ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADO EN INGENIERÍA INFORMÁTICA

Curso Académico 2022/2023

Trabajo Fin de Grado

INGENIERÍA DE DATOS CON EL FRAMEWORK DE BIG DATA SPARK Y SCALA

Autor: Álvaro Sánchez Pérez

Directores: Juan Manuel Serrano Hidalgo

Índice

1. Introducción

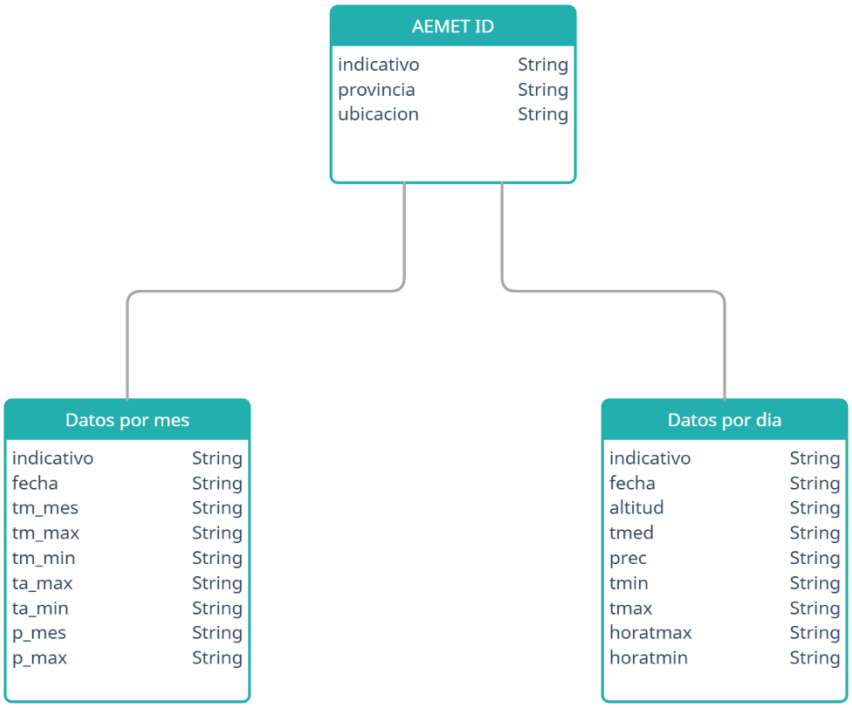
Prueba

2. Objetivos

3. Descripción informática

1. Fuentes de datos

Los datos meteorológicos, han sido obtenidos a través de la página de *AEMET OpenData*, se tratan tanto de datos que muestran la información de forma mensual como de forma diaria.



En el diagrama mostrado, aparecen los identificadores de las columnas y el tipo de valor por defecto cuando realizamos la descarga de estos datos. Resaltar que el diagrama anterior, solo se muestran las columnas más relevantes para la realización de las consultas. Podemos observar que mediante los datos de la tabla *AEMET ID*, podremos relacionar mediante el indicativo de la estación meteorológica, de qué provincia y ubicación se trata, ya que dentro de los datos no se nos ofrece este tipo de información.

La estructura en la que vienen estos datos también se puede observar en el anterior diagrama, todos los valores vienen en formato String, los cuales transformaremos mediante el casteo correspondiente gracias a Spark, para así poder trabajar posteriormente con ellos de una manera más cómoda. Deberemos de realizar lo siguiente para la transformación.

1. Primero creamos el esquema correspondiente a los datos, en este ejemplo lo realizaremos con los datos por día. En este esquema, le indicaremos tanto en nombre de la columna, como el tipo de dato que se trata, en nuestro caso todos String, y por último si la columna puede contener valores nulos.

```
val schema = StructType(
    Array(
        StructField("fecha", StringType, true),
        StructField("indicativo", StringType, true),
        StructField("nombre", StringType, true),
        StructField("provincia", StringType, true),
        StructField("altitud", StringType, true),
        StructField("tmed", StringType, true),
        StructField("prec", StringType, true),
        StructField("tmin", StringType, true),
        StructField("horatmin", StringType, true),
        StructField("tmax", StringType, true),
        StructField("horatmax", StringType, true),
        StructField("dir", StringType, true),
        StructField("velmedia", StringType, true),
        StructField("racha", StringType, true),
        StructField("horaracha", StringType, true),
        StructField("sol", StringType, true),
        StructField("presMax", StringType, true),
        StructField("horaPresMax", StringType, true),
        StructField("presMin", StringType, true),
        StructField("horaPresMin", StringType, true)
    )
)
```

2. A continuación, realizaremos la lectura de los datos y casteo de los datos.

```
val allData = spark.read.schema(schema).json("D:/TFGAlvaroSanchez/data/dayJSONLine/*.json")
    .withColumn("fecha", $"fecha".cast(DateType))
    .withColumn("altitud", $"altitud".cast(IntegerType))
    .withColumn("tmed", func.regexp_replace($"tmed", ",", ".").cast(DoubleType))
    .withColumn("prec", func.regexp_replace($"prec", ",", ".").cast(DoubleType))
    .withColumn("tmin", func.regexp_replace($"tmin", ",", ".").cast(DoubleType))
    .withColumn("tmax", func.regexp_replace($"tmax", ",", ".").cast(DoubleType))
    .withColumn("dir", $"dir".cast(IntegerType))
    .withColumn("velmedia", func.regexp_replace($"velmedia", ",", ".").cast(DoubleType))
    .withColumn("racha", func.regexp_replace($"racha", ",", ".").cast(DoubleType))
    .withColumn("sol", func.regexp_replace($"sol", ",", ".").cast(DoubleType))
    .withColumn("presMax", func.regexp_replace($"presMax", ",", ".").cast(DoubleType))
    .withColumn("horaPresMax", $"horaPresMax".cast(IntegerType))
    .withColumn("presMin", func.regexp_replace($"presMin", ",", ".").cast(DoubleType))
    .withColumn("horaPresMin", $"horaPresmin".cast(IntegerType))
```

2.1. Para la lectura le proporcionaremos el esquema creado anteriormente en la variable *schema*, y a su vez le indicamos en la ubicación donde se encuentran los datos, además del formato de fichero.

```
spark.read.schema(schema).json("D:/TFGAlvaroSanchez/data/dayJSONLine/*.json")
```

2.2. Finalmente, modificaremos el tipo de dato de las columnas que lo requieran. Usaremos *withColumn* donde primero le indicaremos el nuevo nombre de la columna que nos generará, en este caso usaremos el mismo nombre de la comuna que vamos a modificar para que así ocurra una sustitución de esta, y en la segunda parte será donde le indiquemos la columna con la que queremos trabajar y el cambio que deseamos.

```
.withColumn("fecha", $"fecha".cast(DateType))
```

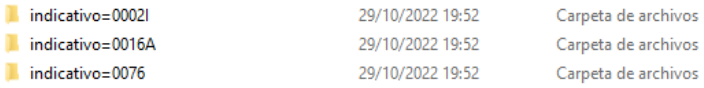
He de mencionar que, los datos que se tuvieron que transformar a tipo Double se debió realizar una modificación previa, sustituyendo las comas por puntos, ya que de otra manera no resultaba posible realizar el casteo debido a que para que se considere un numero decimal debe de venir separada la parte entera de la decimal mediante un punto.

```
.withColumn("tmed", func.regexp_replace($"tmed", ",", ".").cast(DoubleType))
```

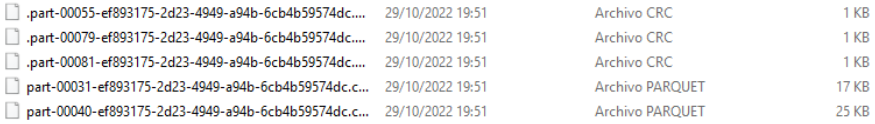
3. Por último, guardaremos los datos en formato Parquet, particionados mediante el campo *indicativo.*

```
allData.write.format("parquet").partitionBy("indicativo").mode("overwrite").save("D:/TFGAlvaroSanchez/data/dayParquet/")
```

Quedando particionado en memoria de la siguiente manera:



Y dentro de cada carpeta algo similar a lo siguiente:



Respecto a los datos mensuales nos quedaríamos únicamente con las siguientes columnas.

indicativo	fecha	tm_mes	tm_max	tm_min	ta_max	ta_min	p_mes	p_max
2331	2010-01-01	2.2	5.1	-0.7	10.7(19)	-16.0(10)	75.5	23.0(13)
2331	2010-02-01	3.0	6.2	-0.3	17.0(27)	-5.5(12)	65.6	11.4(08)
2331	2010-03-01	5.9	10.0	1.7	17.6(18)	-5.5(08)	63.0	11.4(20)
2331	2010-04-01	9.9	16.1	3.7	27.0(28)	-1.8(05)	29.3	9.0(18)
2331	2010-05-01	10.6	16.2	4.9	26.2(24)	-0.3(07)	39.3	7.9(08)
2331	2010-06-01	15.3	21.4	9.3	30.3(05)	4.0(20)	86.7	23.1(11)
2331	2010-07-01	20.5	28.4	12.5	34.5(07)	8.0(24)	5.5	4.4(01)
2331	2010-08-01	19.4	27.2	11.6	34.2(21)	7.6(31)	0.2	0.2(13)

De las que podríamos destacar las siguientes: *tm_mes* la cual nos muestra una temperatura media mensual de la ubicación, *tm_max* y *tm_min* siendo las temperaturas medias máximas y mínimas respectivamente, *ta_max* y *ta_min* como las temperaturas máximas y mínimas absolutas del mes y *p_mes* la cual nos indica precipitación total en ese mes. En caso de querer obtener información acerca de alguna otra variable, se puede acceder a esta de las siguientes maneras: a través del fichero *metadataMonth.json* incluido en la carpeta *data* del proyecto o realizando cualquier consulta sobre climatologías mensuales/anuales en la página de *AEMET OpenData* donde obtendrá un enlace acerca de los metadatos donde encontrar este tipo de información.

