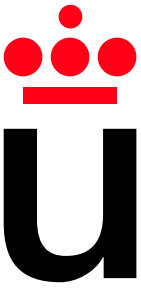

Universidad
Rey Juan Carlos

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:

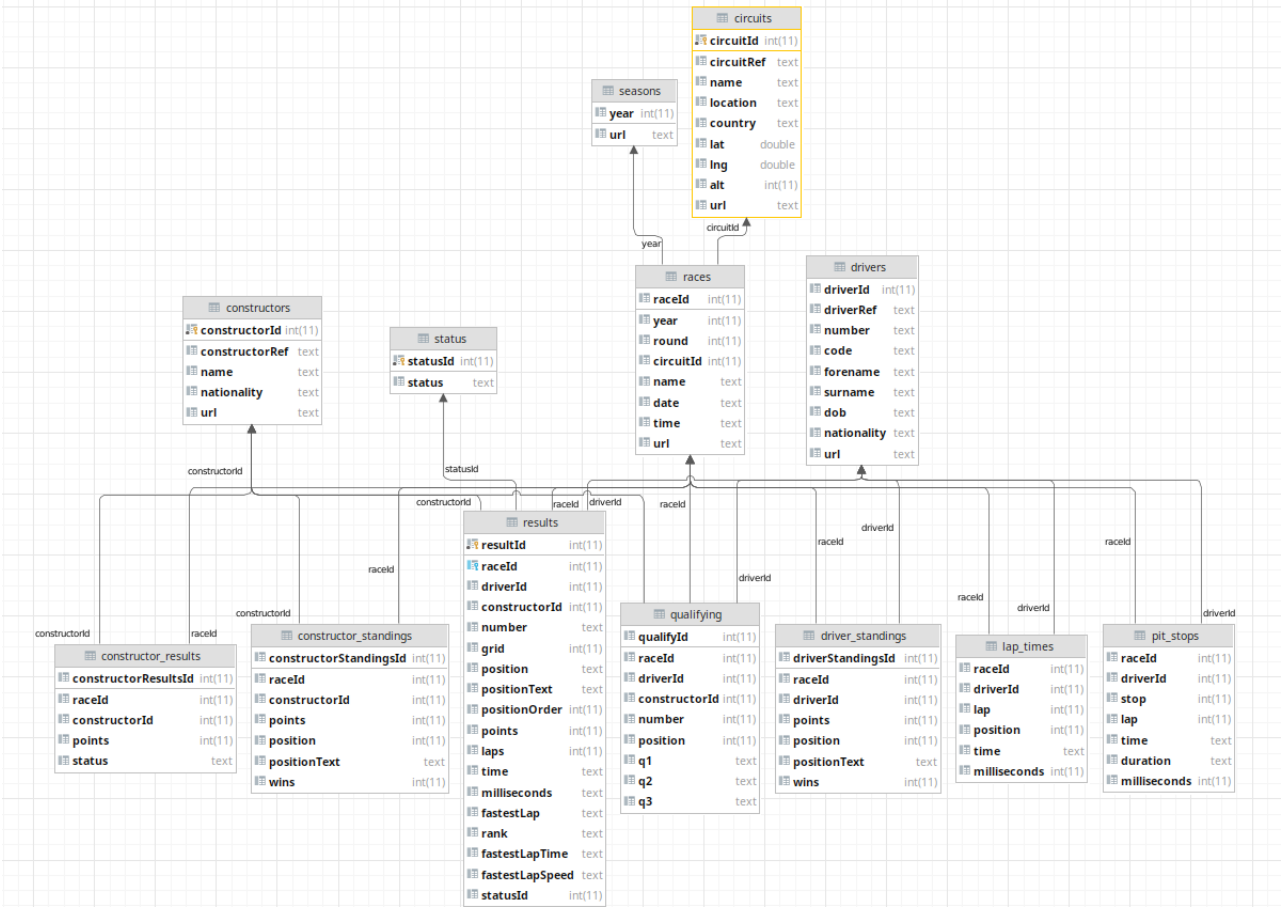

Figura 3.1: Diagrama Entidad-Relación

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. Dominio de fabricantes en la década de los 90

Con esta query se pretende hallar qué fabricante ha sido el más dominante en la década de los 90. En concreto intentaremos hallar el número de mundiales ganados y el número de carreras ganadas.

Se usarán las siguientes fuentes de datos:

- `races.csv`
- `constructor_standings.csv`
- `constructors.csv`

Al igual que en la query anterior, si queremos fijar nuestra atención en un periodo de tiempo, tenemos que hacerlo filtrando la columna `year` de la tabla `races.csv`. En este caso, necesitamos todas las carreras entre el año 1990 y el año 1999.

