una serie de métricas para cada piloto de una determinada temporada. Aunque no es un análisis muy en profundidad, se van a calcular varias métricas que nos ayudarán a arrojar un poco más de luz a la temporada que deseemos analizar. En nuestro caso será la temporada 2021 que, al resultar tan interesante de ver, suscitó preguntas que buscaremos resolver.

Primero necesitaremos información básica del piloto como su código o sus puntos en el campeonato. Tras esto veremos cómo de igualada ha estado la temporada calculando el porcentaje de puntos del resto de pilotos respecto al campeón. Veremos también cuántos puntos ha conseguido de media, su posición media al acabar la carrera y varias más como el porcentaje de vueltas lideradas o el total de adelantamientos. Entraremos más en detalle en la descripción de esta query.

El código es el siguiente:

```
def queryAnalisisTemporada(season: Int, races: DataFrame, lap_times: DataFrame, results:
    DataFrame): DataFrame = {

      val driverWindow = Window.partitionBy("driverId")
      val seasonWindow = Window.partitionBy("year")
      val driverRaceWindow = Window.partitionBy("driverId", "raceId")
      val raceDriverLapWindow = Window.partitionBy("driverId", "raceId").orderBy("lap")

      val races_filtered = races
            .where(col("year") === season)

      val driverStats = lap_times
```

```
                .withColumn("position", col("position").cast(IntegerType))
                .withColumn("lap", col("lap").cast(IntegerType))
                .join(races_filtered, "raceId")

                .withColumn("positionNextLap", lead(col("position"), 1).over(raceDriverLapWindow))
                .withColumn("positionsGainedLap", when(col("positionNextLap") < col("position") ,
                    abs(col("position") - col("positionNextLap"))).otherwise(0))
                .withColumn("positionsLostLap", when(col("positionNextLap") > col("position"),
                    abs(col("position") - col("positionNextLap"))).otherwise(0))
                .withColumn("positionsGained",
                    sum(col("positionsGainedLap")).over(driverRaceWindow))
                .withColumn("positionsLost", sum(col("positionsLostLap")).over(driverRaceWindow))
                .withColumn("lapLeader", when(col("position") === 1, 1).otherwise(0))
                .withColumn("lapsLed", sum(col("lapLeader")).over(driverWindow))
                .withColumn("totalLaps", sum(col("lapLeader")).over(seasonWindow))
                .withColumn("percLapsLed", round(col("lapsLed") / col("totalLaps"), 2))
                .select("raceId", "driverId", "positionsGained", "positionsLost", "lapsLed",
                    "percLapsLed")
                .dropDuplicates()

        results
                .withColumn("position", col("position").cast(IntegerType))
                .withColumn("grid", col("grid").cast(IntegerType))
                .withColumn("points", col("points").cast(IntegerType))

                .join(races_filtered, "raceId")
                .join(driverStats, Seq("raceId", "driverId"), "left")
                .join(drivers, "driverId")

                .withColumn("podium", when(col("position") === 1 || col("position") === 2
                    ||col("position") === 3, lit(1)).otherwise(lit(0)))
                .withColumn("averagePoints", round(avg(col("points")).over(driverWindow), 2))
                .withColumn("maxAvgPoints", max(col("averagePoints")).over(seasonWindow))

                .select(
                       col("code"),
                       sum(col("points")).over(driverWindow).as("champPoints"),
                       col("averagePoints"),
                       round(col("averagePoints") / col("maxAvgPoints"),2).as("pointPercent"),
                       sum(col("podium")).over(driverWindow).as("totalPodiums"),
                       round(avg(col("position")).over(driverWindow), 2).as("avgPosition"),
                       round(sum(col("podium")).over(driverWindow) /
                            count(col("podium")).over(driverWindow), 2).as("podiumPercent"),
                       round(avg(col("position") - col("grid")).over(driverWindow),
                            2).as("positionDelta"),
                       round(avg(col("positionsLost")).over(driverWindow),
                            2).as("avgPositionsLost"),
                       round(avg(col("positionsGained")).over(driverWindow),
                            2).as("avgPositionsWon"),
                       sum(col("positionsLost")).over(driverWindow).as("totalPositionsLost"),
                       sum(col("positionsGained")).over(driverWindow).as("totalPositionsWon"),
                       col("lapsLed"),
                   col("percLapsLed")
                   )

                .na.fill(0)
                .dropDuplicates(Seq("code"))
}
```

Descripción detallada

De nuevo, lo primero es obtener las distintas carreras que se han disputado en la temporada deseada. Para ello y como ya quedó explicado anteriormente, usaremos la tabla `races`, que

filtraremos según la columna `year`.

Una vez obtenidas las carreras, necesitamos obtener información personal de los pilotos para más adelante sustituir su identificador numérico por el código de tres letras personal. Toda esta información la recibimos en el `DataFrame drivers`.

Tras esto, pasamos a crear las ventanas de datos que necesitaremos. En este caso, vamos a necesitar particionar los datos por piloto, por año, por piloto y carrera y de nuevo por piloto y carrera pero ordenando por vueltas.

```
val driverWindow = Window.partitionBy("driverId")
val seasonWindow = Window.partitionBy("year")
val driverRaceWindow = Window.partitionBy("driverId", "raceId")
val raceDriverLapWindow = driverRaceWindow.orderBy("lap")
```

Antes de continuar, necesitaremos obtener ciertos valores estadísticos relacionados con las posiciones del piloto a lo largo de la temporada. En concreto queremos obtener todas las posiciones ganadas y perdidas a lo largo de la carrera y, ya que usaríamos la misma tabla, el número y porcentaje de vueltas que ha liderado a lo largo de la temporada.

Para ello utilizaremos la tabla `lap_times`, que filtraremos según las carreras de la temporada con el filtro que conseguimos antes. Para realizar estos cálculos, es importante además que las columnas `lap` y `position` sean enteros, ya que vamos a hacer comparaciones y sumatorios.

Todo esto lo podemos hacer de la siguiente manera:

```
val driverStats = lap_times
      .withColumn("position", col("position").cast(IntegerType))
      .withColumn("lap", col("lap").cast(IntegerType))
      .join(races, "raceId")
```

Para calcular si un piloto ha ganado o ha perdido su posición en una vuelta, tenemos que saber cuál es su posición en la vuelta siguiente. Para ello podemos utilizar la función `lag` de la siguiente manera:

```
      .withColumn("positionNextLap", lead(col("position"), 1).over(raceDriverLapWindow))
```

Con esto podemos calcular las vueltas ganadas o perdidas en cada vuelta de la siguiente manera:

```
      .withColumn("positionsGainedLap", when(col("positionNextLap") < col("position") ,
          abs(col("position") - col("positionNextLap"))).otherwise(0))
      .withColumn("positionsLostLap", when(col("positionNextLap") > col("position"),
          abs(col("position") - col("positionNextLap"))).otherwise(0))
```

De esta manera, aplicando la función `abs`, que nos devuelve el valor absoluto de la columna que se pasa como argumento, conseguimos dos de las métricas que buscábamos.

Para las otras dos métricas tendremos primero que conseguir las vueltas donde el piloto lideraba la carrera. Como tenemos información de todas las vueltas que han dado todos los pilotos en la temporada, obtener esta información no resulta complicado. Para esta query se ha realizado lo siguiente:

```
      .withColumn("lapLeader", when(col("position") === 1, 1).otherwise(0))
```

Podemos entender esta columna a la que he llamado `lapLeader` de la siguiente manera: si el piloto ha liderado la vuelta, valdrá `1` y en caso contrario `0`. Esto resulta muy útil ya que podemos obtener el número de vueltas que un piloto ha liderado al hacer un sumatorio de todos los elementos de esta columna particionando por piloto, como se puede ver a continuación:

```
        .withColumn("lapsLed", sum(col("lapLeader")).over(driverWindow))
```

Tras esto podemos obtener el porcentaje de vueltas que un piloto ha liderado dividiendo este valor recién calculado entre el total de vueltas dadas.

```
        .withColumn("totalLaps", sum(col("lapLeader")).over(seasonWindow))
        .withColumn("percLapsLed", round(col("lapsLed") / col("totalLaps"), 2))
```

Finalmente, eliminamos duplicados y presentamos el `DataFrame` como consideremos oportuno. En este caso, necesitaré los cuatro valores calculados, el identificador de piloto y el de carrera.

```
        .select("raceId", "driverId", "positionsGained", "positionsLost", "lapsLed",
            "percLapsLed")
        .dropDuplicates()
```

El siguiente paso es obtener la tabla final, y para ello partiremos de la tabla `results`. De nuevo necesitaremos convertir a entero ciertas columnas. En este caso `position`, `grid` y `points`.

Filtramos las carreras de la temporada en cuestión y ampliamos la información con la tabla `driverStats` que acabamos de obtener y `drivers`, esta última para convertir el id de piloto en su código de 3 caracteres. Todo esto lo hacemos de la siguiente manera:

```
results
        .withColumn("position", col("position").cast(IntegerType))
        .withColumn("grid", col("grid").cast(IntegerType))
        .withColumn("points", col("points").cast(IntegerType))

        .join(races, "raceId")
        .join(driverStats, Seq("raceId", "driverId"), "left")
        .join(drivers, "driverId")
```

Para esta query tendremos que calcular el número de puntos obtenidos por el piloto, la media de puntos, el porcentaje de puntos en relación al ganador del campeonato, el número total de podios, el porcentaje de veces que el piloto ha acabado en el podio, el diferencial entre la posición de salida y en la que termina, la media y el total de posiciones perdidas y ganadas y el número y porcentaje de vueltas lideradas.