En relación con los datos diarios, tendríamos lo siguiente:

indicativo	fecha	altitud	tmed	prec	tmin	tmax	horatmax	horatmin
C447A	2013-01-01	632	13.9	0.1	12.3	15.5	15:58	05:50
C447A	2013-01-02	632	13.6	1.2	11.3	15.8	13:47	06:03
C447A	2013-01-03	632	13.4	0.0	11.2	15.6	12:28	19:57
C447A	2013-01-04	632	13.4	null	10.4	16.3	14:47	22:32
C447A	2013-01-05	632	13.6	null	9.5	17.8	13:47	22:46
C447A	2013-01-06	632	12.4	0.0	9.3	15.5	13:19	02:43

En este caso nos hemos quedado con todas las columnas, de las que resaltaríamos las siguientes: *tmed* la cual muestra la temperatura media diaria, *prec* que nos ofrece información acerca de la precipitación diaria, tmax *y tmin* las cuales nos muestran la temperatura máxima y mínima diaria respectivamente, y *horatmax* y *horatmin* que, de forma correspondiente, presentan la hora y minuto de la temperatura máxima y mínima. En caso de querer obtener información acerca de alguna otra variable, se puede acceder a esta de las siguientes maneras: a través del fichero *metadataDay.json* incluido en la carpeta *data* del proyecto o realizando cualquier consulta sobre climatologías diarias en la página de *AEMET OpenData* donde obtendrá un enlace acerca de los metadatos donde encontrar este tipo de información

He de mencionar que, a la hora de descargar los datos se encuentran en un formato JSON multilínea, el cual no llega a ser lo suficientemente óptimo para la lectura con Apache Spark. Por lo tanto, se realizaron cambios en todos los ficheros transformándolos a ficheros JSON que contuvieran toda la información en una única línea, esto se llevó a cabo mediante el siguiente comando utilizando la consola de Windows: **FOR %a IN (../data/*.json) DO jq . -c "%a" > "../JSONLine/%a”**. También cabe destacar que los datos una vez leídos en este formato JSON, la mayoría fueron transformados al formato Parquet, particionados por el indicativo de cada estación, ya que resulta más óptimo para la realización de consultas con Spark.

1.1. Obtención de los datos

Para la obtención de los datos, se crearon unos pequeños programas en el lenguaje Java. Estos programas, se basan en la simulación de los pasos que deberíamos de seguir para la descarga y los podemos encontrar en las carpetas *Descar*gaDatosPorDias y *DescargaDatosPorMeses*. Para la simulación de estos pasos, se hizo uso de la librería *Selenium*, a través de la cual se puede manejar un navegador web pudiendo realizar diversas acciones sobre la página mostrada. Por lo tanto, de manera resumida, los pasos que realizarían ambos programas para la descarga serían los siguientes.

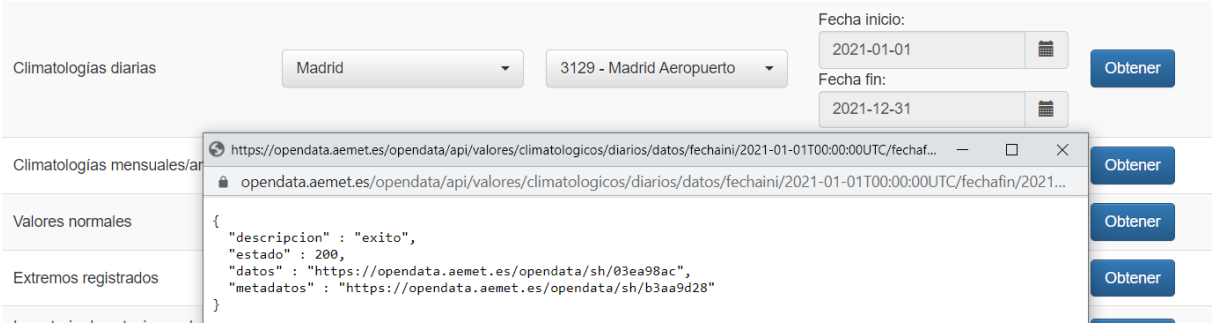
```
WebDriver driver = new ChromeDriver();
driver.get(baseUrl);
```

Abriríamos un nuevo navegador donde accederemos al sitio web de *AEMET OpenData*. En nuestro caso le pasamos la URL del sitio web a través de la variable *baseURL*.

```
driver.findElement(By.id("apikey")).sendKeys(apiKey);
desplegable1 = new Select(driver.findElement(By.id("clim1")));
desplegable1.selectByIndex(provincia);
```

Una vez dentro de esta página, el primer paso sería introducir la clave API correspondiente a nuestro usuario, está la podremos solicitar también a través de la misma página. Posteriormente, dependiendo del tipo de información que quisiéramos obtener, ya fuera información por días o por meses, buscaríamos los desplegables y elementos necesarios mediante su *id* o *xpath*.

Después de seleccionar la información deseada, se nos abriría una nueva página donde nos proporciona diferentes tipos de información, además del enlace de la página donde se encontrarán los datos deseados.



En caso de que la consulta a los datos correspondientes se haya realizado con éxito, accederemos al nuevo enlace que nos muestra, donde obtendremos la información y la guardaremos en un nuevo fichero con el nombre correspondiente a la consulta.

```
String texto = driver.findElement(By.xpath("//pre[contains(@style,'word-wrap')]")).getText();
if(texto.contains("\"descripcion\" : \"exito\"")) {
    String urlDatos = texto.split( regex: "\"")[9];

    driver.get(urlDatos);
    texto = driver.findElement(By.xpath("//pre[contains(@style,'word-wrap')]")).getText();

    //Imprimimos la informacion en un fichero externo
    PrintWriter printWriter = null;
    String ubicacionGuardar = "D:\\TFGAlvaroSanchez\\data\\day\\";
    String nombreFichero = ubicacionGuardar.concat(estacionMeterologica).concat(" (").concat(ano).cor

    try {
        printWriter = new PrintWriter(nombreFichero);
    } catch (FileNotFoundException e) {
        System.out.println("Unable to locate the fileName: " + e.getMessage());
    }
    Objects.requireNonNull(printWriter).println(texto);
    printWriter.close();
}
```

Por último, cerramos todas las pestañas y navegadores que se hubieran abierto, y procederíamos a realizar los mismos pasos con otro rango de fechas o en otra estación meteorológica, el objetivo es obtener la información de todas las estaciones existentes entre el rango de fechas establecido.

He de destacar también, que al comienzo del código existen una serie de variables que pueden ser cambiadas por los usuarios en caso de querer usar el programa. Podrá cambiar el controlador de Chrome en caso de que se esté usando una versión diferente del navegador, introducir su correspondiente clave AP, cambiar las fechas en caso de que se quisieran obtener datos en un rango diferente y por último indicar la ruta donde desea que se guarden los archivos JSON con la información.

```
String baseUrl = "https://opendata.aemet.es/centrodedescargas/productosAEMET?";
System.setProperty("webdriver.chrome.driver", "D:\\TFGAlvaroSanchez\\DescargaDatosPorDias\\chromedriver_win32\\chromedriver.exe");
String apiKey = "eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJhLnNhbmNoZXpwZS4yMDE5QGFsdW1ub3MudXJqYy5lcyIsImp0aSI6ImRhYTliMDk2LWI5OTgtNGY5Yi05
int anoInicio = 1995;
int anoFin = 2007;
String ubicacionGuardar = "D:\\TFGAlvaroSanchez\\data\\day\\";
```

2. Programación de queries

Año de la temperatura máxima promedio en cada mes

En esta consulta, realizaremos una comparación del mismo mes en diferentes años, para saber en qué año se registró la temperatura máxima promedia entre todas las estaciones en este mes. Por último, lo representaremos tanto mediante una tabla, como de manera gráfica para poder observar las diferencias de manera visual entre los distintos años.

Para realizar esta consulta utilizaremos los datos meteorológicos mensuales, ya que nos ofrecen la temperatura máxima registrada en cada mes de una manera bastante sencilla. Por lo tanto, el primer paso que deberíamos de realizar sería la lectura de estos datos.

A continuación, deberíamos de agrupar estos datos por fecha, ya que, al realizar la lectura, tenemos la información de muchas estaciones meteorológicas diferentes en el mismo mes y año, por lo tanto, los agruparemos haciendo que coincida la fecha, y realizando una media de la temperatura máxima en todas las estaciones. Por último, dividiremos los datos por meses y ordenándolos de manera descendente a través de nuestro objetivo, la temperatura máxima, ya que así estaríamos consiguiendo obtener el año, en el cual este mes fue el más caluroso.

```
import org.apache.spark.sql.expressions.Window

val data = spark.read.parquet("D:/TFGAlvaroSanchez/data/monthParquet/*").na.drop()

val window = Window.partitionBy("mes").orderBy($"ta_max".desc)

val dataWindow = data
    .withColumn("ta_max", func.split($"ta_max", "\\(")(0).cast(DoubleType))
    .groupBy($"fecha")
    .agg(func.avg($"ta_max").alias("ta_max"))
    .select(func.month($"fecha").alias("mes"), func.year($"fecha").alias("año"), $"ta_max")
    .withColumn("dense_rank", func.dense_rank().over(window))
    .filter($"dense_rank" === 1)
```

Como se puede observar, tal y como se describió anteriormente, el primer paso que realizamos es la lectura de los datos, los cuales los estamos leyendo en un formato Parquet para que resulte más óptimo y además estamos eliminando las filas que contienen valores nulos mediante la función *na.drop()*, la lectura de estos datos la almacenamos en la variable *data*. Al realizar la lectura de estos, se encontrarían de la siguiente manera.

```
+----------+--------+
|     fecha|  ta_max|
+----------+--------+
|2020-01-01|19.6(31)|
|2020-02-01|24.3(03)|
|2020-03-01|21.0(19)|
|2020-04-01|23.7(09)|
```

```
val window = Window.partitionBy("mes").orderBy($"ta_max".desc)
```

Por otro lado, también nos creamos la función ventana en la variable *window* que utilizaremos posteriormente. Le indicaremos la manera de la cual deseamos que se dividan los datos, en nuestro caso será a través de los meses y a su vez le señalamos la manera en la que se deberán de ordenar, teniendo en cuenta la temperatura máxima.

```
val dataWindow = data
    .withColumn("ta_max", func.split($"ta_max", "\\(")(0).cast(DoubleType))
```

A continuación, procedemos a realizar la consulta que almacenaremos en la variable *dataWindow*. El primer paso, que realizamos con los datos es la modificación de la columna de temperatura máxima, ya que esta nos viene con el siguiente formato temperaturaMáximaAlcanzada(díaDelMesEnElQueSeAlcanzó) por lo que Spark estaría tratando esta columna como un String, cuando nuestro objetivo es manejarlo como un tipo Double. Para esto realizamos lo siguiente, separamos mediante la función *Split* la temperatura máxima del día en que se alcanzó, y nos quedamos únicamente con la temperatura máxima, por último, realizamos un casteo al tipo de datos que deseamos.

```
.groupBy($"fecha")
.agg(func.avg($"ta_max").alias("ta_max"))
```

