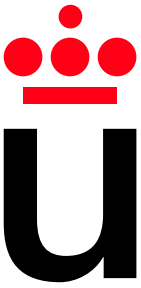

Universidad
Rey Juan Carlos

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

Breves agradecimientos o dedicatoria.

Resumen

Breve resumen del Trabajo de Fin de Grado (TFG). Recomendable entre 250-300 palabras, conteniendo los principales objetivos y resultados derivados del mismo.

Palabras clave:

- Python
- Ciberseguridad
- Aprendizaje automático (pueden ser varias)
- ...

Índice de contenidos

Índice de figuras

Índice de códigos

1
Introducción

Se puede añadir texto antes de empezar la primera sección.

1.1. Contexto y alcance

Contexto. Situar al lector. Objetivo general y alcance del trabajo.

1.2. Estructura del documento

La estructura del TFG no es fija. El tutor indicará una estructura adecuada dependiendo del trabajo concreto.

Se puede incluir dentro de cada apartado secciones adicionales. La copia en papel de la memoria del TFG será encuadernada en pasta dura de color azul (p.e. encuadernación tipo chanel). La portada, que puede ser una pegatina transparente, seguirá el modelo que se adjunta, que incluye el escudo y nombre de la URJC, la titulación cursada por el alumno, el curso académico, el título del TFG, el autor y el o los directores/tutores.

1.2.1. Trabajos de grados en informática

Una posible estructura de la memoria final asociada con cada TFG podría ser la siguiente (leed la normativa de TFG):

1. Introducción
2. Objetivos (incluyendo descripción del problema, estudio de alternativas y metodología empleada)
3. Descripción informática (puede incluir especificación, diseño, implementación y pruebas).
4. Experimentos / validación
5. Conclusiones (incluyendo los logros principales alcanzados y posibles trabajos futuros)
6. Bibliografía
7. Apéndices

1.2.2. Trabajos del grado en matemáticas

Una posible estructura de la memoria final asociada con cada TFG podría ser la siguiente:

1. Introducción
2. Objetivos (incluyendo descripción del problema, estudio de alternativas y metodología empleada)
3. Material y métodos / Metodología / Cuerpo del trabajo (describir las metodologías empleadas en el desarrollo del TFG o el desarrollo del mismo en caso de ser un trabajo de recopilación bibliográfica sobre un tema).
4. Resultados (opcional, dependiendo del tipo de trabajo desarrollado)
5. Conclusiones (incluyendo los logros principales alcanzados y posibles trabajos futuros)
6. Bibliografía
7. Apéndices

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. Para ello utilizaremos un conjunto de datos del dominio de la Fórmula 1 e intentaremos responder a las siguientes preguntas mediante queries como:

- Piloto más consistente en un periodo de tiempo concreto: 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.
- Piloto más dominante en un periodo de tiempo concreto 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.
- Similar al punto anterior, pero con fabricantes. Normalmente cada fabricante tiene varios pilotos, así que se tomarán como valor la media de todos los pilotos en cada métrica.
- En base a lo anterior, cuál ha sido el peor año de esa marca en ese periodo de tiempo teniendo en cuenta resultados de carrera, problemas de fiabilidad y paradas en boxes.
- Análisis de temporada por pilotos y constructores: se calcularán diversas medidas estadísticas para cada piloto o fabricante (utilizando la media de los valores de los pilotos en caso del fabricante). 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.
- Temporada más interesante para el espectador, teniendo en cuenta métricas como el número de adelantamientos, accidentes, retiradas de pilotos, más cambios de líder en la clasificación general, etc.

Además de responder a estas preguntas, también me planteo los siguientes objetivos:

- Visualizar de los resultados de las queries realizadas usando Plotly.
- Migrar queries desde PySpark a Scala Spark, centrando la explicación en las diferencias entre ambas APIs y en detalles a tener en cuenta al hacer una migración de este estilo.
- Medir y comparar el rendimiento de ambas API utilizando la Spark UI, que proporciona métricas de rendimiento en tiempo y memoria.
- Realizar queries a un cluster AWS EMR.