Antes de nada tenemos que calcular 3 valores que servirán para más adelante calcular el resto de métricas. Estos son la media de puntos, la media de puntos más alta y si el piloto ha terminado en podio o no.

De forma similar a lo visto anteriormente, para ver si un piloto ha acabado en podio podemos crear una columna llamada `podium`, que valdrá `1` si el piloto acaba en las tres primeras posiciones y `0` en caso contrario.

```
        .withColumn("podium", when(col("position") === 1 || col("position") === 2
            ||col("position") === 3, lit(1)).otherwise(lit(0)))
```

La media de puntos es sencilla de calcular, y la media más alta se calcula sobre el valor anterior de la siguiente manera:

```
        .withColumn("averagePoints", round(avg(col("points")).over(driverWindow), 2))
        .withColumn("maxAvgPoints", max(col("averagePoints")).over(seasonWindow))
```

Una vez obtenidos estos 3 valores podemos calcular el resto. En general todos son o bien sumatorios o medias sobre ventanas de datos concretas. Para presentar los datos de manera más accesible, se redondean a dos decimales usando la funciíon `round`.

Llegados a este punto me gustaría detenerme para explicar la función `select`. A simple vista parece sencilla si la usamos como lo haríamos en SQL o como hemos hecho hasta ahora, pero existe otra manera de usarla. Si nos vamos a la definición de la función en la documentación de Spark, veremos que le podemos pasar o bien varios String o varios objetos de tipo `Column`. Si utilizamos esta función de esta última manera, se puede obtener una cierta mejora en el plan de Spark y, por lo tanto, es recomendable utilizarla así.

En este caso, he decidido mostrar cómo finalizaríamos la query usando un `select` que recibe columnas en lugar de String.

```
.select(
        col("code"),
        sum(col("points")).over(driverWindow).as("champPoints"),
        col("averagePoints"),
        round(col("averagePoints") / col("maxAvgPoints"),2).as("pointPercent"),
        sum(col("podium")).over(driverWindow).as("totalPodiums"),
        round(sum(col("podium")).over(driverWindow) / count(col("podium")).over(driverWindow),
             2).as("podiumPercent"),
        round(avg(col("position") - col("grid")).over(driverWindow), 2).as("positionDelta"),
        round(avg(col("positionsLost")).over(driverWindow), 2).as("avgPositionsLost"),
        round(avg(col("positionsGained")).over(driverWindow), 2).as("avgPositionsWon"),
        sum(col("positionsLost")).over(driverWindow).as("totalPositionsLost"),
        sum(col("positionsGained")).over(driverWindow).as("totalPositionsWon"),
        col("lapsLed"),
        col("percLapsLed")
)
```

Como se puede observar, podemos pasarle una columna directamente o una operación sobre ciertas columnas que devuelva un objeto de tipo `Column` a la que damos nombre con `as`.

Para calcular todas estas métricas se utiliza siempre una ventana de datos que particiona por piloto, y en los que no se particiona es porque ya existe solamente una entrada por piloto.

Como queda algún registro con valor `null`, nos convendría tratar de alguna manera estos casos, ya que se pretende representar todas estas métricas gráficamente. Para ello se utilizan las funciones presentes en el paquete `na`. Hay tres funciones que cubrirán la mayoría de casos de uso que necesitemos. Estas son: `fill`, `replace` y `drop`. Su función la indica el nombre: `fill` rellena los nulos con un literal que pasamos por parámetro, `replace` sustituye los nulos según se especifique y `drop` elimina las filas que contengan nulos, con la opción de especificar en qué columnas comprueba la existencia de estos valores.

Para la función `replace` he encontrado muy útil que puede recibir como parámetro un objeto de tipo `Map`, en el que la clave será el nombre de la columna y el valor será el valor que queramos que sustituya a los nulos. Un ejemplo podría ser el siguiente: dado un DataFrame en el que tenemos tres columnas llamadas `id`, `name` y `salary`, si utilizásemos la función `na.replace()` pasándole como parámetro `Map(‘‘name’’ -->‘‘Pedro’’, ‘‘salary’’ -->0)` significaría que en la columna `name` los nulos pasarán a valer ”Pedroz para la columna `salary`, los valores nulos

valdrán cero.

En nuestro caso, como se ha observado que los nulos aparecen cuando el piloto no tiene ninguna vuelta que haya liderado y solo en ese caso, podemos utilizar `na.fill(0)` para solventar el problema.

Tras esto solo quedaría eliminar entradas duplicadas y ordenar según la métrica que queramos mostrar gráficamente. Todo esto lo hacemos de la siguiente manera:

```
        .na.fill(0)
        .dropDuplicates(Seq("code"))
        .sort(col("avgPositionsLost").desc)
```

A.2. Programación de queries en PySpark

A.2.1. Mejor piloto de la historia

Para averiguar cuál es el mejor piloto de la historia, debemos fijarnos en varios aspectos del piloto, como sus logros individuales, sin comparar con nadie más a la hora de hacer el cálculo y sus logros relativos a los compañeros de equipo a lo largo de su carrera.

Esta cuestión siempre es motivo de discusión en la comunidad de seguidores de este deporte: cada uno tiene su opinión basada en factores más o menos objetivos. Por esto mismo vamos a tratar de ver cuáles han sido los mejores de todos los tiempos y vamos a intentar establecer una clasificación en base a un sencillo sistema de puntuaciones que vamos a establecer.

Este sistema de puntos va a consistir en lo siguiente: dado el caso de que un piloto sea el mejor en cierta métrica y el tercero en otra, se hará la media de estas dos y obtendrá un total de dos puntos. Como vamos a tener varias métricas, la media se hará de su posición en el *ranking* de cada una de ellas. Finalmente se hará una clasificación nueva en base a esa media. El que un piloto tenga una puntuación menor que otro nos indicará que está por encima de él.

Como es un hecho que no todas las carreras profesionales duran lo mismo, se va a tratar de "relativizar" todo lo posible estas métricas, es decir, se van a obtener métricas como "porcentaje de temporadas con victorias" o "posibilidad de pole". En resumen, la mayoría de métricas van a ser medias o sumatorios divididos por o bien el número de carreras en las que ha participado o el número de temporadas en las que ha competido.

En la descripción detallada de esta query veremos más concretamente qué métricas calculamos y cómo lo hacemos.