El segundo paso, de la consulta seria agrupar los datos de las diferentes estaciones por fecha, para ello utilizamos la función *groupBy* indicándole la columna a través de la cual deseamos realizar la agrupación. Mediante la función *agg* y *avg*, obtenemos y calculamos la media de la temperatura máxima de las filas que hemos agrupado. Actualmente tendríamos el DataFrame que se muestra a continuación, que contendría únicamente las fechas y sus respectivas temperaturas.

```
+----------+------------------+
|     fecha|            ta_max|
+----------+------------------+
|2013-01-01| 20.84285714285715|
|2010-06-01| 32.37096774193548|
|2021-03-01|  25.4304347826087|
|2016-10-01|28.869696969696975|
|2019-01-01| 18.59714285714286|
```

```
.select(func.month($"fecha").alias("mes"), func.year($"fecha").alias("año"), $"ta_max")
```

En el siguiente paso, obtendremos tanto el mes y el año, a partir de la fecha. Para esto utilizamos las funciones *month* y *year* con las cuales obtenemos tanto el mes y el año respectivamente. Mediante la función *select* seleccionaremos únicamente las columnas que vamos a necesitar.

```
.withColumn("dense_rank", func.dense_rank().over(window))
```

Para ir finalizando, añadimos una columna con el valor que nos devuelve la función *dense_rank*, que hace uso de la ventana *window* anteriormente definida. Mediante esta función, obtenemos la posición en la que se encuentra un valor teniendo en cuenta la expresión *orderBy* establecida en la función ventana (*window*). Por lo tanto, en nuestro nos devolverá una clasificación de los años más calurosos dividido por meses, ya que era la partición que se estableció en la función *window*. Quedando la consulta de la siguiente manera después de aplicar la función ventana.

```
+---+----+------------------+----------+
|mes| año|            ta_max|dense_rank|
+---+----+------------------+----------+
|  7|2022| 39.46785714285714|         1|
|  7|2015|             37.08|         2|
|  7|2016| 37.00499999999999|         3|
|  7|2020| 36.78799999999999|         4|
|  7|2017| 36.67575757575758|         5|
|  7|2013|35.952777777777776|         6|
|  7|2021| 35.90769230769231|         7|
|  7|2019| 35.85454545454545|         8|
|  7|2012|  35.5258064516129|         9|
|  7|2014|35.071875000000006|        10|
|  7|2010|          34.84375|        11|
|  7|2018|34.155882352941184|        12|
|  7|2011| 33.51379310344828|        13|
| 11|2015|25.378787878787882|         1|
| 11|2020|25.307407407407414|         2|
```

```
.filter($"dense_rank" === 1)
```

Por último, realizamos un filtrado quedándonos únicamente con las filas que poseen en la columna *dense_rank* un valor igual a 1, obteniendo así el año en el que se obtuvo la mayor temperatura máxima en un determinado mes.

Olas de calor

En esta consulta, desearemos obtener las diferentes olas de calor que han producido cada año en España, con el objetivo de observar si es verdad que se está produciendo un aumento de las temperaturas durante los últimos años. Para ello, calcularemos las olas de calor teniendo en cuenta la temperatura máxima diaria en cada estación meteorológica, y realizaremos los cálculos necesarios para finalmente mostrar los resultados sobre un mapa donde el usuario podrá elegir el año del cual desea obtener la información. Resaltar que, se considerará ola de calor cuando la temperatura máxima supere o iguale los 40 grados durante más de tres días consecutivos.

Debido a la exactitud que necesitamos para esta consulta se utilizarán los datos meteorológicos diarios, ya que requeriremos los datos de las temperaturas máximas con una gran exactitud en el marco temporal. Por lo tanto, el primer paso para comenzar con esta consulta sería realizar la lectura de estos datos.

Una vez leídos los datos, deberemos de eliminar aquellos donde no se llegue a alcanzar la temperatura establecida, en nuestro caso eliminaremos aquellos que tengan una temperatura menor a 40 grados. Posteriormente, deberemos de identificar posibles olas de calor, aun sin tener en cuenta la duración de estas, para ello dividiremos los datos por estación meteorológica y año, y a cada fecha se le asignara un identificador, en caso de que las fechas que sean consecutivas se les asignara el mismo identificador. Por ir resumiendo, ahora mismo lo que tendríamos serían los días que presentan temperaturas mayores o iguales a 40 grados, donde además se les habrá asignado un identificador, con el que posteriormente podremos determinar la duración de la ola de calor.

A continuación, agruparemos los datos por el identificador asignado a cada fecha, donde contaremos el número de apariciones de cada identificador, y por último eliminaremos aquellos no válidos, quedándonos únicamente con los que se encuentran más de 3 veces. Ya que como se comentó anteriormente, gracias al número de veces que se encuentre un identificador podremos determinar la duración de la ola de calor.

```
import org.apache.spark.sql.expressions.Window

val data = spark.read.parquet("D:/TFGAlvaroSanchez/data/dayParquet/")
val stations = spark.read.option("delimiter", ";").csv("D:/TFGAlvaroSanchez/data/aemetID.csv")
    .toDF("provincia", "indicativo", "ubicacion")

val window = Window.partitionBy($"indicativo", $"año").orderBy($"fecha")

val results = data
    .filter(!func.isnull($"tmax") && $"tmax" >= 40)
    .withColumn("año", func.year($"fecha"))
    .withColumn("n_fila", func.row_number().over(window))
    .withColumn("id", func.expr("date_sub(fecha, n_fila)"))
    .groupBy($"indicativo", $"año", $"id")
    .agg(func.count($"id").alias("dias"), func.avg($"tmax"), func.max($"tmax"), func.min($"tmax"))
    .filter($"dias" > 3)
    .join(stations, "indicativo")
    .select($"ubicacion", $"provincia", $"año", $"dias", func.round($"avg(tmax)", 2).alias("avg(tmax)"),
            $"max(tmax)", $"min(tmax)")
```

```
val resultsSave = results
    .groupBy($"provincia", $"año")
    .agg(func.count($"provincia"), func.avg($"dias"), func.avg($"avg(tmax)"), func.max($"max(tmax)"), func.min($"min(tmax)"))
    .select($"provincia", $"año", $"count(provincia)".alias("nº de olas de calor"),
            $"avg(dias)".alias("duracion media"), $"avg(avg(tmax))".alias("avg(tmax)"),
            $"max(max(tmax))".alias("max(tmax)"), $"min(min(tmax))".alias("min(tmax)"))

resultsSave
    .withColumnRenamed("nº de olas de calor", "nOlasCalor")
    .withColumnRenamed("duracion media", "duracionMedia")
    .withColumnRenamed("avg(tmax)", "avgTmax")
    .withColumnRenamed("max(tmax)", "maxTmax")
    .withColumnRenamed("min(tmax)", "minTmax")
    .write.format("parquet").partitionBy("provincia").mode("overwrite").save("D:/TFGAlvaroSanchez/data/resultadoOlasCalor/")
```

La consulta se podría dividir en dos partes, una primera donde calculamos las diferentes olas de calor en cada estación meteorológica, y una segunda donde se agruparían estas olas de calor por provincias y obtenemos la información necesaria para representarla posteriormente en forma de mapa.

El primer paso como se indicó anteriormente sería la lectura de los diferentes datos. Primero de ello, leeremos los datos meteorológicos, los cuales guardaremos en la variable *data*. Podemos observar únicamente 3 columnas ya que, al tratarse de una lectura de tipo Parquet Spark únicamente lee las columnas que utiliza.

```
+----------+----------+----+
|indicativo|     fecha|tmax|
+----------+----------+----+
|     C447A|2013-01-01|15.5|
|     C447A|2013-01-02|15.8|
|     C447A|2013-01-03|15.6|
|     C447A|2013-01-04|16.3|
|     C447A|2013-01-05|17.8|
```

También leeremos el fichero a través del cual relacionaremos el indicativo de las estaciones con su provincia y ubicación. Estos datos los guardaremos en la variable *stations* y contendrá una información similar a la siguiente:

```
+---------+----------+--------------------+
|provincia|indicativo|           ubicacion|
+---------+----------+--------------------+
| A Coruna|      1351|     Estaca de Bares|
| A Coruna|     1363X|           As Pontes|
| A Coruna|      1387|            A Coruna|
| A Coruna|     1387E| A Coruna Aeropuerto|
```

```
val window = Window.partitionBy($"indicativo", $"año").orderBy($"fecha")
```

Además, al comienzo de la consulta nos crearemos nuestra función ventana en la variable *window*. La utilizaremos posteriormente puesto que, a través de esta dividiremos los datos mediante el año y su estación meteorológica, y a su vez ordenaremos estas particiones mediante la fecha de manera ascendente.

```
val results = data
    .filter(!func.isnull($"tmax") && $"tmax" >= 40)
```

Seguidamente, comenzamos con la primera parte de la consulta donde obtendremos las distintas olas de calor en cada estación, guardaremos esta información en la variable *results*. El primero de los pasos, será eliminar las filas que no utilizaremos, para ello realizamos un filtro en la columna *tmax*, la cual nos proporciona la información de la temperatura máxima, eliminando aquellas filas que no contengan ningún tipo de información, y aquellas que no superen la temperatura mínima establecida.

```
.withColumn("año", func.year($"fecha"))
```

A continuación, haremos uso de la función de Spark *year* con la cual obtendremos el año a través de la columna *fecha*. Guardaremos esta nueva información en una nueva columna a la que llamaremos *año*.

```
.withColumn("n_fila", func.row_number().over(window))
```

A su vez, también crearemos una nueva columna a la cual llamaremos *n_fila*, en esta le asignaremos un número a cada fila utilizando la función *row_number*, la cual a su vez hará uso de la función ventana que hemos creado anteriormente. Mediante la función *row_number*, asignaremos valores de manera secuencial comenzando desde el 1. Y gracias a la función ventana, estamos consiguiendo asignar un numero de fila que va irá en aumento dependiendo de la fecha ya que era el orden que le habíamos establecido, a su vez, cada vez que se trate de un nuevo año o de una nueva estación comenzará a contar de nuevo desde el 1, ya que eran las divisiones que le habíamos indicado cuando nos creamos la función. Quedándonos la consulta por el momento de la siguiente manera:

```
+----------+----------+----+----+------+
|indicativo|     fecha|tmax| año|n_fila|
+----------+----------+----+----+------+
|     0149X|2015-07-05|40.3|2015|     1|
|     0149X|2015-07-07|40.2|2015|     2|
|     0149X|2019-06-28|42.4|2019|     1|
|     0149X|2021-08-12|41.0|2021|     1|
|     0149X|2021-08-13|40.4|2021|     2|
|     1002Y|2020-07-30|40.0|2020|     1|
|     1002Y|2022-06-18|42.0|2022|     1|
|     1002Y|2022-07-17|40.7|2022|     2|
|     1002Y|2022-07-18|43.4|2022|     3|
|     1002Y|2022-07-24|41.1|2022|     4|
```

```
.withColumn("id", func.expr("date_sub(fecha, n_fila)"))
```