Una vez obtenidas todas las carreras de la década, tenemos que obtener la última carrera de cada temporada. Esto es debido a que en `constructor_standings.csv` tenemos la clasificación resultante al final de cada carrera. Para ello, usaremos crearemos una ventana en la que particionaremos los datos por año y que usaremos con la función `max()` sobre la columna `round`, que nos indica el índice de la carrera, es decir, la primera carrera de la temporada tendrá `round === 1`, para crear una columna llamada `max` en la que guardaremos el índice de la última carrera de la temporada. Finalmente, filtraremos los datos para quedarnos con aquellos en los que la columna `round === max`.

Tras esto último, unimos las tabla `constructor_standings.csv` con la recién obtenida para quedarnos con los resultados en las últimas carreras y filtramos según la columna `position === 1` para quedarnos con los ganadores. Teniendo esto, podemos ver también que la columna `wins` nos proporciona el número de victorias de cada escudería en esa temporada, así que, creando una ventana en la que particionemos por fabricante podemos hallar tanto la suma de victorias como el conteo de apariciones de cada una.

Pasando ya a la presentación de los datos, se filtrarían los constructores duplicados y se ordenarían los datos según el total de campeonatos ganados primero y, en caso de empate, por número de victorias. Además, se hace un join con la tabla de constructores para obtener su nombre.

Lo interesante de esta query es que se puede usar para cualquier periodo de tiempo. Podemos averiguar por ejemplo qué fabricante ha sido el más dominante en toda la historia de la competición y qué constructor ha dominado ciertos años concretos.

3.2.3. 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. Al final la query para esta tabla quedaría tal que:

```
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")

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

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 co-

lumna 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)
```

Al final, el código completo de la query queda tal que:

```
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")

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

  .select(
    col("code"),
    sum(col("points")).over(driverWindow).as("champPoints"),
    col("averagePoints"),
    round(col("averagePoints") /
        col("maxAvgPoints"),2).as("pointPercent"),
```

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

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

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

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc

vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.

Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

Sed commodo posuere pede. Mauris ut est. Ut quis purus. Sed ac odio. Sed vehicula hendrerit sem. Duis non odio. Morbi ut dui. Sed accumsan risus eget odio. In hac habitasse platea dictumst. Pellentesque non elit. Fusce sed justo eu urna porta tincidunt. Mauris felis odio, sollicitudin sed, volutpat a, ornare ac, erat. Morbi quis dolor. Donec pellentesque, erat ac sagittis semper, nunc dui lobortis purus, quis congue purus metus ultricies tellus. Proin et quam. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent sapien turpis, fermentum vel, eleifend faucibus, vehicula eu, lacus.

Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Donec odio elit, dictum in, hendrerit sit amet, egestas sed, leo. Praesent feugiat sapien aliquet odio. Integer vitae justo. Aliquam vestibulum fringilla lorem. Sed neque lectus, consectetuer at, consectetuer sed, eleifend ac, lectus. Nulla facilisi. Pellentesque eget lectus. Proin eu metus. Sed porttitor. In hac habitasse platea dictumst. Suspendisse eu lectus. Ut mi mi, lacinia sit amet, placerat et, mollis vitae, dui. Sed ante tellus, tristique ut, iaculis eu, malesuada ac, dui. Mauris nibh leo, facilisis non, adipiscing quis, ultrices a, dui.

Morbi luctus, wisi viverra faucibus pretium, nibh est placerat odio, nec commodo wisi enim eget quam. Quisque libero justo, consectetuer a, feugiat vitae, porttitor eu, libero. Suspendisse sed mauris vitae elit sollicitudin malesuada. Maecenas ultricies eros sit amet ante. Ut venenatis velit. Maecenas sed mi eget dui varius euismod. Phasellus aliquet volutpat odio. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Pellentesque sit amet pede ac sem eleifend consectetuer. Nullam elementum, urna vel imperdiet sodales, elit

ipsum pharetra ligula, ac pretium ante justo a nulla. Curabitur tristique arcu eu metus. Vestibulum lectus. Proin mauris. Proin eu nunc eu urna hendrerit faucibus. Aliquam auctor, pede consequat laoreet varius, eros tellus scelerisque quam, pellentesque hendrerit ipsum dolor sed augue. Nulla nec lacus.

Suspendisse vitae elit. Aliquam arcu neque, ornare in, ullamcorper quis, commodo eu, libero. Fusce sagittis erat at erat tristique mollis. Maecenas sapien libero, molestie et, lobortis in, sodales eget, dui. Morbi ultrices rutrum lorem. Nam elementum ullamcorper leo. Morbi dui. Aliquam sagittis. Nunc placerat. Pellentesque tristique sodales est. Maecenas imperdiet lacinia velit. Cras non urna. Morbi eros pede, suscipit ac, varius vel, egestas non, eros. Praesent malesuada, diam id pretium elementum, eros sem dictum tortor, vel consectetuer odio sem sed wisi.