Al calcular tantas métricas, queda un código bastante extenso que se proporciona a continuación completo para que el lector se pueda situar a medida que vayamos explicando la query:

```
def averageRank(cols):
        return sum(cols) / len(cols)

averageRank = F.udf(averageRank, T.DoubleType())

def queryAnalisisTemporada(races, drivers, drivers_constr_season, driver_standings, results):

        raceDriverLapWindow = Window.partitionBy("driverId", "raceId").orderBy("lap")
        driverWindow = Window.partitionBy("driverId")
```

```
        seasonWindow = Window.partitionBy("year")
        teammateWindow = Window.partitionBy("year", "constructorId")
        raceConstructorWindow = Window.partitionBy("raceId", "constructorId")
        seasonConstructorWindow = Window.partitionBy("year", "constructorId")
        driverSeasonWindow = Window.partitionBy("driverId", "year")

        races = races\
                .select("raceId", "year")

        drivers = drivers\
                .select(F.col("driverId"),
                        F.concat(F.col("forename"), F.lit(" "),
                            F.col("surname")).alias("fullName"))

        lastRaces = races\
                .withColumn("round", F.col("round").cast(T.IntegerType()))\
                .withColumn("max", F.max(F.col("round")).over(Window.partitionBy("year")))\
                .where(F.col("round") == F.col("max"))\
                .select("raceId", "year")

        driverDomination = driver_standings\
                .join(lastRaces, ["raceId"], "right")\
                .join(driverConstSeasonMap, ["driverId", "year"], "left")\
                .withColumn("teamPointsPerc", F.col("points") /
                    F.sum(F.col("points")).over(teammateWindow))\
                .withColumn("bestOfTeam", F.max(F.col("teamPointsPerc")).over(teammateWindow))\
                .withColumn("dominatedTeammate", F.when(F.col("teamPointsPerc") ==
                    F.col("bestOfTeam"), 1).otherwise(0))\
                .withColumn("dominationPerc",
                    F.round(F.sum(F.col("dominatedTeammate")).over(driverWindow) /
                    F.count(F.col("year")).over(driverWindow) * 100, 2))\
                .select("year", "driverId", "dominatedTeammate", "dominationPerc")

        driverFilter = results\
                .withColumn("finished", F.when(F.col("statusId") == 1, 1).otherwise(0))\
                .withColumn("numberOfFinishes", F.sum(F.col("finished")).over(driverWindow))\
                .where(F.col("numberOfFinishes") < 5)\
                .select("driverId")\
                .distinct()

        teammateComparison = results\
                .join(driverFilter, ["driverId"], "leftanti")\
                .join(races, "raceId")\
                .withColumn("position", F.col("position").cast(T.IntegerType()))\
                .withColumn("grid", F.col("grid").cast(T.IntegerType()))\
                .na.fill({"position" : 100, "grid" : 100})\
                .withColumn("grid", F.when(F.col("grid") == 0, 100).otherwise(F.col("grid")))\
                .withColumn("topPos", F.min(F.col("position")).over(raceConstructorWindow))\
                .withColumn("constructorBestPos", F.when(F.col("topPos") == F.col("position"),
                    1).otherwise(0))\
                .withColumn("topPosPerc",
                    F.sum(F.col("constructorBestPos")).over(driverSeasonWindow) /
F.count(F.col("raceId")).over(driverSeasonWindow) * 100)\
                .withColumn("constTopPosPerc",
                    F.max(F.col("topPosPerc")).over(seasonConstructorWindow))\
                .withColumn("driverDomConstPos", F.when(F.col("constTopPosPerc") ==
                    F.col("topPosPerc"), 1).otherwise(0))\
                .withColumn("topGrid", F.min(F.col("grid")).over(raceConstructorWindow))\
                .withColumn("constructorBestGridPos", F.when(F.col("topGrid") == F.col("grid"),
                    1).otherwise(0))\
                .withColumn("topGridPerc",
                    F.sum(F.col("constructorBestGridPos")).over(driverSeasonWindow) /
F.count(F.col("raceId")).over(driverSeasonWindow) * 100)\
                .withColumn("constTopGridPerc",
                    F.max(F.col("topGridPerc")).over(seasonConstructorWindow))\
                .withColumn("driverDomConstGrid", F.when(F.col("constTopGridPerc") ==
                    F.col("topGridPerc"), 1).otherwise(0))\
```

```
                .dropDuplicates(["driverId", "year"])\
                .withColumn("avgTopPosPerc", F.avg(F.col("topPosPerc")).over(driverWindow))\
                .withColumn("avgTopGridPerc", F.avg(F.col("topGridPerc")).over(driverWindow))\
                .withColumn("avgPosDom", F.avg(F.col("driverDomConstPos")).over(driverWindow))\
                .withColumn("avgGridDom", F.avg(F.col("driverDomConstGrid")).over(driverWindow))\
                .dropDuplicates(["driverId"])\
                .select("driverId", "avgTopPosPerc", "avgTopGridPerc", "avgPosDom", "avgGridDom")

        return results\
                .join(driverFilter, ["driverId"], "leftanti")\
                .join(races, "raceId")\
                .withColumn("grid", F.col("grid").cast(T.IntegerType()))\
                .withColumn("position", F.col("position").cast(T.IntegerType()))\
                .withColumn("firstRowStart", F.when((F.col("grid") == 1) | (F.col("grid") == 2),
                    1).otherwise(0))\
                .withColumn("firstRowChance",
                    F.round(F.sum(F.col("firstRowStart")).over(driverWindow) /
                    F.count(F.col("firstRowStart")).over(driverWindow), 4) * 100)\
                .join(driverDomination, ["driverId", "year"], "left")\
                .join(teammateComparison, ["driverId"], "left")\
                .withColumn("avgGridStart", F.round(F.avg(F.col("grid")).over(driverWindow), 2))\
                .withColumn("avgFinish", F.round(F.avg(F.col("position")).over(driverWindow), 2))\
                .withColumn("pole", F.when(F.col("grid") == 1, 1).otherwise(0))\
                .withColumn("totalPolePositions", F.sum(F.col("pole")).over(driverWindow))\
                .withColumn("poleChance", F.round(F.col("totalPolePositions") /
                    F.count(F.col("pole")).over(driverWindow), 4) * 100)\
                .withColumn("polesPerSeason", F.sum(F.col("pole")).over(driverSeasonWindow))\
                .withColumn("poleChance", F.round(F.col("totalPolePositions") /
                    F.count(F.col("raceId")).over(driverWindow) * 100, 2))\
                .withColumn("hasPoleThisSeason", F.when(F.col("polesPerSeason") > 0,
                    1).otherwise(0))\
                .withColumn("percSeasonsWithPole",
                    F.round(F.sum(F.col("hasPoleThisSeason")).over(driverWindow) /
                    F.count(F.col("year")).over(driverWindow), 4) * 100)\
                .withColumn("win", F.when(F.col("position") == 1, 1).otherwise(0))\
                .withColumn("totalVictories", F.sum(F.col("win")).over(driverWindow))\
                .withColumn("victoryChance", F.round(F.col("totalVictories") /
                    F.count(F.col("win")).over(driverWindow), 4) * 100)\
                .withColumn("winsPerSeason", F.sum(F.col("win")).over(driverSeasonWindow))\
                .withColumn("hasWonThisSeason", F.when(F.col("winsPerSeason") > 0,
                    1).otherwise(0))\
                .withColumn("percSeasonsWithWins",
                    F.round(F.sum(F.col("hasWonThisSeason")).over(driverWindow) /
                    F.count(F.col("year")).over(driverWindow), 4) * 100)\
                .withColumn("podium", F.when((F.col("position") == 1) | (F.col("position") == 2)
                    | (F.col("position") == 3), 1).otherwise(0))\
                .withColumn("podiumChance", F.round(F.sum(F.col("podium")).over(driverWindow) /
                    F.count(F.col("podium")).over(driverWindow), 4) * 100)\
                .dropDuplicates(["driverId"])\
                .select("driverId", "firstRowChance", "avgGridStart", "avgFinish",
"totalPolePositions", "poleChance", "percSeasonsWithPole",
"percSeasonsWithWins", "podiumChance", "dominationPerc",
"avgTopPosPerc", "avgTopGridPerc", "avgPosDom", "avgGridDom")\
                .withColumn("rankFRC",
                    F.rank().over(Window.orderBy(F.col("firstRowChance").desc())))\
                .withColumn("rankAGS",
                    F.rank().over(Window.orderBy(F.col("avgGridStart").asc())))\
                .withColumn("rankAF", F.rank().over(Window.orderBy(F.col("avgFinish").asc())))\
                .withColumn("rankTPP",
                    F.rank().over(Window.orderBy(F.col("totalPolePositions").desc())))\
                .withColumn("rankPSWP",
                    F.rank().over(Window.orderBy(F.col("percSeasonsWithPole").desc())))\
                .withColumn("rankPSWW",
                    F.rank().over(Window.orderBy(F.col("percSeasonsWithWins").desc())))\
                .withColumn("rankPC",
                    F.rank().over(Window.orderBy(F.col("podiumChance").desc())))\
                .withColumn("rankDom",
```

```
            F.rank().over(Window.orderBy(F.col("dominationPerc").desc())))\
        .withColumn("rankPoleC",
            F.rank().over(Window.orderBy(F.col("poleChance").desc())))\
        .withColumn("rankPosPerc",
            F.rank().over(Window.orderBy(F.col("avgTopPosPerc").desc())))\
        .withColumn("rankGridPerc",
            F.rank().over(Window.orderBy(F.col("avgTopGridPerc").desc())))\
        .withColumn("rankPosDom",
            F.rank().over(Window.orderBy(F.col("avgPosDom").desc())))\
        .withColumn("rankGridDom",
            F.rank().over(Window.orderBy(F.col("avgGridDom").desc())))\
        .withColumn("stats", averageRank(
              F.array(F.col("rankFRC"),
                      F.col("rankAGS"),
                      F.col("rankAF"),
                      F.col("rankTPP"),
                      F.col("rankPSWP"),
                      F.col("rankPSWW"),
                      F.col("rankPC"),
                      F.col("rankDom"),
                      F.col("rankPoleC"),
                      F.col("rankPosPerc"),
                      F.col("rankGridPerc"),
                      F.col("rankPosDom"),
                      F.col("rankGridDom")
              )
        ))\
        .withColumn("rank", F.rank().over(Window.orderBy(F.col("stats").asc())))\
        .sort(F.col("rank").asc())\
        .join(driverInfo, "driverId")
```

Descripción detallada

El primer paso para programar esta query es construir varios `DataFrames` auxiliares. Necesitaremos una tabla que relacione la carrera con la temporada en que tuvo lugar, otra que nos relacione el identificador de piloto con su nombre completo, una que nos proporcione información sobre qué carreras fueron las últimas de cada temporada y por último una que establezca una relación entre los pilotos que formaron un equipo en cada temporada.

Para obtener la primera de ellas se usará la tabla `races` y nos quedaremos con las columnas `raceId` y `year`.

```
races = races\
       .select("raceId", "year")
```

Para la segunda usaremos la tabla `drivers` y nos quedaremos con las columnas `driverId` y una concatenación de las columnas `forename` y `surname` que llamaremos `fullName`.

```
driverInfo = drivers\
       .select(F.col("driverId"),
       F.concat(F.col("forename"), F.lit(" "), F.col("surname")).alias("fullName"))
```

También necesitamos obtener la última carrera de cada temporada. Para ello usamos de nuevo la tabla `races` y convertimos la columna `round` a tipo `IntegerType`. Acto seguido, utilizamos una ventana que particione los datos según la columna `year` para obtener el máximo de la `round`. Por último, nos quedamos con las entradas donde `round` sea igual al máximo obtenido. El código es el siguiente:

```
lastRaces = races\
        .withColumn("round", F.col("round").cast(T.IntegerType()))\
        .withColumn("max", F.max(F.col("round")).over(Window.partitionBy("year")))\
        .where(F.col("round") == F.col("max"))\
        .select("raceId", "year")
```

Adicionalmente, en algún momento necesitaremos filtrar pilotos para eliminar outliers. En el caso de esta query, se ha concluido que estos serán aquellos pilotos que no hayan terminado cinco carreras o más. Para ello, usamos la tabla `results` y nos fijamos en la columna `statusId`. Si el valor es `1`, entonces quiere decir que el piloto ha pasado por meta. Para calcular el número de carreras que ha terminado, podemos crear una columna nueva que contenga un `1` si el `statusId` tiene ese valor y `0` en caso contrario.

```
driverFilter = results\
        .withColumn("finished", F.when(F.col("statusId") == 1, 1).otherwise(0))\
```

Acto seguido podemos hacer un sumatorio de esta columna particionando por piloto y después filtrar los que no lleguen a cinco. Al poder tener varias entradas por piloto, usamos la función `distinct()` para deshacernos de estos valores repetidos.

```
        .withColumn("numberOfFinishes", F.sum(F.col("finished")).over(driverWindow))\
        .where(F.col("numberOfFinishes") < 5)\
        .select("driverId")\
        .distinct()
```

Por último, como ya comentamos cómo obtener una relación entre pilotos, fabricante y temporada en la sección 3.1.1, no entraremos al detalle de cómo se obtiene y simplemente la usaremos según nos viene dada en los argumentos de la función.