Continuaremos creándonos una nueva columna llamada *id*, en esta realizaremos una resta de la columna *fecha* con el *n_fila*. De esta manera estaríamos consiguiendo identificar las distintas

posibles olas de calor, ya que en caso de tratarse de días consecutivos en la nueva columna *id* contendrán el mismo valor. Por ejemplo, en la anterior imagen tenemos estas filas:

```
+----------+----------+----+----+------+
|indicativo|     fecha|tmax| año|n_fila|
+----------+----------+----+----+------+
|     1002Y|2022-07-17|40.7|2022|     2|
|     1002Y|2022-07-18|43.4|2022|     3|
```

Como podemos observar se tratan de fechas consecutivas, por lo tanto, si realizamos la resta de la columna *fecha – n_fila* nos quedaría lo siguiente en ambas:

```
+----------+----------+----+----+------+----------+
|indicativo|     fecha|tmax| año|n_fila|        id|
+----------+----------+----+----+------+----------+
|     1002Y|2022-07-17|40.7|2022|     2|2022-07-15|
|     1002Y|2022-07-18|43.4|2022|     3|2022-07-15|
```

Como podemos, observar tendrían el mismo valor en la columna id y gracias a este estaríamos identificando días consecutivos con temperaturas altas. Por lo tanto, este proceso lo realizamos con todos nuestros datos, y tendríamos identificadas las distintas posibles olas de calor, ya que aun deberemos tener él cuenta el número de días que duran para que se consideren validas.

```
.groupBy($"indicativo", $"año", $"id")
.agg(func.count($"id").alias("dias"), func.avg($"tmax"), func.max($"tmax"), func.min($"tmax"))
```

Retomando la consulta, ahora agruparemos las distintas posibles olas de calor, para ello realizaremos *groupBy* con las columnas *indicativo*, *año* e *id*. Posteriormente, mediante la función *agg*, llamaremos a diferentes funciones Spark a través de las cuales obtendremos información. Por ejemplo, mediante la función *count* con la columna *id*, obtendremos la duración en días de las altas temperaturas, ya que estaremos contando el número de veces que podemos encontrar ese mismo *id* en la misma estación meteorológica y año, a su vez, haciendo uso de la función *alias* le establecemos el nombre a la nueva columna que esta nos generará. Mediante las funciones *avg*, *max* y *min* obtendremos la temperatura media, máxima y mínima respectivamente acerca de esa posible ola de calor.

```
.filter($"dias" > 3)
```

Para ir finalizando con la primera parte de la consulta, realizamos un nuevo filtrado de los datos, en este caso quedándonos con lo que nosotros hemos considerado olas de calor, aquellas cuya duración es mayor a 3 días.

```
.join(stations, "indicativo")
```

Realizamos una unión mediante la función *join* con nuestros datos guardados en la variable *stations*, la cual contenía información extra acerca de las estaciones. Estos datos tenían tres

columnas, *provincia*, *indicativo* y *ubicación*, por lo tanto, los unimos mediante la columna común a ambas tablas, *indicativo*.

```
.select($"ubicacion", $"provincia", $"año", $"dias", func.round($"avg(tmax)", 2).alias("avg(tmax)"),
        $"max(tmax)", $"min(tmax)")
```

Por último, elegimos las columnas relevantes para nuestro resultado y mediante la función *round* redondeamos el valor perteneciente a la columna *avg(tmax)*, estableciéndole que contenga únicamente dos decimales.

Ahora mismo, tendríamos en la variable *results* las distintas olas de calor en cada estación meteorológica, viéndose de la siguiente manera:

ubicacion	provincia	año	dias	avg(tmax)	max(tmax)	min(tmax)
Loja	Granada	2021	4	44.28	45.6	42.3
Navalmoral de la ...	Caceres	2015	4	40.7	41.2	40.3
Navalmoral de la ...	Caceres	2015	4	40.95	42.5	40.3
Baza	Granada	2007	4	40.85	41.3	40.1
Antequera	Malaga	2017	5	42.68	44.4	40.1
Merida	Badajoz	2017	5	42.46	44.0	40.5
La Roda de Andalucia	Sevilla	2015	5	41.14	43.2	40.0
La Roda de Andalucia	Sevilla	2015	4	40.6	41.1	40.2
ecija	Sevilla	2007	6	42.08	43.2	41.0
Madrid Aeropuerto	Madrid	2022	4	41.0	42.2	40.5

Donde la columna *dias* representaría la duración de esa ola de calor, *avg(tmax)* la temperatura media, *max(tmax)* y *min(tmax)* las temperaturas máximas y mínimas respectivamente.

En la segunda parte de la consulta, preparamos los datos para su representación en el mapa provincial de España, además de guardarlos en un formato Parquet para su posterior lectura en Python, ya que actualmente en Scala no es posible realizar una representación sobre un mapa.

```
val resultsSave = results
    .groupBy($"provincia", $"año")
    .agg(func.count($"provincia"), func.avg($"dias"), func.avg($"avg(tmax)"), func.max($"max(tmax)"), func.min($"min(tmax)"))
```

Como primer paso, nos creamos una variable que hemos llamado *resultsSave*, en ella se encontrarán los datos agrupados por provincias preparados para su posterior representación.

Por lo tanto, el siguiente paso a realizar seria agrupar los datos por provincia y año mediante la función *groupBy*. Mediante la función *agg* ejecutaremos diferentes funciones, por medio de la función *count* obtendremos el número de olas de calor en un mismo año en una provincia, ya que gracias a contar número de veces que aparece una provincia en los resultados anteriores podremos determinar este valor; y mediante las funciones *avg*, *max* y *min* obtendremos de manera respectiva la temperatura media, máxima y minina para esa provincia, con respecto a las olas de calor pertenecientes a un año.

```
.select($"provincia", $"año", $"count(provincia)".alias("nº de olas de calor"),
        $"avg(dias)".alias("duracion media"), $"avg(avg(tmax))".alias("avg(tmax)"),
        $"max(max(tmax))".alias("max(tmax)"), $"min(min(tmax))".alias("min(tmax)"))
```

Por último, seleccionamos las columnas deseadas mediante *select* y cambiamos el nombre de algunas de ellas mediante la función *alias*.

```
resultsSave
    .withColumnRenamed("nº de olas de calor", "nOlasCalor")
    .withColumnRenamed("duracion media", "duracionMedia")
    .withColumnRenamed("avg(tmax)", "avgTmax")
    .withColumnRenamed("max(tmax)", "maxTmax")
    .withColumnRenamed("min(tmax)", "minTmax")
    .write.format("parquet").partitionBy("año").mode("overwrite").save("D:/TFGAlvaroSanchez/data/resultadoOlasCalor/")
```

Para guardar los datos en formato Parquet, el nombre de las columnas no puede contener ningún espacio en blanco, por lo tanto, deberemos de darles un nuevo nombre a estas columnas. Lo realizamos a través de la función *withColumnRenamed* donde primero le indicamos el nombre actual de la columna, y a continuación el nuevo nombre que le queremos establecer.

3. Visualización de queries

Año de la temperatura máxima promedio en cada mes

Antes de pasar con la representación en forma de gráfica, se muestran también los resultados en forma de tabla, donde se puede obtener algo más de información, además de poder usarse para verificar que los datos que se muestran en la gráfica son correctos.

```
dataWindow
    .withColumn("temperatura maxima", func.round($"ta_max", 2))
    .orderBy($"mes".asc)
    .select($"mes", $"año", $"temperatura maxima")
    .show()
```

El anterior sería el código a ejecutar, para que quede de una manera más visual se redondea el valor de la temperatura máxima mediante las funciones *round* y *withColumn*. Además de ordenar los resultados por el número de mes de manera ascencente a través de *orderBy* y seleccionar las columnas relevantes a mostrar por medio de la función *select*.

```
+---+----+------------------+
|mes| año|temperatura maxima|
+---+----+------------------+
|  1|2021|             21.84|
|  2|2020|             24.42|
|  3|2017|             27.82|
|  4|2011|             29.48|
|  5|2015|             33.73|
|  6|2017|             36.86|
|  7|2022|             39.47|
|  8|2022|             37.74|
|  9|2016|             36.39|
| 10|2011|             31.52|
| 11|2015|             25.38|
| 12|2021|             22.29|
+---+----+------------------+
```

Para la representación de los resultados de forma gráfica, nos creamos la función *countByYear*, la cual nos facilitará los datos que utilizaremos posteriormente para la representación, ya que nos devuelve la cantidad de meses por cada año que se encuentran en la consulta realizada anteriormente, estos datos los guardaremos en variables con diferentes nombres dependiendo del año de la información.

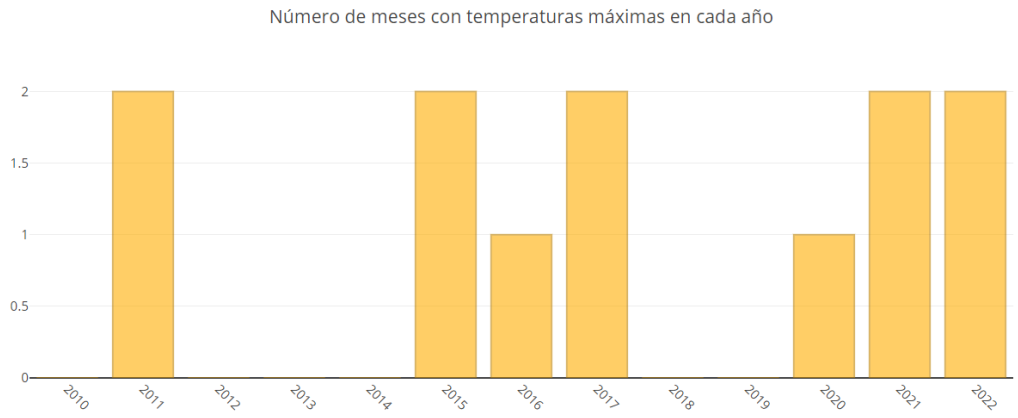
```
def countByYear(year : Int) : Long = {
    dataWindow
        .filter($"año" === year)
        .count()
}

val year2010 = countByYear(2010)
val year2011 = countByYear(2011)
```

A continuación, definiremos la manera en la cual deseamos mostrar la información.

```
val dataToPlot = Seq(
  Bar(
    Seq(2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020, 2021, 2022),
    Seq(year2010, year2011, year2012, year2013, year2014, year2015, year2016,
        year2017, year2018, year2019, year2020, year2021, year2022),
    marker = Marker(
        color = Color.RGB(255, 174, 0),
        opacity = 0.6,
        line = Line(
            color = Color.RGB(189, 129, 0),
              width = 1.5
        )
    )
  )
)

plot(dataToPlot)
```

Nos crearemos el grafico en una variable llamada *dataToPlot*, que posteriormente pintaremos por pantalla. En esta variable deberemos indicarle el tipo de grafico que deseamos, en nuestro caso le indicamos que queremos un grafico de barras gracias a la palabra *Bar*. Dentro de este le indicamos la información a representar, primero la información correspondiente al eje X mediante una secuencia de años, y en el eje Y le pasaremos los valores contenidos en las variables que hemos calculado anteriormente. Por último, mediante el atributo *marker*, definimos características visuales del gráfico, como el color, la opacidad y características acerca de la línea exterior, como grosor y color.