3

Descripción Informática

3.1. Fuentes de datos

Como se mencionó brevemente en el apartado de Objetivos, se ha utilizado un conjunto de datos de la Fórmula 1 que fue obtenido del siguiente enlace: click aquí. Concretamente, este dataset 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:

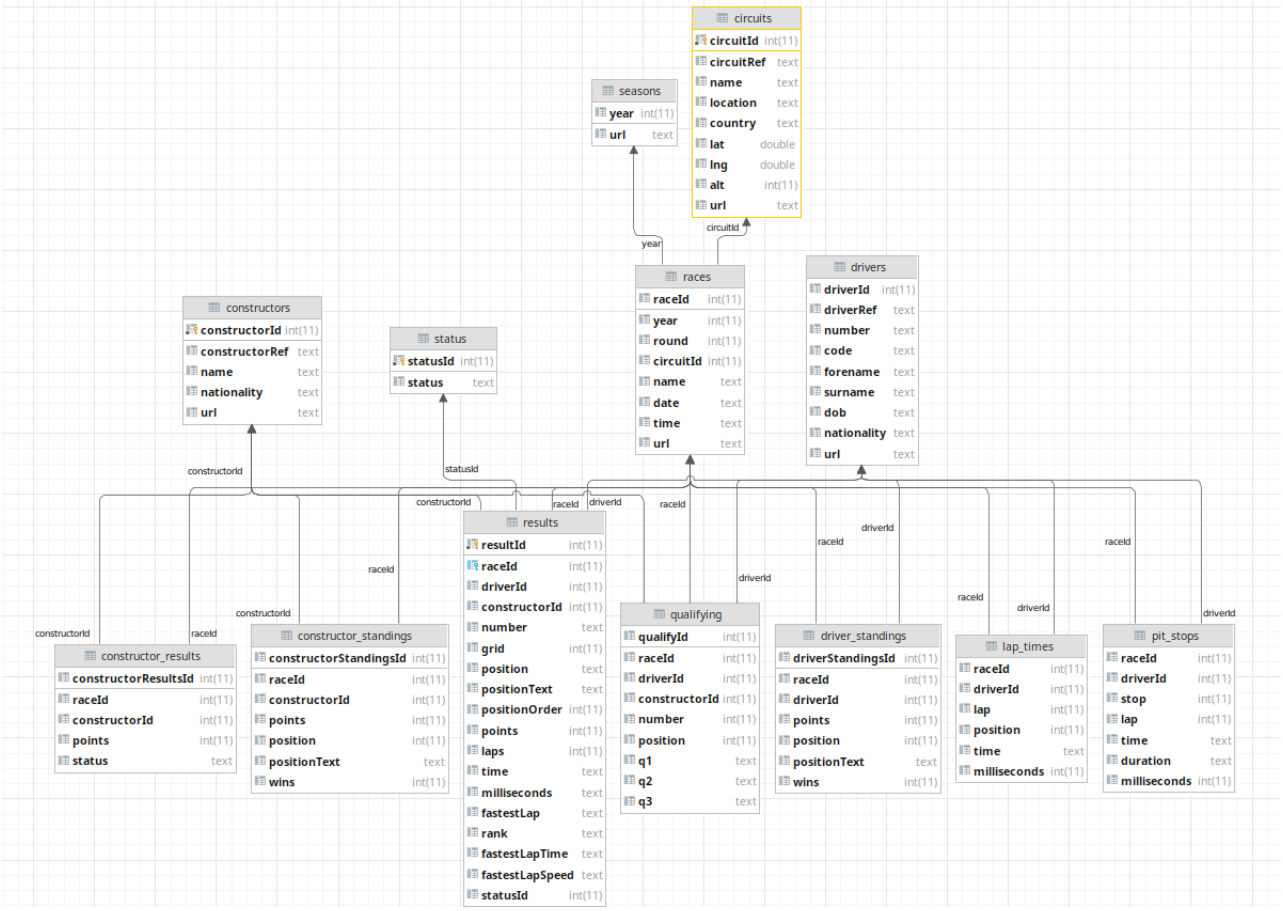

Figura 3.1: Diagrama Entidad-Relación

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 crear una tabla nueva que relacionase cada piloto con su constructor en

cada temporada. Principalmente se necesita para poder hacer comparativas entre pilotos del mismo equipo o bien globalmente 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

3.2. Programación de queries en Scala/Spark

3.2.1. Piloto más consistente en un periodo concreto de tiempo

En esta query intentaremos averiguar cuál ha sido el piloto más consistente en un periodo de tiempo dado. 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.