También necesitaremos las siguientes ventanas de datos auxiliares:

```
raceDriverLapWindow = Window.partitionBy("driverId", "raceId").orderBy("lap")
driverWindow = Window.partitionBy("driverId")
seasonWindow = Window.partitionBy("year")
teammateWindow = Window.partitionBy("year", "constructorId")
raceConstructorWindow = Window.partitionBy("raceId", "constructorId")
seasonConstructorWindow = Window.partitionBy("year", "constructorId")
driverSeasonWindow = Window.partitionBy("driverId", "year")
```

A continuación podemos pasar a la primera query, con la que trataremos de averiguar el porcentaje de temporadas en las que, a lo largo de su carrera, un ha terminado por delante de su compañero de equipo.

Para ello, primero usaremos la tabla `driver_standings`, que interseccionaremos con `lastRaces` para quedarnos con la clasificación en la última carrera de cada temporada y con `drivers_constr_season` para completar con la información que nos proporciona dicha tabla.

```
driverDomination = driver_standings\
        .join(lastRaces, ["raceId"], "right")\
        .join(drivers_constr_season, ["driverId", "year"], "left")\
```

El siguiente paso es averiguar el porcentaje de puntos que cada piloto ha obtenido para el equipo en cada temporada. Para ello dividiremos los puntos obtenidos por un piloto entre el total del equipo, calculado haciendo un sumatorio de los puntos según la ventana `teammateWindow`.

Para ver si un piloto ha dominado a su compañero de equipo en una temporada en concreto, podemos usar este último valor calculado y averiguar el máximo entre los integrantes del equipo. Cuando en una entrada de la tabla tengamos que un piloto iguala en puntos a este máximo, sabremos que ha sido el dominante en esa temporada.

```
        .withColumn("teamPointsPerc", F.col("points") /
            F.sum(F.col("points")).over(teammateWindow))\
        .withColumn("bestOfTeam", F.max(F.col("teamPointsPerc")).over(teammateWindow))\
        .withColumn("dominatedTeammate", F.when(F.col("teamPointsPerc") == F.col("bestOfTeam"),
            1).otherwise(0))\
```

Con este último cálculo podemos ver cuántas temporadas ha dominado cada piloto, pero es más útil para nosotros relativizarlo a la carrera de cada piloto. Por ejemplo, si un piloto compite cuatro años y domina tres de ellos, su total de temporadas dominadas será tres, mientras que el porcentaje de su carrera en la que ha dominado a su compañero de equipo es del 75 %. Esto nos es útil ya que el número de temporadas en las que un piloto compite en el deporte puede variar drásticamente, desde una sola temporada a veinte por poner alguna cifra.

Con esto en mente, podemos contar el número de temporadas en las que ha dominado y dividirlo entre el número de temporadas en las que ha competido, utilizando para ambos parámetros una ventana de datos que particione por piloto

```
        .withColumn("dominationPerc",
            F.round(F.sum(F.col("dominatedTeammate")).over(driverWindow) /
            F.count(F.col("year")).over(driverWindow) * 100, 2))\
```

Calculada esta métrica, podemos pasar a calcular el resto. Para ello primero usamos la tabla `results` y filtramos los outliers intersecamos con la tabla `driverFilter`. Para llevar a cabo este filtrado, debemos hacer la intersección de tipo `leftanti`. También interseccionamos con la tabla `races` para obtener información sobre la temporada en la que se llevó a cabo cada carrera.

Como las columnas `position` y `grid` son de tipo `String`, debemos cambiarlas a tipo entero y tratar los nulos en ambos casos. Esto último es importante, ya que al ordenar estas columnas los nulos quedan por encima del resto. Estos nulos aparecen debido a que no todas las entradas de esta columna contienen números. En el caso de que un piloto no haya terminado la carrera aparecerá un "`\N`".

Para encargarnos de estos valores nulos, se puede usar la función `na.fill()`, que en este caso recibirá un diccionario cuya clave será un `String` que denote el nombre de la columna y el valor será el que queremos dar en caso de encontrar un nulo. Para este caso se ha decidido que los nulos se sustituyan por el valor `100`, ya que en ambos casos nunca ha habido una carrera con ese número de competidores, asegurando así que siempre aparecerá el último si quisiéramos ordenar de menor a mayor.

```
teammateComparison = results\
        .join(driverFilter, ["driverId"], "leftanti")\
        .join(races, "raceId")\
        .withColumn("position", F.col("position").cast(T.IntegerType()))\
        .withColumn("grid", F.col("grid").cast(T.IntegerType()))\
        .na.fill({"position" : 100, "grid" : 100})\
```

Habiendo terminado con la preparación del `DataFrame`, podemos calcular las siguientes métricas: el porcentaje de carreras en las que un piloto ha acabado mejor que su compañero en la parrilla de salida y al acabar y el porcentaje total de temporadas en las que ha quedado

mejor que el compañero de equipo igual, tanto en la parrilla de salida como al acabar. Para ello lo primero será averiguar cuándo un piloto ha acabado por delante de su compañero de equipo en una carrera. Esto lo conseguiremos sacando el mínimo valor de la columna `position` particionando sobre fabricante y carrera y comparando con la posición en la que ha terminado cada piloto.

```
    .withColumn("topPos", F.min(F.col("position")).over(raceConstructorWindow))\
    .withColumn("constructorBestPos", F.when(F.col("topPos") == F.col("position"),
        1).otherwise(0))\
```

Tras esto podemos calcular el porcentaje de carreras en las que cada piloto ha estado por delante de su compañero dividiendo la suma de la última columna entre el conteo de carreras en la temporada del piloto.

```
    .withColumn("topPosPerc", F.sum(F.col("constructorBestPos")).over(driverSeasonWindow) /
    F.count(F.col("raceId")).over(driverSeasonWindow) * 100)\
```

El siguiente paso será cuantificar este dominio calculando el porcentaje de carreras de la temporada en la que ha quedado por delante del resto del equipo. Para ello primero tenemos que averiguar carrera a carrera dentro del equipo quien tiene el `topPosPerc` más alto, y "marcar" cuando nuestro `topPosPerc` sea el más alto para luego ver el porcentaje de temporadas que ha dominado.

```
    .withColumn("constTopPosPerc", F.max(F.col("topPosPerc")).over(seasonConstructorWindow))\
    .withColumn("driverDomConstPos", F.when(F.col("constTopPosPerc") == F.col("topPosPerc"),
        1).otherwise(0))\
```

Con una serie de operaciones muy similares, podemos calcular lo mismo pero en lugar de tener en cuenta la posición en la que termina la carrera, la posición en la parrilla de salida:

```
    .withColumn("topGrid", F.min(F.col("grid")).over(raceConstructorWindow))\
    .withColumn("constructorBestGridPos", F.when(F.col("topGrid") == F.col("grid"),
        1).otherwise(0))\
    .withColumn("topGridPerc",
        F.sum(F.col("constructorBestGridPos")).over(driverSeasonWindow) /
    F.count(F.col("raceId")).over(driverSeasonWindow) * 100)\
    .withColumn("constTopGridPerc",
        F.max(F.col("topGridPerc")).over(seasonConstructorWindow))\
    .withColumn("driverDomConstGrid", F.when(F.col("constTopGridPerc") ==
        F.col("topGridPerc"), 1).otherwise(0))\
```

Como queremos quedarnos con una entrada por piloto y temporada, aplicamos un `dropDuplicates` a las columnas pertinentes.

```
    .dropDuplicates(["driverId", "year"])\
```

Para calcular las métricas finales, haremos la media de las columnas `avgTopDom` y `avgDom`. La media de la primera columna nos dará el porcentaje de veces que ha quedado por delante de un compañero a lo largo de la carrera, tanto para la posición de salida como la de finalizado y la media de la segunda nos dará el porcentaje de temporadas en las que ha dominado a su compañero de equipo.

```
    .withColumn("avgTopPosPerc", F.avg(F.col("topPosPerc")).over(driverWindow))\
    .withColumn("avgTopGridPerc", F.avg(F.col("topGridPerc")).over(driverWindow))\
    .withColumn("avgPosDom", F.avg(F.col("driverDomConstPos")).over(driverWindow))\
    .withColumn("avgGridDom", F.avg(F.col("driverDomConstGrid")).over(driverWindow))\
```

Por último, como solo queremos una entrada por piloto, hacemos otro `dropDuplicates` y seleccionamos las columnas que queremos que queden en el DataFrame.

```
        .dropDuplicates(["driverId"])\
        .select("driverId", "avgTopPosPerc", "avgTopGridPerc", "avgPosDom", "avgGridDom")
```

Sin embargo, estas no son las únicas métricas que queremos calcular. Estamos interesados también en las siguientes:

- Posibilidad de que empiece la carrera en primera fila.
- Posición media de salida.
- Posición media al acabar la carrera.
- Total de poles.
- Porcentaje de poles relativo al número de carreras en las que se ha participado.
- Porcentaje de temporadas en las que ha conseguido una pole.
- Porcentaje de temporadas en las que ha ganado una carrera.
- Porcentaje de podios relativo al número de carreras en las que se ha participado.