Para finalizar ofrecemos los resultados de forma gráfica haciendo uso del método *plot*.



Olas de calor

```
results
    .withColumnRenamed("dias", "duracion (dias)")
    .withColumnRenamed("avg(tmax)", "temperatura media")
    .withColumnRenamed("max(tmax)", "temperatura maxima")
    .withColumnRenamed("min(tmax)", "temperatura minima")
    .orderBy($"ubicacion", $"año")
    .show()
```

Podremos observar los resultados de diferentes maneras, una de ellas sería mostrar las diferentes olas de calor a través de una tabla, donde se han cambiado los nombres de algunas columnas para hacer el resultado mas visual, de esta manera podremos observar toda la información de manera detalla y ordenada por ubicación y año:

ubicacion	provincia	año	duracion (dias)	temperatura media	temperatura maxima	temperatura minima
Alajar	Huelva	2018	4	41.65	42.5	40.7
Alcaniz	Teruel	2019	4	40.95	41.2	40.8
Andujar	Jaen	2008	5	41.32	42.7	40.6
Andujar	Jaen	2009	4	41.5	42.7	40.4
Andujar	Jaen	2009	6	41.82	42.6	40.5
Andujar	Jaen	2010	4	42.35	43.1	40.9
Andujar	Jaen	2010	4	42.35	43.8	41.1
Andujar	Jaen	2010	7	41.39	42.7	40.4
Andujar	Jaen	2012	4	41.78	42.3	40.6
Andujar	Jaen	2015	4	42.38	43.6	40.8
Andujar	Jaen	2015	6	41.5	42.9	40.0
Andujar	Jaen	2015	6	41.92	44.2	40.0
Andujar	Jaen	2016	4	40.88	41.2	40.6

```
results
    .groupBy($"año")
    .agg(func.count($"año").alias("nº olas de calor"))
    .orderBy($"año")
    .show()
```

Otra manera sería la siguiente, con la que podremos observar el numero de olas de calor en los diferentes años. Se agruparían los años mediante la funcion *groupBy*, y contariamos las diferentes olas de calor gracias a las funciones *agg* y *count*. Como se puede ver en los ultimos años estas han aumentado de una manera considerable.

año	nº olas de calor
2007	6
2008	1
2009	9
2010	6
2012	6
2013	4
2014	1
2015	28
2016	11
2017	36
2018	23
2019	10
2020	11
2021	24
2022	68

Y por ultimo, una representación mediante un mapa, para requerimos anteriormente guardar los datos de los resultados, ya que en Scala aun no existen librerias que nos permitan este tipo de representaciones, por lo tanto haremos uso de Python.

Para esta representacion, importaremos las siguientes librerias en Python:

```
import json
import pandas as pd
import plotly.express as px
import ipywidgets as widgets
```

Utilizaremos la librería *json* para leer el fichero geoJSON, el cual es un fichero que nos ofrece datos geofraficos con el cual representaremos las diferentes pronvincias españolas. La librería *pandas* la usaremos para la lectura y manipulacion de los datos, mediante *ploty* podremos representar nuestro gráfico y por ultimo utilizaremos *ipywidgets* para poder utilizar elementos interactivos de HTML en Jupyter notebook.

Lo primero que realizaremos será la lectura tanto de los datos que vamos a representar, como del fichero geoJSON.

1. Para la lectura del fichero geoJSON, le deberemos de indicar el tipo de codificacion de caracteres que incluye el fichero. Posteriormente abrimos el fichero mediante la función *open*, donde le indicaremos la ruta del fichero, el modo en el que deseamos abrirlo y la codificacion. Por último, utilizaremos la librería *json* con la función *load* para obtener un objeto Python del fichero, en nuestro caso un diccionario.

```
enc = "utf-8"
f = open("D:/TFGAlvaroSanchez/data/spainProvinces.json", "r", encoding=enc)
provincias = json.load(f)
```

2. Posteriormente, leemos nuestros datos a representar, para esto hacemos uso de la librería *pandas* que nos permitirá leer ficheros Parquet usando la función *read_parquet* devolviéndonos un DataFrame en la variable *data*. También gracias a la función *rename* cambiamos los nombres de algunas columnas para que posteriormente cuando se realice la visualización quede de una manera más limpia.

```
data = pd.read_parquet("D:/TFGAlvaroSanchez/data/resultadoOlasCalor/")
data = data.rename(columns = {"nOlasCalor": "numero de olas de calor", "avgTmax": "temperatura media"})
```

A continuación, realizaremos una lectura del fichero geoJSON, para obtener tanto el nombre de la provincia como el identificador asignado en el mapa a esta. Necesitaremos este identificador, ya que será la manera a través de la cual le indicaremos la información a representar en esa área.

```
provinciaIdMap = {}
for feature in provincias["features"]:
    feature["id"] = feature["properties"]["cod_prov"]
    provinciaIdMap[feature["properties"]["name"]] = feature["id"]
```

Por lo tanto, lo primero que realizamos es crearnos un diccionario, donde guardaremos el nombre de la provincia como clave y su identificador como valor. Dentro del identificador *features* del archivo JSON podremos encontrar lo siguiente: una característica llamada *type*; *geometry*, donde se nos indicara la forma de la provincia; y *properties* donde encontraremos diferentes características como estas:

Aquí podremos obtener tanto el nombre de la provincia como el identificador establecido en el fichero geoJSON. Por lo tanto, iremos guardando toda esta información en el diccionario creado, quedado de esta manera:

```
'Teruel': '44',
'València/Valencia': '46',
'Valladolid': '47',
'Bizkaia/Vizcaya': '48',
```

Como se puede ver en la anterior imagen, podemos encontrar los nombres de las provincias con tildes o acentos graves. Deberemos de sustituir estos caracteres por el propio caracter, pero sin esta característica. Realizaremos un bucle que recorrerá las claves del diccionario, en caso de encontrar algún carácter con las características mencionadas, se realizará una sustitución por el correcto.

```
keys = list(provinciaIdMap.keys())
for key in keys:
    if key.find("á") != -1 or key.find("é") != -1 or key.find("í") != -1 or\
        key.find("ó") != -1 or key.find("ú") != -1 or key.find("è") != -1:
        valor = provinciaIdMap[key]
        provinciaIdMap.pop(key)
        key = key.replace("á", "a")
        key = key.replace("é", "e")
        key = key.replace("í", "i")
        key = key.replace("ó", "o")
        key = key.replace("ú", "u")
        key = key.replace("è", "e")
        provinciaIdMap[key] = valor
```

Para realizar la sustitución deberemos de guardar el valor que contenía la clave en una variable, en nuestro caso llamada *valor*, ya que vamos a eliminar la clave del diccionario para sustituirla por la nueva. Eliminamos la clave mediante el método *pop*, indicándole dentro de este la clave

a eliminar, y una vez cambiados los caracteres necesarios mediante la función *replace* se realiza una nueva inserción de la clave con su valor.

```
data['id'] = data['provincia'].apply(lambda x: provinciaIdMap[x])
```

En nuestros datos a representar, nos creamos una nueva columna que llamamos *id*, y para darle valor a esta columna hacemos uso de una función lambda, la cual nos devolverá el id correspondiente a la provincia en el mapa, gracias al diccionario creado anteriormente.

```
nMaxOlas = data["numero de olas de calor"].max()

def representarAno(ano):
    dataAux = data[data["año"] == ano]

    anadir = []
    for provincia in provinciaIdMap.keys():
        if len(dataAux[dataAux["provincia"] == provincia]) == 0:
            anadir.append([ano, 0, 0, 0, 0, 0, provincia, provinciaIdMap[provincia]])

    dataAux = pd.concat([dataAux, pd.DataFrame(anadir, columns=data.columns)], ignore_index=True)

    fig = px.choropleth(dataAux,
                        locations="id",
                        geojson=provincias,
                        color="numero de olas de calor",
                        scope="europe",
                        hover_name="provincia",
                        hover_data = ["temperatura media"],
                        color_continuous_scale = ["grey", "yellow", "orange", "red"],
                        range_color = [0, nMaxOlas])
    fig.update_geos(fitbounds="locations")
    fig.update_layout(margin = {"r":0,"t":0,"l":0,"b":0})
    fig.show()
```

Para ir finalizando, nos creamos el método al cual llamaremos cada vez que queramos realizar la representación de los datos.