Sed feugiat. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Ut pellentesque augue sed urna. Vestibulum diam eros, fringilla et, consectetuer eu, nonummy id, sapien. Nullam at lectus. In sagittis ultrices mauris. Curabitur malesuada erat sit amet massa. Fusce blandit. Aliquam erat volutpat. Aliquam euismod. Aenean vel lectus. Nunc imperdiet justo nec dolor.

Etiam euismod. Fusce facilisis lacinia dui. Suspendisse potenti. In mi erat, cursus id, nonummy sed, ullamcorper eget, sapien. Praesent pretium, magna in eleifend egestas, pede pede pretium lorem, quis consectetuer tortor sapien facilisis magna. Mauris quis magna varius nulla scelerisque imperdiet. Aliquam non quam. Aliquam porttitor quam a lacus. Praesent vel arcu ut tortor cursus volutpat. In vitae pede quis diam bibendum placerat. Fusce elementum convallis neque. Sed dolor orci, scelerisque ac, dapibus nec, ultricies ut, mi. Duis nec dui quis leo sagittis commodo.

Aliquam lectus. Vivamus leo. Quisque ornare tellus ullamcorper nulla. Mauris porttitor pharetra tortor. Sed fringilla justo sed mauris. Mauris tellus. Sed non leo. Nullam elementum, magna in cursus sodales, augue est scelerisque sapien, venenatis congue nulla arcu et pede. Ut suscipit enim vel sapien. Donec congue. Maecenas urna mi, suscipit in, placerat ut, vestibulum ut, massa. Fusce ultrices nulla et nisl.

Etiam ac leo a risus tristique nonummy. Donec dignissim tincidunt nulla. Vestibulum rhoncus molestie odio. Sed lobortis, justo et pretium lobortis, mauris turpis condimentum augue, nec ultricies nibh arcu pretium enim. Nunc purus neque, placerat id, imperdiet sed, pellentesque nec, nisl. Vestibulum imperdiet neque non sem accumsan laoreet. In hac habitasse platea dictumst. Etiam condimentum facilisis libero. Suspendisse in elit quis nisl aliquam dapibus. Pellentesque auctor sapien. Sed egestas sapien nec lectus. Pellentesque vel dui vel neque bibendum viverra. Aliquam porttitor nisl nec pede. Proin mattis libero vel turpis. Donec rutrum mauris et libero. Proin euismod porta felis. Nam lobortis, metus

quis elementum commodo, nunc lectus elementum mauris, eget vulputate ligula tellus eu neque. Vivamus eu dolor.

Nulla in ipsum. Praesent eros nulla, congue vitae, euismod ut, commodo a, wisi. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Aenean nonummy magna non leo. Sed felis erat, ullamcorper in, dictum non, ultricies ut, lectus. Proin vel arcu a odio lobortis euismod. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Proin ut est. Aliquam odio. Pellentesque massa turpis, cursus eu, euismod nec, tempor congue, nulla. Duis viverra gravida mauris. Cras tincidunt. Curabitur eros ligula, varius ut, pulvinar in, cursus faucibus, augue.

Nulla mattis luctus nulla. Duis commodo velit at leo. Aliquam vulputate magna et leo. Nam vestibulum ullamcorper leo. Vestibulum condimentum rutrum mauris. Donec id mauris. Morbi molestie justo et pede. Vivamus eget turpis sed nisl cursus tempor. Curabitur mollis sapien condimentum nunc. In wisi nisl, malesuada at, dignissim sit amet, lobortis in, odio. Aenean consequat arcu a ante. Pellentesque porta elit sit amet orci. Etiam at turpis nec elit ultricies imperdiet. Nulla facilisi. In hac habitasse platea dictumst. Suspendisse viverra aliquam risus. Nullam pede justo, molestie nonummy, scelerisque eu, facilisis vel, arcu.

Curabitur tellus magna, porttitor a, commodo a, commodo in, tortor. Donec interdum. Praesent scelerisque. Maecenas posuere sodales odio. Vivamus metus lacus, varius quis, imperdiet quis, rhoncus a, turpis. Etiam ligula arcu, elementum a, venenatis quis, sollicitudin sed, metus. Donec nunc pede, tincidunt in, venenatis vitae, faucibus vel, nibh. Pellentesque wisi. Nullam malesuada. Morbi ut tellus ut pede tincidunt porta. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam congue neque id dolor.

Donec et nisl at wisi luctus bibendum. Nam interdum tellus ac libero. Sed sem justo, laoreet vitae, fringilla at, adipiscing ut, nibh. Maecenas non sem quis tortor eleifend fermentum. Etiam id tortor ac mauris porta vulputate. Integer porta neque vitae massa. Maecenas tempus libero a libero posuere dictum. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Aenean quis mauris sed elit commodo placerat. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Vivamus rhoncus tincidunt libero. Etiam elementum pretium justo. Vivamus est. Morbi a tellus eget pede tristique commodo. Nulla nisl. Vestibulum sed nisl eu sapien cursus rutrum.

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