Primero, como viene siendo habitual, preparamos el `DataFrame` para los cálculos. Usamos la tabla `results` e interseccionamos con los `DataFrames` `driverDomination` según las columnas `driverId` y `year` y `teammateComparison` según la columna `driverId`. En ambos casos hacemos una intersección de tipo left. También interseccionamos con `races` para añadir información sobre la temporada en la que se ha disputado cada carrera.

Por último hacemos la intersección con `driverFilter` tomando como referencia la columna `driverId`. Sin embargo, esta intersección se hará de tipo "leftanti". Esto quiere decir que el DataFrame resultante contendrá todos los elementos que no estén en el que queremos intersecccionar por la derecha.

Esta es una manera eficiente de hacer un filtro en Spark. Otra manera sería hacer un `collect()` de `driverFilter` y obtenerlo como una lista. Llegados a este punto, en lugar de hacer una intersección "leftanti", haríamos algo como:

```
        .where(~F.col('driverId').isin(lista))
```

Sin embargo, hacer un `collect()` en Spark resulta muy costoso y por ello se ha optado por la primera opción.

De nuevo tendremos que convertir a tipo entero las columnas `grid` y `position`. El código para preparar este `DataFrame` es el siguiente:

```
results = results\
        .join(driverFilter, ["driverId"], "leftanti")\
        .join(races, "raceId")\
        .join(driverDomination, ["driverId", "year"], "left")\
        .join(teammateComparison, ["driverId"], "left")\
        .withColumn("grid", F.col("grid").cast(T.IntegerType()))\
        .withColumn("position", F.col("position").cast(T.IntegerType()))\
```

Lo primero que podemos calcular es el porcentaje de salidas en primera fila. Para ello primero tendremos que definir una columna que valga `1` cuando estemos en primera o segunda posición y `0` en otro caso. Para calcular la métrica podemos hacer sumar la columna recién definida particionando por piloto.

```
        .withColumn("firstRowStart", F.when((F.col("grid") == 1) | (F.col("grid") == 2),
            1).otherwise(0))\
        .withColumn("firstRowChance", F.round(F.sum(F.col("firstRowStart")).over(driverWindow) /
            F.count(F.col("firstRowStart")).over(driverWindow), 4) * 100)\
```

Lo siguiente que podemos calcular es la posición de salida y la posición al acabar media. Esto se puede hacer con las siguientes líneas de código:

```
        .withColumn("avgGridStart", F.round(F.avg(F.col("grid")).over(driverWindow), 2))\
        .withColumn("avgFinish", F.round(F.avg(F.col("position")).over(driverWindow), 2))\
```

También podemos calcular el total de poles, el porcentaje de poles respecto a todas las carreras disputadas y el porcentaje de temporadas en las que ha conseguido una pole.

Para ello primero tenemos que diferenciar las poles, esto es, cuando el piloto sale primero al iniciar la carrera. Después podemos sumar todas las poles de forma global para la primera métrica y particionando por temporada también para continuar calculando las siguientes.

El porcentaje de poles respecto al número de carreras podemos obtenerlo dividiendo el total con el conteo de carreras en las que ha participado el piloto. Para calcular el porcentaje de temporadas con pole, primero debemos de nuevo definir en qué temporadas ha conseguido alguna y hacer una cuenta similar a la anterior. Sumando todas las temporadas en las que se ha conseguido y dividiendo entre el total de campeonatos disputados. El código es el siguiente:

```
        .withColumn("pole", F.when(F.col("grid") == 1, 1).otherwise(0))\
        .withColumn("totalPolePositions", F.sum(F.col("pole")).over(driverWindow))\
        .withColumn("polesPerSeason", F.sum(F.col("pole")).over(driverSeasonWindow))\
        .withColumn("poleChance", F.round(F.col("totalPolePositions") /
            F.count(F.col("raceId")).over(driverWindow) * 100, 2))\
        .withColumn("hasPoleThisSeason", F.when(F.col("polesPerSeason") > 0, 1).otherwise(0))\
        .withColumn("percSeasonsWithPole",
            F.round(F.sum(F.col("hasPoleThisSeason")).over(driverWindow) /
            F.count(F.col("year")).over(driverWindow), 4) * 100)\
```

Querríamos también hallar estas mismas métricas, pero para victorias. Esto resulta muy similar, salvo que en lugar de usar la columna `grid`, se usa la columna `position`:

```
        .withColumn("win", F.when(F.col("position") == 1, 1).otherwise(0))\
        .withColumn("totalVictories", F.sum(F.col("win")).over(driverWindow))\
        .withColumn("victoryChance", F.round(F.col("totalVictories") /
            F.count(F.col("win")).over(driverWindow), 4) * 100)\
        .withColumn("winsPerSeason", F.sum(F.col("win")).over(driverSeasonWindow))\
        .withColumn("hasWonThisSeason", F.when(F.col("winsPerSeason") > 0, 1).otherwise(0))\
        .withColumn("percSeasonsWithWins",
            F.round(F.sum(F.col("hasWonThisSeason")).over(driverWindow) /
            F.count(F.col("year")).over(driverWindow), 4) * 100)\
```

También nos interesaría calcular el porcentaje de carreras en las que se ha obtenido un podio. Para ello primero definimos cuándo se ha conseguido un podio, es decir, cuando se ha terminado primero, segundo o tercero. Después y de forma muy parecida a los cálculos anteriores, podemos hacer una suma de todos los podios y dividir por el número de carreras disputadas.

```
        .withColumn("podium", F.when((F.col("position") == 1) | (F.col("position") == 2) |
            (F.col("position") == 3), 1).otherwise(0))\
        .withColumn("podiumChance", F.round(F.sum(F.col("podium")).over(driverWindow) /
            F.count(F.col("podium")).over(driverWindow), 4) * 100)\
```

Por último, eliminamos la duplicidad según la columna `driverId` y seleccionamos las columnas que queremos que continúen.

```
        .dropDuplicates(["driverId"])\
        .select("driverId", "firstRowChance", "avgGridStart", "avgFinish",
        "totalPolePositions", "poleChance", "percSeasonsWithPole",
        "percSeasonsWithWins", "podiumChance", "dominationPerc",
        "avgTopPosPerc", "avgTopGridPerc", "avgPosDom", "avgGridDom")
```

Para finalizar esta query, crearemos una clasificación para cada métrica que hemos calculado. Algunas de menor a mayor y otras al contrario, y aplicaremos la UDF que programamos en la query anterior para hacer la media estas clasificaciones.

```
return results\
        .withColumn("rankFRC", F.rank().over(Window.orderBy(F.col("firstRowChance").desc())))\
        ...
        .withColumn("rankGridDom", F.rank().over(Window.orderBy(F.col("avgGridDom").desc())))\
        .withColumn("stats", averageRank(
                F.array(F.col("rankFRC"),
                        F.col("rankAGS"),
                        ...
                        F.col("rankGridDom")
                )
        ))\
```

Además, creamos una columna que clasifique la columna `stats` de menor a mayor y ordenamos el `DataFrame` y ya por último completamos la información del piloto, en este caso nombre y apellido, interseccionando con el `DataFrame driverInfo`.

B

Migración de queries

B.1. Expresiones regulares

Sustituciones Directas

Estas sustituciones son aquellas no requieren del uso de expresiones regulares complejas. Un ejemplo de sustitución de este tipo serían los operadores `===` y `=!=`. Simplemente podríamos sustituirlos por su variante de Python comentada anteriormente. Otros ejemplos incluyen:

- Comentarios de una sola línea: de `//` a `#`.
- Valores nulos: `null` a `None`.
- Valores booleanos: `true` a `True` y `false` a `False`.
- Descriptores de orden para ventanas: `desc` a `desc()` y `asc` a `asc()`
- La función `isInCollection` pasa a `isin`

Estandarización de las invocaciones a `col`

Una vez hemos terminado con estas sustituciones sencillas, podemos empezar a aprovechar la potencia de las expresiones regulares. Una de las consecuencias de utilizar estas herramientas será que acabaremos con un código estandarizado. Por ejemplo, en Scala Spark es posible invocar una columna de las siguientes maneras si importamos los implícitos de Spark:

```
import spark.implicits._

col("colName")
$"colName"
'colName
```

Mientras que en Python podemos hacerlo de las siguientes formas:

```
import pyspark.sql.functions as F
```

```
F.col("colName")
dataframe.colName
```

Como para ambas API podemos usar la función `col`, entonces nos interesaría transformar las otras dos maneras de invocar columnas a esta. Para ello podemos usar las siguientes expresiones regulares:

- Con `[$]([^ ]*)` detectamos el primer caso, y lo sustituimos por `col(\1)`
- Con `[\']([a-zA-Z0-9]*))` detectamos el segundo y sustituimos por `col(\"\1\")`

Corregir las invocaciones a los tipos de Spark

Lo siguiente será añadir el prefijo `T.` y los paréntesis a los tipos de Spark. En esta ocasión no fui capaz de listarlos como en el caso anterior y se hizo a mano, pero la idea sigue siendo colocar un operador OR entre cada elemento posible, todo rodeado por paréntesis.