1. El primero de los pasos, será obtener el valor máximo perteneciente al número de olas de calor, ya que, el rango de la escala de colores con la que representaremos la información tendrá como valor mínimo el 0 y el máximo el almacenado en esta variable, *nMaxOlas*, para que así se pueda observar una consistencia en la representación respecto a los diferentes años.

```
nMaxOlas = data["numero de olas de calor"].max()
```

2. A continuación, procedemos con el inicio del método, donde el primero de los pasos será realizar un filtrado de los datos quedándonos únicamente con aquellos del año que deseamos representar en la variable *dataAux*.

```
dataAux = data[data["año"] == ano]
```

3. Seguidamente deberemos de añadir datos para aquellas provincias que no presentan olas de calor, ya que en otro caso no se pintaría nada sobre el mapa al no disponer de

información. Por consiguiente, añadimos en estas provincias datos indicando que han existido 0 olas de calor durante ese año en nuestro DataFrame, en la variable *dataAux*. Para realizar esto, vamos recorriendo las claves del diccionario *provinciaIdMap*, que serán las distintas provincias, en caso de que con alguna de ellas no tuviera resultados en nuestro DataFrame añadiríamos una nueva columna con todos los datos a cero, y el nombre de la provincia y su identificador en el mapa.

```
anadir = []
for provincia in provinciaIdMap.keys():
    if len(dataAux[dataAux["provincia"] == provincia]) == 0:
        anadir.append([ano, 0, 0, 0, 0, 0, provincia, provinciaIdMap[provincia]])

dataAux = pd.concat([dataAux, pd.DataFrame(anadir, columns=data.columns)], ignore_index=True)
```

4. Por último, haremos uso de la librería *ploty* para la representación en el mapa, lo primero que debemos de indicarle son los datos a representar los cuales los tenemos en la variable *dataAux*; posteriormente en el atributo *locations* debemos de indicarle en que columna de *dataAux* podemos encontrar el identificador para relacionar esa información con la provincia en el mapa, en este caso estaríamos relacionando el *cod_prov* que venía en nuestro geoJSON con la columna *id* que hemos creado anteriormente; en el atributo *geoJSON*, le indicaremos en que variable se encuentra nuestro archivo geométrico; mediante *color* le indicamos la columna de los datos que deseamos representar; *scope* sirve para delimitar el continente a representar en nuestro caso Europa; *hover_name* es el nombre que queremos que se muestre cuando nos situemos encima de esa provincia en el mapa; *hover_data* será la información que queremos que se muestre al situarnos encima de la provincia; en *color_continous_scale* le indicaremos el rango de colores para la representación; y por último *range_color* atributo a través del cual le indicaremos el rango de valores mínimo y máximo para la escala, por lo tanto seria el mínimo sería el 0 y el máximo el valor contenido en la variable *nMaxOlas*.
   Para finalizar, mediante la función *update_geos* le estableceremos donde la ubicación donde queremos que se enfoque el mapa, con *update_layout* indicaremos los márgenes a la hora de representar el mapa y gracias al método *show* mostraremos el mapa por pantalla.

```
fig = px.choropleth(dataAux,
                    locations="id",
                    geojson=provincias,
                    color="numero de olas de calor",
                    scope="europe",
                    hover_name="provincia",
                    hover_data = ["temperatura media"],
                    color_continuous_scale = ["grey", "yellow", "orange", "red"],
                    range_color = [0, nMaxOlas])
fig.update_geos(fitbounds="locations")
fig.update_layout(margin = {"r":0,"t":0,"l":0,"b":0})
fig.show()
```

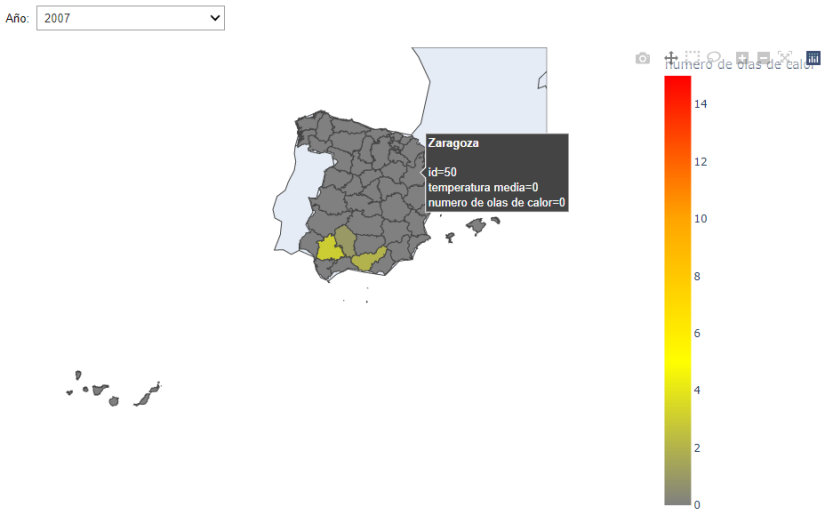
Para tratar de facilitar la representación y no tener que estar llamando a la anterior función de forma manual se añade un desplegable mediante el siguiente código, donde lo único que debe hacer el usuario sería cambiar el año a través de este y la representación se realizaría de forma automática. Se realiza de la siguiente manera:

```
anoInicio = min(data["año"].values.tolist())
anoFin = max(data["año"].values.tolist())
listaAnos = list(range(anoInicio, anoFin+1))

w = widgets.Dropdown(
    options = listaAnos,
    value = 2022,
    description="Año:"
)


widgets.interact(representarAno, ano = w)
```

Mediante la lista creada en la variable *listaAnos*, le indicamos la información a mostrar en el desplegable. Esta lista se creará con todos los valores entre el año tomado como inicio y el año indicado como final inclusive. Se realiza obteniendo todos los valores de la columna *años* mediante el método *values* y transformando estos a una lista mediante el método *tolist*. Una vez tenemos todos los valores en una lista, simplemente accederemos al valor menor y mayor mediante los métodos *min* y *max* respectivamente y los guardaremos en sus respectivas variables, *anoInicio* y *anoFin*. Para finalizar en la variable *listaAnos* generaremos la lista con todos los valores que se encuentren entre este rango, ambos incluidos, se realiza mediante el método *range*, que convertiremos a una lista mediante *list*.

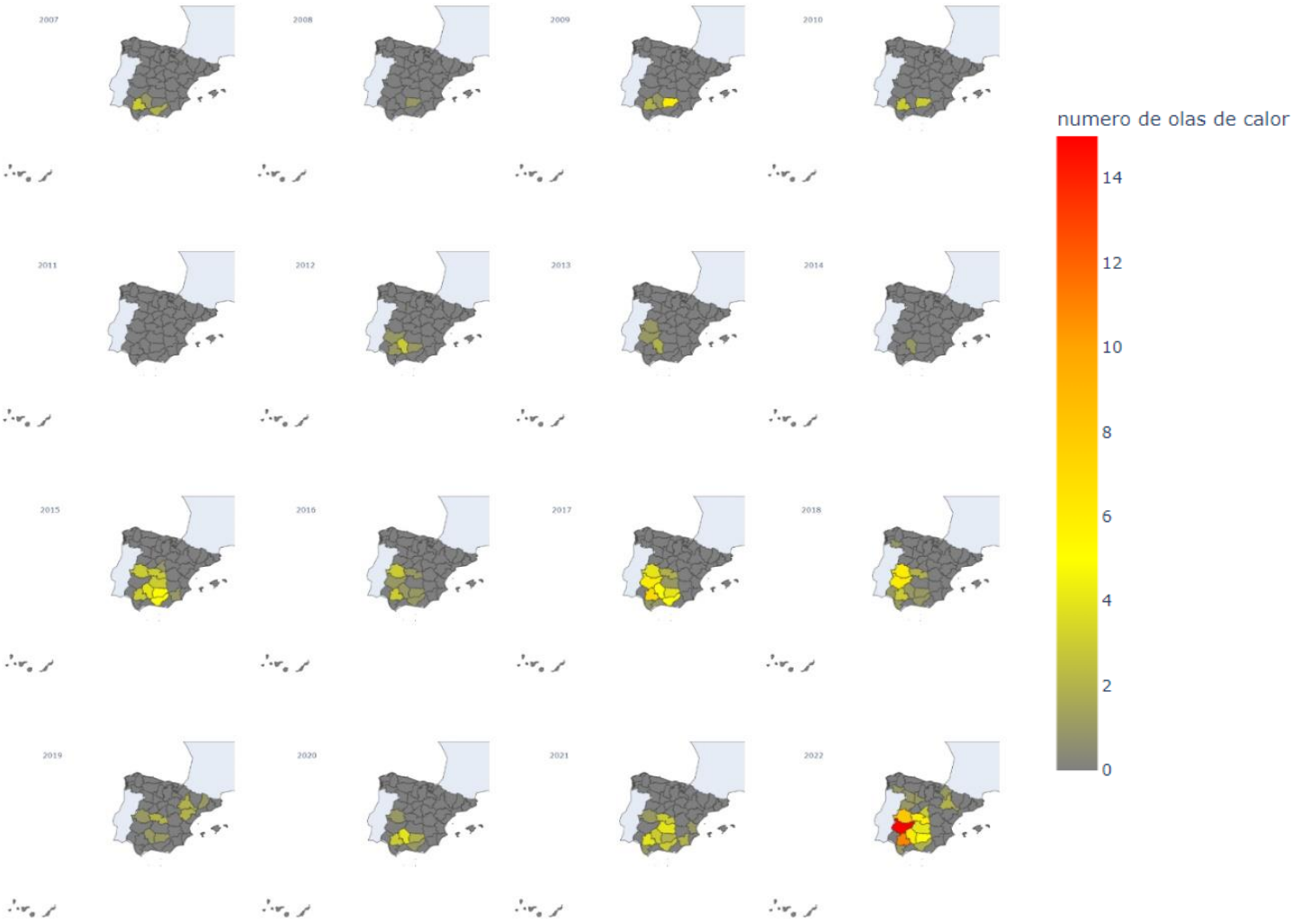
Gracias a la librería *ipywidgets* nos podremos crear un desplegable similar a los que podemos encontrar en HTML. Esto lo realizamos mediante la función *Dropdown* la cual nos creará un desplegable con las opciones que le indicamos en el atributo *options*, en nuestro caso los años con información disponible; por medio de *value* le indicamos el valor por defecto al iniciar el desplegable; y con *description* estableceremos un texto justo antes de este desplegable. Guardaremos este desplegable en la variable *w* que llamaremos posteriormente para hacer uso de él.

Gracias al método *interact* que nos proporciona esta librería, podemos indicarle el método al que deseamos llamar, y a continuación indicarle la manera a través le pasamos los distintos atributos, en nuestro caso aquí será donde le pasemos el desplegable anteriormente creado. Quedando todo esto de la siguiente manera:



Cuando vamos pasando el ratón por encima del mapa podremos obtener información adicional a la que se muestra, gracias a los atributos que le hemos establecido anteriormente.

En la siguiente imagen se podrá observar cómo ha ido cambiando el número de olas de calor en España durante los diferentes años:



4. Despliegue en AWS EMR

Para desplegar exitosamente una de nuestras consultas en un clúster, en este caso AWS Elastic MapReduce (EMR), es necesario seguir una serie de pasos específicos. Nosotros nos enfocaremos en el despliegue de la consulta de "olas de calor".