Necesitaremos cruzar varias fuentes de datos para esto:

- `races.csv`
- `lap_times.csv`
- `drivers.csv`
- `results.csv`

Primero de todo, queremos leer la fuente de datos `races.csv`, ya que nos permite filtrar por temporadas mediante la columna `year`. Para ello, ejecutamos las siguientes líneas de código:

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

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.

Tras esto 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 (ya sea en forma de lista o como un solo entero), 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 = spark.read.format("csv")
  .option("header", "true")
  .option("sep", ",")
  .load("data/races.csv")
  .withColumn("year", col("year").cast(IntegerType))
  .where(col("year").isInCollection(seasons))
```

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` se encuentre entre los valores que hemos recibido". Para ello podemos utilizar la función de columna `isInCollection`, que permite utilizar listas como filtros. En nuestro caso, `seasons` es la lista de temporadas en las que nos queremos centrar.

Resumiendo, con estas pocas líneas de código hemos obtenido todas las carreras 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 todas las queries en las que se requiere centrarse en un periodo concreto de tiempo, ya que la tabla `seasons` está, en mi opinión, incompleta y solamente contiene información de cada temporada. Es posible que más adelante añada funcionalidad a esta tabla con una columna que contenga todos los id de las carreras celebradas en esa temporada para ahorrar 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, primero tendremos que cargar la tabla `lap_times.csv` de la misma manera que hicimos anteriormente con `races.csv`. Después, 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 = spark.read.format("csv")
  .option("header", "true")
  .option("sep", ",")
  .load("data/lap_times.csv")
  .join(races, Seq("raceId"), "right")
  .withColumn("lapsPerDriver", count(col("lap")).over(driverWindow))
```

Como ya ha quedado claro cómo se carga información en formato CSV, paso a la siguiente línea, en la que 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. Más adelante pondré un ejemplo de esto mismo.

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, cargamos de nuevo la tabla `lap_times.csv`, 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 = spark.read.format("csv")
  .option("header", "true")
  .option("sep", ",")
  .load("data/lap_times.csv")
  .withColumnRenamed("time", "lapTime")
  // filtro las vueltas de las carreras en el periodo de tiempo dado
  .join(races, 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 cargar la tabla `results.csv` 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 hicieste 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:

```
spark.read.format("csv")
  .option("header", "true")
  .option("sep", ",")
  .load("data/results.csv")
  // filtro por temporada
  .join(races, 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, cargamos la tabla 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")
```

Una vez 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.

Al final, el código para obtener este DataFrame final quedaría tal que:

```
spark.read.format("csv")
  .option("header", "true")
  .option("sep", ",")
  .load("data/results.csv")
  // filtro por temporada
  .join(races, Seq("raceId"), "right")
  .na.drop(Seq("fastestLapTime"))
  // paso la vuelta rapida de tiempo por vuelta a ms
  .withColumn("fastestLapTimeMs", lapTimeToMsUDF(col("fastestLapTime")))
  // saco la media de vueltas rapidas
  .withColumn("avgFastestLapMs", avg(col("fastestLapTimeMs")).over(driverWindow))
  .dropDuplicates("driverId")
  .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)))
  // filtro pilotos "experimentados"
  .where(col("driverId").isInCollection(experiencedDrivers))
  // concateno el nombre y apellido de los pilotos
  .join(drivers, "driverId")
  .withColumn("driver", concat(col("forename"), lit(" "), col("surname")))
  .select("driver", "avgDiff")
  .orderBy("avgDiff")
```

3.2.2. Análisis de temporada por piloto

Esta query consiste en obtener una serie de métricas de cada piloto en una determinada temporada. Para ello, se obtiene como parámetro la temporada en cuestión, que usaremos para filtrar.