Esta expresión regular luego la sustituiríamos por `T.\1()`.

Eliminación de `val` y `var`

Lo siguiente será eliminar los `var` y `val` de las declaraciones, lo cual podemos conseguir mediante la siguiente expresión regular:

```
(val|var)[ ]+
```

Todo lo que detecte deberá ser sustituido por la cadena vacía, esencialmente eliminándolos. Cabe destacar que en el caso de que tengamos funciones definidas como `val`, se necesitará intervención manual como comentaremos más adelante.

Traducción de listas, secuencias y arrays

Lo siguiente que querríamos cambiar son las listas, secuencias y arrays por arrays de Python. Estas se usan por ejemplo en las intersecciones para denominar las columnas que queremos que coincidan, o para establecer filtros que luego usamos en `isInCollection`, o `isin` en PySpark.

Esta expresión regular es la más larga de todas, ya que debemos distinguir entre listas de `String` y listas de otro tipo de datos. Esto es así ya que el delimitador de fin de lista es el paréntesis cerrado, indicando que ya no va a haber ningún elemento más, pero es posible que en una lista de `String` uno de los elementos contenga este carácter.

La expresión regular es la siguiente:

```
((\bList\b)|(\bSeq\b)|(\bArray\b))\((([ |\n]*"[^\n"]*([ |\n]*,[ |\n]*"[^\n"]*"[ |\n]*)*)|
       ([ |\n]*[^\n"\)]*([ |\n]*,[ |\n]*"[^\n"]*"[ |\n]*)*))\)
```

Y la podemos sustituir por `[\5]`

Cabe destacar que tanto esta como la siguiente expresión no la podemos utilizar en `sed`, debido a que es posible encontrarnos con que tanto mapas como listas estén definidas en varias líneas y `sed` no es capaz de detectar el carácter de retorno de carro. Es por esto que sebe utilizar `perl` como para aplicar nuestras expresiones regulares. Lo haríamos de la siguiente manera:

```
perl -007 -pe 's/<regex>/<sustitución>/sg' <archivo de entrada>
```

Entramos más en detalle sobre Perl y el formato del mandato en el apéndice B.2.

Perl nos permite usar expresiones regulares no codiciosas, de forma que en el caso de querer detectar únicamente hasta la primera aparición de cierto carácter, sería posible. Esto nos será útil en la siguiente tarea.

Traducción de Mapas

De forma similar, debemos cambiar los `Map` a diccionarios. Para ello buscamos detectar los contenidos del mapa como tales y envolverlos entre llaves. Esto lo podemos hacer con la siguiente expresión:

```
Map\((.*?)\)
```

Y lo que detecte lo sustituimos por {`$1`}.

Después sustituimos los caracteres `->` por `:` y, por último, tenemos que tener en cuenta que hay algunos mapas que tienen valores por defecto en caso de que no se encuentre la clave. Estos se distinguen añadiendo `.withDefaultValue()` y pasando como argumento el valor por defecto. En Python sin embargo, este tipo de diccionarios se crean de forma distinta, usando `defaultdict`.

Para cambiar un valor por defecto por otro tendremos que detectar cuál es ese valor y hacer el cambio a la manera de implementarlo con Python. Primero detectamos este caso por defecto con:

```
({.*?}).withDefaultValue\((.*?)\)
```

Y sustituimos lo que detecte por:

```
defaultdict(lambda:$2,$1)
```

Como hemos visto, se ha utilizado el carácter `?` dentro de las expresiones regulares, que viene a significar "hasta la siguiente aparición del carácter que defino a continuación", seguido del carácter límite. Este nos permite evaluar de forma no codiciosa, quedándonos en la primera aparición del carácter que deseemos que se encuentre a continuación.

Otros ajustes menores

El siguiente paso es añadir un paréntesis después de `isNull` e `isNotNull`, lo cual haremos primero detectando estos casos con:

```
((\bisNull\b)|(\bisNotNull\b))([\)| |,|\n])
```

Y que sustituiremos por `\1()\4`

A la hora de hacer una intersección, es común encontrar una expresión como:

```
col("colName") === otherDf("otherCol")
```

La forma para denominar una columna de un `DataFrame` que se observa en la segunda parte de la comparación no está permitida en Python, y por tanto debemos cambiarla a otra como:

```
otherDf.otherCol
```

Como este tipo de denominación se suele dar en intersecciones a la hora de establecer la

comparación, podemos usar una expresión regular como la siguiente para detectarlos:

```
(.join\(.*== )((.*)\("(.*)"\)),
```

Que sustituiremos por `\1\3.\4,`.

Añadir paréntesis a condiciones

En Scala Spark es posible ejecutar el siguiente código sin problemas:

```
dataframe
       .when(col("a") === "foo" && col("b") =!= "bar", ...)
```

Sin embargo, en Python tenemos que separar cada condición con paréntesis, ya que el intérprete tratará de hacer la operación "bitwise" `&` entre `"foo"` y `col("b")`, lo cual resultará en que tengamos que insertar cada condición entre paréntesis:

```
dataframe
       .when((col("a") === "foo") & (col("b") =!= "bar"), ...)
```

Al ser algo repetitivo, lo ideal sería poder diseñar una expresión regular que se encargase de ello, pero en este caso debemos hacerlo en dos partes si usamos `sed`: sustituir los operadores lógicos por su homólogo en Python de forma que tengamos ya colocados los paréntesis interiores y añadir los paréntesis exteriores.

Lo primero lo podemos hacer con las siguientes expresiones regulares:

```
\&\&
\|\|
```

Que sustituiremos por:

```
) \& (
) \| (
```

Por último, debemos añadir los paréntesis exteriores. Como podemos encontrarnos condiciones tanto en `where` como en `when`, debemos distinguir entre los dos casos ya que para cada uno el delimitador exterior es distinto. En el primero es el cierre de paréntesis y en el segundo la coma. Aún con eso, las expresiones quedan muy similares como se puede ver a continuación:

```
(\bwhere\b)\( *(.*(\&|\|).*) *[\)\\]
(\bwhen\b)\( *(.*(\&|\|).*) *,
```

Y sustituiremos, respectivamente, según los patrones:

```
\1((\2)\\
\1((\2),
```

Donde podemos apreciar con más claridad que el único cambio es el delimitador de fin de condición.

B.2. Desarrollo del *script*

En este apartado vamos a entrar más en el detalle de cómo ha sido la implementación del *script* de automatización de la migración y aspectos que se han tenido en cuenta a la hora de desarrollarlo.

Como ya hemos comentado, gran parte de esta automatización se hace a través del comando `sed`. Este nos permite aplicar expresiones regulares y hacer sustituciones con los patrones detectados. Utiliza el estándar POSIX ERE y presenta algunas limitaciones que tendremos que sortear, como la incapacidad de detectar el carácter de retorno de carro.

La idea inicial para automatizar el proceso consistía en concatenar mandatos que aplicasen cada uno una expresión regular, pero indagando en el manual de `sed`, con `info sed` y `man sed` se vio la posibilidad de aplicar todas nuestras expresiones de golpe mediante el uso de un *script* de `sed`.

Este nos permite crear un archivo de texto en el que colocaremos todas las instrucciones que queremos que se lleven a cabo sobre nuestro *stream* de texto o archivo en orden secuencial. También permite insertar comentarios y líneas en blanco, que son muy útiles para facilitar la lectura y que el usuario pueda comprender qué está pasando.

El orden en el que se aplican las instrucciones depende de las dependencias que existan entre ellas. Por ejemplo, no podemos primero añadir el `F.` y luego estandarizar las llamadas a las columnas. Por norma general existen pocas dependencias de este tipo, pero las que existen se han tenido en cuenta a la hora de establecer el orden de aplicación. El archivo completo se puede consultar en el repositorio de Github asociado al proyecto. En el apéndice D podemos encontrar un enlace directo al repositorio.

Otro aspecto a tener en cuenta es la aplicación de algunas expresiones regulares con Perl. Esto se ha hecho debido a que el estándar PCRE2 nos proporciona la posibilidad de evaluar de forma no codiciosa, es decir, nos permite quedarnos con lo que deseemos hasta la primera aparición de cierto carácter y debido también a que podemos distinguir el carácter de retorno de carro. Esto ha sido esencial a la hora de desarrollar las expresiones para sustituir listas y mapas, ya que normalmente están definidas en varias líneas para mejorar la legibilidad.