1. Como primer paso deberemos de realizar la instalación de AWS CLI, la cual es una herramienta de línea de comandos que nos permitirá interactuar con los servicios de AWS a través de una terminal local en nuestro ordenador.
2. En segundo lugar, procederemos a desarrollar nuestra aplicación que contendrá la consulta que deseamos ejecutar en el clúster. Para llevar a cabo esta tarea, hemos utilizado *IntelIJ* como entorno de desarrollo donde hemos creado un nuevo proyecto Scala utilizando sbt. Aquí se nos solicitaran varios parámetros de configuración, como la versión de Scala que deseamos utilizar. En este caso, hemos elegido utilizar la versión 2.12.8.

   Una vez que hayamos creado nuestro proyecto, crearemos un archivo llamado Main.scala, que será el lugar donde agregaremos el código específico para la consulta que deseamos ejecutar. Este archivo inicialmente se creará en forma de clase, por lo que debemos definirlo como un objeto y agregar la función principal de ejecución. Dentro de esta función, incluiremos el código necesario para llevar a cabo nuestra consulta. El código que se utilizará para realizar esta consulta es similar al que se describió anteriormente, pero con algunos pequeños cambios. En primer lugar, al inicializar el programa Spark, se realizará una modificación a la hora de inicializar la sesión Spark, quedando de la siguiente manera:

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

   Además, se añadirán algunas líneas al final del código para que se pueda ver en la consola de salida del clúster si la consulta se ha realizado correctamente y dónde se han guardado los resultados obtenidos. Esto nos ayudará a asegurarnos de que la consulta se ha ejecutado correctamente y donde podemos encontrar los resultados para su uso posterior.

   En nuestro caso, hemos implementado el uso de un archivo de configuración denominado config.conf, en el cual deberemos definir tres parámetros clave para el correcto funcionamiento del programa:

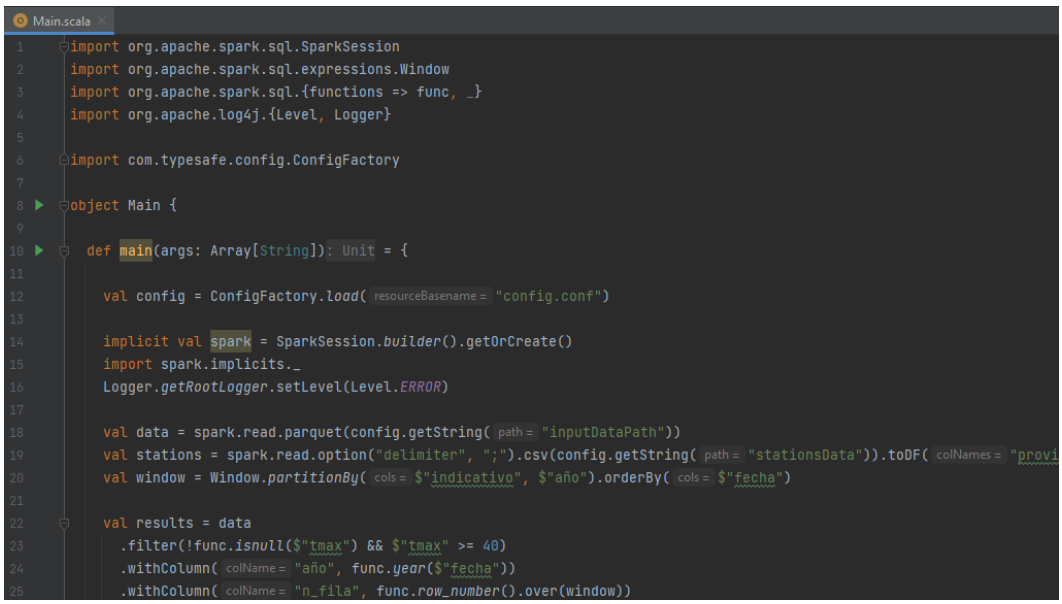
   - *inputDataPath*: se utiliza para especificar la ruta donde se encuentran los datos de entrada necesarios para realizar la consulta deseada.
   - *outputDataPath*: indica la ruta donde deseamos guardar los resultados correspondientes a la consulta realizada.

- *stationsData:* especifica la ubicación y nombre del archivo que contiene la información que relaciona el ID de la estación meteorológica con su provincia y ubicación geográfica.

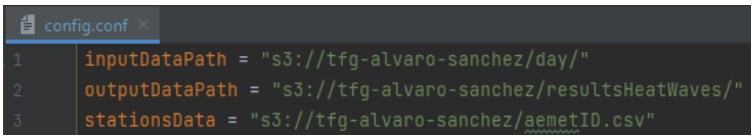
De esta manera, evitamos que, si algún otro usuario desea modificar estos parámetros de entrada, deba de acceder al código pudiendo realizar modificaciones no deseadas y que podrían causar problemas en el funcionamiento del programa.

Finalmente, en cuanto al programa, es necesario realizar algunas modificaciones en el archivo *build.sbt*. En este archivo deberemos especificar las librerías que vamos a utilizar para el programa, incluyendo el uso de Spark y la librería config, la cual nos permitirá utilizar el archivo de configuración mencionado anteriormente. Además, deberemos añadir una estrategia de fusión para cuando utilicemos el comando *assembly* de sbt, el cual se utilizará para crear el archivo JAR, lo cual se explicará con más detalle más adelante.

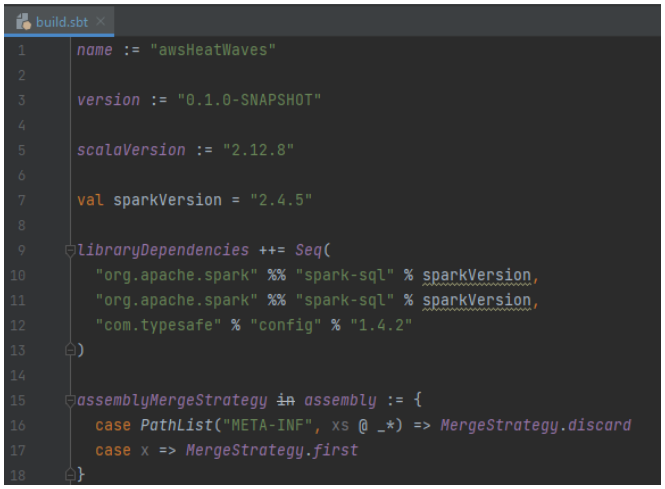
Como resultado de todo esto tendríamos lo siguiente. El fichero *Main.scala* con el código correspondiente a la consulta quedando de la siguiente manera:



El archivo *config.conf* que presentara un aspecto similar a este:

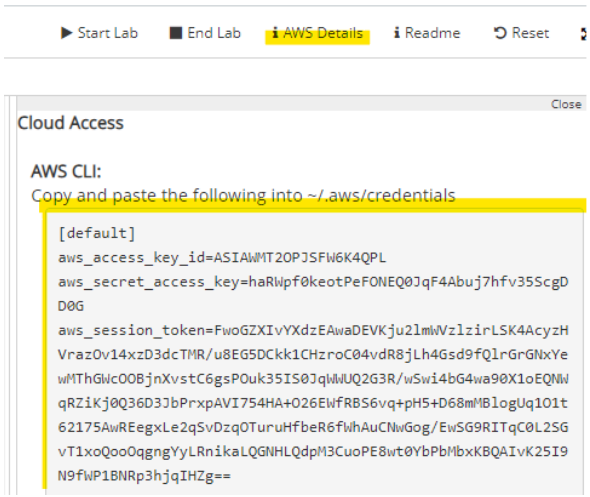


Y, por último, el fichero de configuración sbt, *build.sbt*, que presentará un aspecto similar al siguiente:



Una vez que tenemos todo preparado, crearemos el archivo con extensión .jar que se ejecutará en el clúster. Este ejecutable se genera ejecutando el comando *sbt clean* assembly. Podremos encontrar el archivo de aplicación en la carpeta del proyecto target/scala-<<versión>>, donde <<versión>> es la versión de Scala utilizada.

3. Una vez tenemos todos los recursos necesarios, necesitamos obtener las credenciales específicas de AWS. Estas credenciales nos permiten ejecutar comandos en la consola de AWS desde nuestro ordenador. A continuación, se describen los pasos detallados para hacerlo:
    - Inicializamos la consola AWS en página web de Amazon
    - Una vez dentro, buscaremos la sección llamada *AWS Details*, donde obtenemos las credenciales para la sesión



- Copiamos estas credenciales a nuestro fichero a *credentials*. Este se suele encontrar en Windows dentro de la carpeta C:\Users\<<nombreUsuario>>\.aws siendo el *nombreUsuario* el nombre del perfil que estemos utilizando.

4. Una vez tenemos configuradas las credenciales, deberemos de crear un contenedor en AWS S3 en el cual almacenaremos los datos de entrada, este contenedor se tratará como una carpeta en la nube. Para su creación debemos ejecutar el comando *aws s3 mb s3://nombre*, donde *nombre* es el nombre que se desea darle al contenedor, en este caso se ha creado con el nombre *tfg-alvaro-sanchez*. Para comprobar, que el contenedor se ha creado correctamente, se puede ejecutar el comando *aws s3 ls* el cual nos mostrará una lista de los contenedores existentes en la cuenta AWS.
5. A continuación, subiremos los archivos a nuestro contenedor. En este caso, estamos subiendo la información meteorológica de entrada en formato Parquet haciendo uso del comando *aws s3 cp --recursive ./dayParquet s3://tfg-alvaro-sanchez/day/*. Este comando copia todos los archivos de la carpeta local *dayParquet* en el contenedor *tfg-alvaro-sanchez/day/*, lo que crea una nueva carpeta en el contenedor y agregará todos los archivos dentro de ella. También subiremos el archivo *aemetID.csv* utilizando el comando *aws s3 cp ./aemetID.csv s3://tfg-alvaro-sanchez/* y el archivo .jar generando: *aws s3 cp target\scala-2.12\awsHeatWaves-assembly-0.1.0-SNAPSHOT.jar s3://tfg-alvaro-sanchez/jars/*.
6. Seguidamente deberemos de inicializar un clúster de EMR. Deberemos seleccionar la opción de crear clúster, en cuanto a la configuración de software elegiremos Spark, y por último indicaremos el número de instancias que deseamos utilizar.
7. Para finalizar, incluiremos la ejecución de la consulta almacenada en el archivo .jar como un paso adicional en el proceso del clúster. Para lograr esto, utilizaremos el siguiente script:

```
1  #!/bin/bash
2
3  CLUSTER_ID=$1
4  MAIN_JAR=s3://$2/jars/awsHeatWaves-assembly-0.1.0-SNAPSHOT.jar
5  PROFILE=default
6
7  aws emr add-steps \
8    --cluster-id $CLUSTER_ID \
9    --steps Type=Spark,Name="My program",ActionOnFailure=CONTINUE,Args=[--class,Main,$MAIN_JAR] \
10   --profile $PROFILE
```

En este script, será necesario modificar la línea 4, donde se especifica el nombre del archivo .jar que se desea ejecutar. Además, este script requiere dos parámetros de entrada. El primer parámetro es el ID del clúster donde deseamos realizar la ejecución y el segundo parámetro es el contenedor donde la aplicación debe buscar los datos necesarios para ejecutar la consulta. Para realizar la ejecución del script, se puede utilizar un comando similar a este *sh runpipeline.sh j-394197QJGELX4 tfg-alvaro-sanchez*.