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. Como siempre, cargamos la tabla de la siguiente manera:

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

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.csv`, 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 = 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))
  .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` como si fuera un booleano. 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:

```
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")
```

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)
```

3.3. Programación de queries en PySpark

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.

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 = spark.read.format("csv")\
  .option("header", "true")\
  .option("sep", ",")\
  .load("../data/races.csv")\
  .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 ca-

rrera y temporada y las ventanas de particionado, podemos calcular el número de adelantamientos. Para ello hemos de cargar la tabla `lap_times`, que contiene información de todas las vueltas de cada piloto.

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

Después, viendo que tanto la columna `position` como `lap` son de tipo `String`, debemos pasarlo 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 cargamos 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 = spark.read.format("csv")\
  .option("header", "true")\
  .option("sep", ",")\
  .load("../data/results.csv")\
  .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"))\
```

Tras esto, tendremos 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 = spark.read.format("csv")\
  .option("header", "true")\
  .option("sep", ",")\
  .load("../data/results.csv")\
  .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 teniendolas 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 mostrarlo por pantalla. El código es el siguiente:

```
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")\
  .show()
```

3.3.2. 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.

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 cargará la tabla `races` y nos quedaremos con las columnas `raceId` y `year`.

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

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

```
driverInfo = spark.read.format("csv")\
  .option("header", "true")\
  .option("sep", ",")\
  .load("../data/drivers.csv")\
  .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 cargamos 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 = spark.read.format("csv")\
  .option("header", "true")\
  .option("sep", ",")\
  .load("../data/races.csv")\
  .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, cargamos 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 = spark.read.format("csv")\
  .option("header", "true")\
  .option("sep", ",")\
  .load("../data/results.csv")\
  .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, no entraremos al detalle de cómo se obtiene y simplemente cargaremos dicha tabla con:

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

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 cargaremos la tabla `driver_standings`, que interseccionaremos con `lastRaces` para quedarnos con la clasificación en la última carrera de cada temporada y con `driverConstSeasonMap` para completar con la información que nos proporciona dicha tabla.

```
driverDomination = spark.read.format("csv")\
  .option("header", "true")\
  .option("sep", ",")\
  .load("../data/driver_standings.csv")\
  .join(lastRaces, ["raceId"], "right")\
  .join(driverConstSeasonMap, ["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 cargamos la tabla `results` y filtramos los outliers interseccionando 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 = spark.read.format("csv")\
  .option("header", "true")\
  .option("sep", ",")\
  .load("../data/results.csv")\
  .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. Cargamos 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 interseccionar 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 = spark.read.format("csv")\
  .option("header", "true")\
  .option("sep", ",")\
  .load("../data/results.csv")\
  .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.

```
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`.

3.3.3. Migración de queries de Scala Spark a PySpark

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 3.2.2.

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 varios, entre ellos `max`, `min` o `avg`.

Por tanto, 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.

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:

```
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.

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)
```

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)
```

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.

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")
```

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")
```

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

Con estos cambios mencionados podrían cubrirse la mayoría de queries sencillas que nos encontremos, pero existen detalles importantes a tener en cuenta si el código utiliza funciones más avanzadas.

3.3.4. Expresiones regulares para facilitar la migración

En esta sección me gustaría hablar de una herramienta que puede facilitar mucho la migración de queries de Scala Spark a PySpark. Estoy hablando de ciertas expresiones regulares que nos ahorrarían potencialmente 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.
- Eliminación de `val` y `var`.
- Adición del carácter \ al final de cada línea.
- Traducción de listas, secuencias y arrays.
- Traducción de Mapas.
- Otros ajustes menores.

Sustituciones Directas

Podemos empezar hablando de sustituciones sencillas que no requieren del uso de expresiones regulares complejas. Un ejemplo de sustitución de este tipo serían los operadores lógicos `&&` y `||`. Simplemente podríamos sustituirlos por su variante de Python comentada anteriormente. Otros ejemplos incluyen:

- Comentarios de una sola línea: de `//` a `#`.
- Operadores de igualdad: `===` a `==` y `=!=` 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`

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 `'(.+?(?=[)| |.|,]))` detectamos el segundo y sustituimos por `col("$1")`

Con esto tendríamos gran parte de las invocaciones a columnas estandarizadas. Nótese que no hemos añadido el prefijo `F.`, ya que esto lo haremos a continuación de forma estándar para todas las funciones.