Un ejemplo de cómo se haría una sustitución con Perl sería el siguiente:

```
perl -0777 -pe 's/Map\((.*?)\)/{$1}/sg' filename
```

El manual de usuario de Perl nos dice que `-0777` hacer que Perl funcione en modo "*slurp*", lo cual quiere decir que traerá todo el archivo leído a memoria para aplicar las operaciones que deseemos. Las opciones `-pe` nos indica que coge un archivo y crea otro como la salida de lo que queramos aplicar. El resto se parece mucho a lo que hemos hecho ya con `sed`: primero le decimos que vamos a sustituir con `s/`, luego va la expresión regular, seguido de `/` y la expresión de sustitución y al final los modos en los que se va a aplicar, en este caso de forma global como viene siendo habitual. Por último tendremos el archivo del que leeremos los datos de entrada. Los datos de salida los redirigiremos o bien a otra operación de Perl, `sed` o al archivo de salida.

Habiendo visto esto, podemos explicar el código del *script* de *shell* desarrollado que se encarga de todo esto. Es el siguiente:

```
#!/bin/sh
filename="${1%.*}"

perl -0777 -pe 's/Map\((.*?)\)/{$1}/sg' "$1" | \
perl -0777 -pe 's/({.*?}).withDefaultValue\((.*?)\)/defaultdict(lambda: $2, $1)/sg' | \
perl -0777 -pe 's/((\bList\b)|(\bSeq\b)|(\bArray\b))\((([ |\n]*"[^\n"]*"([ |\n]*,[ |\n]*"[^\n"]*"[ |\n]*)*)|([ |\n]*[^\n"\)]*([ |\n]*,[ |\n]*"[^\n"]*"[ |\n]*)*))\)/[$5]/sg' | \
sed -E -f scriptfile.sed > "$filename".py;
printf "\n\n" >> "$filename".py
```

Figura B.1: Código del script de automatización

Aquí podemos ver que lo primero que se hace es leer el nombre del archivo de entrada, dejando de lado la extensión. Después pasa ya a aplicar las expresiones regulares que se encargan de traducir listas y mapas a sus contrapartes en Python para pasar a aplicar todas las instrucciones que hemos definido en el *script* de `sed` y escribir el resultado en un archivo con el mismo nombre de entrada pero extensión `.py`. Como último paso se añade una línea vacía al final para evitar problemas al ejecutar como ya se comentó en el apartado correspondiente de la memoria.

Un aspecto importante a destacar de este *script* es que es "POSIX compliant", es decir, que cumple con todos los estándares de POSIX y se debe de poder ejecutar en cualquier *shell* y sistema, ya sea OSX, BSD o Linux. Esto resulta interesante porque no limita al usuario a un sistema o *shell* en concreto, como sí habría pasado si se hubiera desarrollado con "*bashismos*".

Una herramienta clave a la hora de garantizar esta portabilidad ha sido `shellcheck`, que dispone tanto de versión de línea de comandos como web[1]. Analiza el código y señala errores, y, en el caso de este *script*, fue capaz de aplicar los ajustes necesarios para que garantizar lo que buscábamos.

Además, como medida extra para asegurar la portabilidad del *script*, se ha ejecutado en distintas *shell*. En concreto se ha ejecutado en `bash`, `zsh` y, lo más importante, `posh`. `posh` es una *shell* que se ajusta a las políticas POSIX. En el caso de que no consiguiésemos ejecutar nuestro *script* en `posh`, tendríamos que revisar si existe algún error en la forma en la que está programado.

[1] `https://www.shellcheck.net/`

C

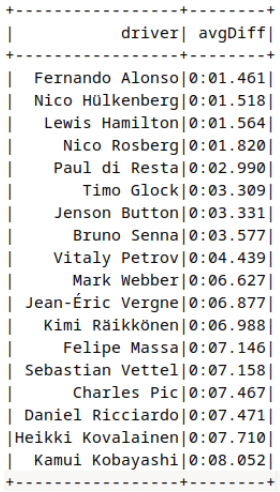
Resultados obtenidos

C.1. Piloto más consistente en la temporada 2012

Con esta query el objetivo era averiguar qué piloto ha sido el más consistente en un periodo de tiempo determinado, en este caso la temporada 2012. Como ya explicamos anteriormente, hemos definido "consistencia" como la diferencia entre la media de las vueltas rápidas del piloto y la media de todas las vueltas dadas en todos los grandes premios de ese periodo.

En este caso, el resultado final de la query nos proporciona directamente la respuesta que buscamos. Ordena según el parámetro de consistencia calculado de mayor a menor, y lo acompaña con el nombre y el apellido del piloto en cuestión.

A continuación se puede ver el resultado de la query para la temporada 2012:



driver	avgDiff
Fernando Alonso	0:01.461
Nico Hülkenberg	0:01.518
Lewis Hamilton	0:01.564
Nico Rosberg	0:01.820
Paul di Resta	0:02.990
Timo Glock	0:03.309
Jenson Button	0:03.331
Bruno Senna	0:03.577
Vitaly Petrov	0:04.439
Mark Webber	0:06.627
Jean-Éric Vergne	0:06.877
Kimi Räikkönen	0:06.988
Felipe Massa	0:07.146
Sebastian Vettel	0:07.158
Charles Pic	0:07.467
Daniel Ricciardo	0:07.471
Heikki Kovalainen	0:07.710
Kamui Kobayashi	0:08.052

Figura C.1: Correspondencia entre nombre del piloto y su código

Como se puede observar, el piloto más consistente es Fernando Alonso, con una diferencia entre las vueltas rápidas y todas las vueltas de la temporada de 1.461 segundos. Podemos ver también que el campeón de ese año, Sebastian Vettel, queda en la parte baja de esta clasificación, con una diferencia al primero de casi 6 segundos.

C.2. Mejor piloto de la historia

El objetivo de esta query era claro: intentar averiguar cuál ha sido el mejor piloto de la historia en base a varias métricas, que principalmente se dividen en dos grupos: comparativas con compañeros de equipo y estadísticas individuales. Por norma general se ha tratado de usar estadísticas relativas, es decir, en lugar de hacer un conteo de victorias, se ha hecho el conteo dividido por el número de carreras que en las que ha competido.

Como métricas individuales del piloto tenemos:

- Posibilidad de empezar la carrera en primera fila (primero o segundo).
- Media de posición en parrilla.
- Media de posición final.
- Posibilidad de hacer pole (clasificar primero).
- Porcentaje de temporadas con pole.
- Porcentaje de temporadas con al menos una victoria.
- Posibilidad de terminar la carrera en el podio.
- Total de poles.

Cabe destacar que se ha tratado de encontrar métricas que sean imparciales. Por ejemplo, es probable que el piloto empiece su carrera en un equipo pequeño con un coche peor y que más adelante de el salto a otro más importante y con posibilidades de victorias y poles. Sin embargo, en mi opinión, un gran piloto debe poder llevar a cabo ciertos hitos en coches peores. Como ejemplos podríamos poner a Ayrton Senna en Mónaco 1984 o Fernando Alonso durante toda la temporada 2012. En ambos casos, la habilidad del piloto hizo posible conseguir resultados que de otra manera no hubiesen sido posibles. En resumen, si un piloto puede conseguir buenos resultados con coches no competitivos, entonces debe ser beneficiado.

También se ha buscado cierta justicia hacia el piloto, es decir, probablemente no siempre el mejor quede o clasifique primero. A veces la diferencia entre el primero y el segundo es mínima, y pienso que esto no debe penalizar. Por ello se han usado métricas como la posibilidad de acabar en podio o la de empezar en primera fila.

Como métricas que comparan pilotos del mismo equipo tenemos:

- Porcentaje de temporadas de su carrera en las que un piloto ha dominado a su compañero de equipo en puntos.
- Porcentaje de carreras en las que un piloto ha terminado por delante de su compañero de equipo.
- Porcentaje de carreras en las que un piloto ha clasificado por delante de su compañero de equipo.
- Porcentaje de temporadas en las que un piloto ha superado a su compañero de equipo en clasificación.
- Porcentaje de temporadas en las que un piloto ha superado a su compañero de equipo al acabar la carrera.

Comparar pilotos de distintas épocas es complejo ya que el deporte en sí ha variado enormemente. Una de las mejores maneras de determinar si un piloto es bueno es comparar con su compañero de equipo en una determinada temporada, ya que en principio deberían tener el mismo coche. Si uno de ellos ha quedado siempre por delante del otro en clasificación, podemos asegurar que es más rápido.

Para obtener los resultados de esta query en forma de clasificación sencilla en la que se ordene a los pilotos de mejor a peor debemos buscar una forma de "unir" todas estas métricas para formar una sola. En este caso esto lo hemos hecho obteniendo un ranking de cada métrica y haciendo la media de las posiciones que ocupa cada piloto en los rankings de las métricas. Por ejemplo, si un piloto está en la primera posición en una métrica y tercero en otra, se obtiene como valor total la media de ambos, en este caso 1,5.