Para hacer esto debemos conseguir todos los nombres de todas las funciones del paquete `functions` y formar una expresión regular que las detecte. Esto lo podemos hacer de la siguiente manera:

```
from pyspark.sql import functions

regex = "(" + "|".join(dir(functions)[39:]) + ")\("
```

La función `dir` lista los contenidos de un paquete en este caso, y necesitamos descartar los 39 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.

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

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().

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

`(val|var)[ ]+?(?=[a-zA-Z])`

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.

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 las expresiones regulares anteriores:

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

Si aplicamos la expresión regular `(\))(\n)` y sustituimos por `$1\\$2`, la query quedaría tal que:

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

Sin embargo, esto sigue 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_]*)\n` , podemos sustituir por `$1 = $2\\\n`, de forma que la query ya quedaría como:

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

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]`

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:

```
\bMap\b\((([ |\n]*[^\n]*,[ |\n]*)*[^\n\.]*)\)
```

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)
```

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,`.

Con este conjunto de sustituciones directas y expresiones regulares 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 se puede hacer en Scala Spark que no se puede traducir directamente a PySpark, sino que requeriría una parte de interpretación, análisis de la funcionalidad y replicado de la misma.

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 más de mil 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.

Por estos motivo considero que crear una serie de expresiones regulares generalizadas para migrar código entre APIs, y que merecería un TFG de por sí.

3.4. Despliegue en AWS EMR

4

Experimentos / Validación

El primer paso para llevar a cabo esta query es cargar las fuentes de datos mencionadas. Para ello necesitamos haber creado un objeto `SparkSession`. En nuestro caso, esto se hace de la siguiente manera en el objeto `Main`:

```
val spark: SparkSession = SparkSession
  .builder()
  .master("local[*]")
  .getOrCreate()
```

En nuestro caso con estas opciones es suficiente, ya que estamos dedicando todos los núcleos de nuestra máquina local 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")`. Un parámetro que puede resultar muy útil modificar es el de `spark.sql.broadcastTimeout`, que por defecto tiene un valor de 300 (segundos), si no tenemos muchos recursos y vemos que la aplicación para inesperadamente con una excepción que muestra el mensaje "Could not execute broadcast in 300 secs". Para hacer esto, la creación de la `SparkSession` sería tal que:

```
val spark: SparkSession = SparkSession
  .builder()
  .master("local[*]")
  .config("spark.sql.broadcastTimeout", "36000")
  .getOrCreate()
```

De igual manera, si quisiéramos modificar algún parámetro distinto, lo haríamos añadiendo más modificaciones tal que:

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

4.1. Análisis de requisitos no funcionales

5

Conclusiones y trabajos futuros

En este capítulo se detallan las conclusiones derivadas del TFG y la propuesta de posibles trabajos futuros.

Las citas del texto Autor [1], Autor [2], Autor [3], Autor [4] y Autor [5] deben ir referenciadas en la bibliografia.

5.1. Texto de relleno

Bibliografía

[1] M. Giaquinta and S. Hildebrandt, *Calculus of variations II.* Springer Science and Business Media, 2013, vol. 311.

[2] S. Fortune and C. J. Van Wyk, "Efficient exact arithmetic for computational geometry," in *Proceedings of the Ninth Annual Symposium on Computational Geometry*, 1993, pp. 163–172.

[3] S. Fortune, "Voronoi diagrams and delaunay triangulations," *Computing in Euclidean geometry*, pp. 225–265, 1995.

[4] J. C. Mitchell, "Social networks," *Annual review of anthropology*, vol. 3, no. 1, pp. 279–299, 1974.

[5] C. B. Morrey Jr, *Multiple integrals in the calculus of variations.* Springer Science and Business Media, 2009.

Apéndices

A

Apéndice de figuras