Si se aplica esto podemos obtener un resultado como el siguiente:

stats	rank	fullName
4.923076923076923	1	Juan Fangio
5.384615384615385	2	Ayrton Senna
8.923076923076923	3	Jim Clark
11.615384615384615	4	Lewis Hamilton
12.692307692307692	5	Alain Prost
13.615384615384615	6	Michael Schumacher
15.153846153846153	7	Jackie Stewart
21.0	8	Nelson Piquet
21.307692307692307	9	Max Verstappen
22.923076923076923	10	Sebastian Vettel
23.076923076923077	11	James Hunt
24.384615384615383	12	Mika Häkkinen
26.923076923076923	13	Alberto Ascari
27.53846153846154	14	Damon Hill
29.46153846153846	15	Fernando Alonso
30.076923076923077	16	Stirling Moss
35.07692307692308	17	Nico Rosberg
37.92307692307692	18	Charles Leclerc
38.38461538461539	19	Emerson Fittipaldi
38.53846153846154	20	John Surtees
39.92307692307692	21	Jochen Rindt
41.76923076923077	22	René Arnoux
43.15384615384615	23	José Froilán Gonz...
44.53846153846154	24	Alan Jones
45.07692307692308	25	Tony Brooks
45.30769230769231	26	Juan Pablo Montoya
46.84615384615385	27	Mark Webber
46.92307692307692	28	Jacky Ickx
47.61538461538461	29	Dan Gurney
47.92307692307692	30	Gerhard Berger

Figura C.2: Ranking de mejores pilotos de la historia

Para contrastar el resultado podemos ver que todos estos pilotos de una manera u otra han pasado a la historia. En concreto las 6 primeras entradas independientemente del orden que cada uno pueda dar son considerados por muchos los mejores de todos los tiempos.

Además del resultado final podemos fijarnos también en los resultados de cada métrica, ya que en el `DataFrame` final se muestran tanto los resultados de cada una de ellas como el ranking de todas y el final. Por ilustrar esto, veamos el ranking de pilotos que han dominado a sus compañeros de equipo durante más temporadas a lo largo de su carrera:

dominationPerc	fullName
100.0	Felipe Nasr
88.89	Fernando Alonso
87.5	James Hunt
86.67	Lewis Hamilton
83.33	Emerson Fittipaldi
81.82	Ayrton Senna
80.0	Jackie Stewart
77.78	Juan Fangio
77.78	Mika Salo
76.92	Alain Prost
75.0	Charles Leclerc
75.0	Jim Clark
75.0	Max Verstappen
75.0	Sebastian Vettel
73.33	John Surtees
72.73	Mika Häkkinen
70.0	Michael Schumacher
69.23	Jacques Laffite
68.75	Dan Gurney
66.67	Pierre Gasly
66.67	Heinz-Harald Fren...
66.67	Jody Scheckter
66.67	Sébastien Buemi
66.67	Jean-Éric Vergne
63.64	Sergio Pérez
63.64	Alan Jones
63.64	Nico Rosberg
62.5	Elio de Angelis
62.5	Damon Hill
62.5	Carlos Sainz

Figura C.3: Ranking de pilotos más dominantes a lo largo de su carrera

Por clarificar el significado de esta métrica, `dominationPerc` nos indica el porcentaje de temporadas en las que un piloto ha dominado a su compañero de equipo.

Podemos observar que el primero de ellos es Felipe Nasr que, a pesar de tener una corta carrera de dos años, consiguió dominar a todos sus compañeros de equipo. Más notable es el caso de Fernando Alonso, que a pesar de su larga carrera en el deporte ha conseguido dominar al 88.89 % de sus compañeros de equipo. En las posiciones más altas de esta clasificación podemos ver a otros pilotos notables como James Hunt, Emerson Fittipaldi, Charles Leclerc o Lewis Hamilton.

Por poner varios ejemplos de métricas, podemos ver los pilotos con mejor posición de salida:

fullName	avgGridStart	rankAGS
Juan Fangio	2.36	1
Ayrton Senna	3.13	2
Nino Farina	3.19	3
Jim Clark	3.36	4
Luigi Fagioli	3.63	5
Lewis Hamilton	3.68	6
Alain Prost	4.14	7
Jackie Stewart	4.34	8
Michael Schumacher	4.91	9
José Froilán Gonz...	5.14	10
Stirling Moss	5.37	11
Juan Pablo Montoya	5.37	11
Max Verstappen	5.63	13
Damon Hill	5.79	14
Mike Hawthorn	5.9	15
Sebastian Vettel	6.05	16
Valtteri Bottas	6.06	17
Jochen Rindt	6.24	18
Nigel Mansell	6.28	19
Alberto Ascari	6.33	20
Gerhard Berger	6.37	21
Tony Brooks	6.38	22
Nico Rosberg	6.9	23
Dan Gurney	7.03	24
Mika Häkkinen	7.07	25
James Hunt	7.13	26
Denny Hulme	7.21	27
Nelson Piquet	7.38	28
Jack Brabham	7.41	29
Richie Ginther	7.52	30

Figura C.4: Ranking de pilotos según su posición media de salida

O los pilotos con mayor posibilidad de clasificar primeros respecto al número de carreras en las que ha competido:

fullName	poleChance	rankPoleC
Juan Fangio	51.52	1
Jim Clark	46.6	2
Ayrton Senna	40.12	3
Lewis Hamilton	35.76	4
Alberto Ascari	34.78	5
Stirling Moss	25.78	6
Michael Schumacher	21.66	7
Sebastian Vettel	19.79	8
Jackie Stewart	18.58	9
Nigel Mansell	16.67	10
Damon Hill	16.39	11
Alain Prost	16.34	12
Nino Farina	16.22	13
Jochen Rindt	16.13	14
Mika Häkkinen	15.76	15
Nico Rosberg	14.56	16
Juan Pablo Montoya	13.68	17
Mario Andretti	13.61	18
James Hunt	12.96	19
Niki Lauda	12.77	20
Luigi Fagioli	12.5	21
Valtteri Bottas	11.17	22
Charles Leclerc	11.11	23
José Froilán Gonz...	10.81	24
Tony Brooks	10.71	25
René Arnoux	10.4	26
Nelson Piquet	10.34	27
Ronnie Peterson	10.0	28
Phil Hill	9.46	29
Jack Brabham	8.84	30

Figura C.5: Ranking de pilotos según su porcentaje de poles

C.3. Mejor temporada

Con esta query se buscaba averiguar cuál ha sido la mejor temporada de la historia. Para ello se han tenido en cuenta dos factores: el número de adelantamientos, el número de pilotos

distintos que han liderado la clasificación general y el número de pilotos que han ganado al menos una carrera.

Como en la query anterior, debemos tener en cuenta varias métricas y se va a utilizar la misma metodología: hacer un ranking de cada una y hacer la media de todos los rankings. Tras aplicar esto, podemos ver que el resultado es el siguiente:

year	overallRank	positionsGainedSeason	distinctLeaders	distinctWinners
2012	1	5077	4	8
2008	2	3123	4	7
2003	3	2955	3	8
1997	4	2642	3	6
2010	5	2747	6	5
1999	6	2151	3	6
2021	7	2901	2	6
2013	8	4697	2	5
2019	9	3201	2	5
2007	10	2974	3	4
2018	11	2692	2	5
2020	11	2612	2	5
2006	13	2524	2	5
2005	13	2325	2	5
2017	15	2167	2	5
2016	16	4613	2	4
2009	16	2768	1	6
2011	18	4627	1	5
2004	19	3194	1	5
2000	20	2744	2	4
2014	21	3873	2	3
2001	22	2256	1	5
2002	23	2294	1	4
2015	24	2747	1	3
1996	24	1946	1	4
1998	24	1827	1	4

Figura C.6: Ranking de temporadas

Sin embargo, esta query tiene una limitación grande: no tenemos datos de las posiciones de los pilotos vuelta a vuelta de antes de 1996. En este caso, una de las métricas quedaría como nulo para las temporadas anteriores a la mencionada, y al hacer la media de los rankings el resultado no sería válido. Aun así podríamos interpretar esta query como "la mejor temporada en la historia reciente del deporte".

Esto nos afecta también a la hora de calcular el piloto más consistente de la temporada, ya que en ese caso no podemos obtener el tiempo por vuelta.

C.4. Fabricante más dominante en un periodo concreto de tiempo

En este caso se buscaba ver cuál ha sido el fabricante con más victorias en carreras y campeonatos de un periodo concreto de tiempo. Para ello se realizaron unos conteos sencillos y se presentó la información resultante de la siguiente manera:

totalChampWins	totalRaceWins	name
5	47	Williams
3	23	McLaren
1	11	Benetton
1	6	Ferrari

Figura C.7: Dominio de constructores en la década de los 90

D

Repositorio de Github

En el enlace `https://github.com/nipsn/Spark-TFG` se puede consultar todo el material desarrollado a lo largo de esta práctica. El `README` es el siguiente:

```
Spark-TFG

Comparativa de las API de Spark en Scala y Python

Este repositorio contiene todo el contenido desarrollado para la realizacion del TFG del mismo
    titulo.

En concreto contiene:

 * Notebooks con el codigo desarrollado en la carpeta "notebooks"
 * Script de automatizacion de migracion + ejemplo de entrada y salida en la carpeta "autoregex"
 * Archivos relacionados con la memoria en "memoria".
 * Archivos relacionados con la presentacion en "presentacion".
 * En "data" esta la tabla desarrollada a partir de otras que relaciona piloto con constructor y
     temporada.
 * En "memoria" ademas tenemos la hoja de calculo con los datos recogidos para la comparativa en
     formato de hoja de calculo de LibreOffice.

Para realizar esta comparativa, han realizado las siguientes queries tanto en la API de Spark de
    Python (PySpark) como en la API de Scala:

 * Fabricante mas dominante en la decada de los 90 (campeonatos ganados y carreras ganadas).
 * Piloto mas consistente en la temporada 2012 (tiempos de vuelta en carrera vs tiempo en
     clasificacion)
 * Mejor piloto de la historia.
 * Temporada mas interesante para el espectador.
 * Analisis de temporada por piloto.
```