2. Experimentos / validación

2.1. Consultas realizadas

2.2. Análisis de requisitos no funcionales

En cuanto al tiempo de ejecución de la consulta, se puede observar que se ha hecho uso de un total de 28 Cores para el clúster y de 4 en local, además al aumentar el número de ejecutores se producirá un aumento de memoria. Sin embargo, esto no debería suponer un problema ya que la consulta no requiere una gran cantidad de espacio, por lo que no se espera una diferencia significativa en el rendimiento. En consecuencia, se espera que los tiempos de ejecución sean bastante más rápidos en el clúster que en local debido al gran número de ejecutores.

A continuación, se detallan las características de los ejecutores en el clúster:

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Blacklisted
Active(8)	0	0.0 B / 36.8 GB	0.0 B	28	0	0	547	547	7.4 min (25 s)	38.3 MB	252.4 KB	252.4 KB	0
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	0
Total(8)	0	0.0 B / 36.8 GB	0.0 B	28	0	0	547	547	7.4 min (25 s)	38.3 MB	252.4 KB	252.4 KB	0

Y las características de los ejecutores locales:

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Blacklisted
Total(1)	0	159 KB / 455.5 MB	0.0 B	4	0	0	615	615	5.4 min (5 s)	98.7 MB	242.4 KB	242.4 KB	0
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	0
Active(1)	0	159 KB / 455.5 MB	0.0 B	4	0	0	615	615	5.4 min (5 s)	98.7 MB	242.4 KB	242.4 KB	0

Al evaluar el desempeño de la consulta en ambos entornos, se constata que el clúster proporciona mejores resultados. En particular, se observa una reducción significativa en el tiempo de ejecución, ya que en el clúster la consulta se completó en 15 segundos, mientras que en el entorno local tardó 1.1 minutos. Esto se debe a la capacidad del clúster presenta una mayor capacidad para ejecutar varias tareas en paralelo al disponer de un mayor número de ejecutores.

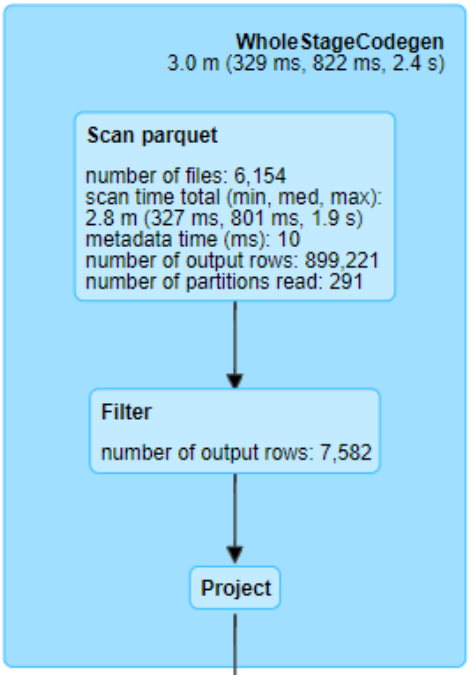
Ejecución en el clúster

Ejecución en local

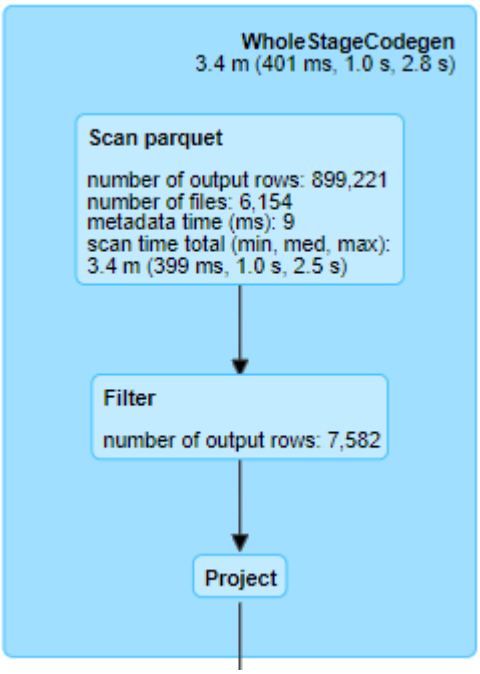
Pero podemos observar pequeñas diferencias al analizar el proceso interno de la consulta.

La primera tarea que se lleva a cabo al ejecutar la consulta es la lectura de los archivos de entrada. Además de llevarse a cabo la lectura, se realiza un filtrado de los datos, se muestra a continuación una comparación entre la ejecución en local y la ejecución en el clúster.

Ejecución en el clúster



Ejecución en local



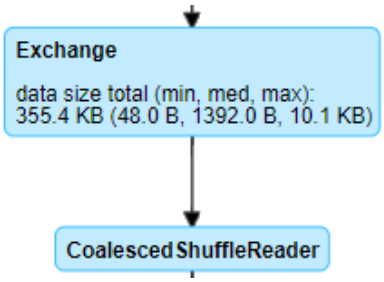
En este primer trabajo, podemos obtener información para el análisis del rendimiento de la consulta. Por ejemplo, los datos mostrados entre paréntesis al inicio del trabajo a la derecha, los cuales proporcionan información sobre el tiempo de ejecución de las tareas de una etapa, incluyendo el tiempo mínimo, medio y máximo de ejecución respectivamente. Estos tiempos pueden ayudarnos a identificar cuellos de botella en el rendimiento de la consulta. Además, como se mencionó anteriormente, en esta etapa encontramos dos tareas diferentes: la lectura de los archivos y el filtrado de estos. En el trabajo denominado "scan Parquet", se pueden encontrar información relevante como:

- *Number of files*: siendo el número de archivos leídos.
- *Scan time total*: nos indica el tiempo total que se ha tardado en leer los archivos por etapas, de la misma manera que antes siendo el primer valor el tiempo de lectura mínimo, el segundo el tiempo medio, y el tercero el tiempo máximo de lectura.
- *Metadata time*: se refiere al tiempo empleado en leer los metadatos de los archivos Parquet.
- *Number of output rows*: número de filas que tenemos una vez leídos todos los archivos.
- *Number of partitions read*: El número de particiones leídas durante la lectura.

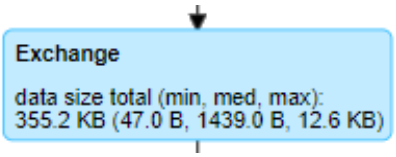
Se puede observar además que, una vez realizado el filtro en los datos, el número de filas disminuye considerablemente, por lo que resultara mucho más sencillo y eficiente su tratamiento.

Una vez que se ha completado la lectura y filtrado de los datos, el siguiente trabajo que se ejecuta en ambos entornos (clúster y local) es *Exchange*. Este trabajo es responsable de distribuir los datos entre los nodos ejecutores y organizarlos de manera que el nodo disponga de los datos necesarios para cada tarea.

Ejecución en el clúster



Ejecución en local

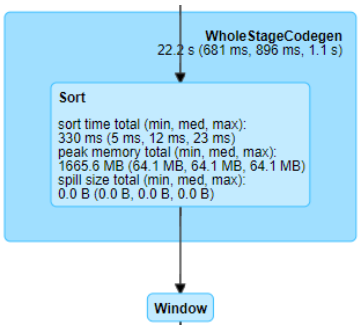


Los números presentados junto al *data size total* son medidas del tamaño de los datos intercambiados. En este caso, se presentan tres medidas: el tamaño mínimo de los datos intercambiados en todas las tareas, el tamaño medio, y el tamaño máximo respectivamente. Estos valores nos permiten entender el volumen de datos que se están moviendo y evaluar el rendimiento de la operación de intercambio de datos.

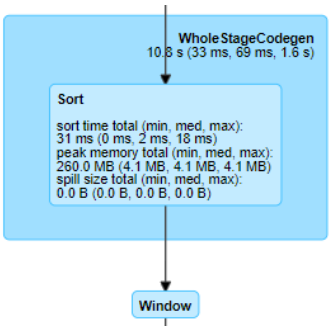
En el caso del clúster, además podemos observar la implementación de una optimización llamada CoalescedShuffleReader. Esta optimización se utiliza para mejorar el rendimiento de la operación de intercambio de datos. Se realiza agrupando bloques de datos en conjuntos más grandes antes de transferirlos entre los nodos ejecutores. Esto reduce el tráfico en la red entre los distintos nodos ejecutores y mejora el rendimiento de la consulta.

A continuación, Spark llevará a cabo un trabajo de ordenamiento, el cual es necesario para poder calcular la función *row_number()*, correspondiente al siguiente paso, ya que esta función asignará un número de fila a cada una sobre los resultados ordenados por la función ventana que habíamos especificado.

Ejecución en el clúster



Ejecución en local



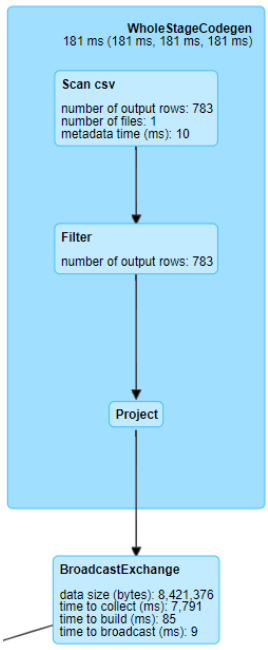
En la ejecución de este trabajo podemos encontrar la siguiente información:

- *Sort sort time total*: el cual nos proporciona información sobre el tiempo mínimo, medio y máximo que ha tardado cada tarea de ordenación.
- *Peak memory total*: nos indica el uso de memoria que se ha producido durante la ejecución de la tarea de ordenación, además de proporcionarnos información sobre el uso mínimo, medio y máximo de memoria en cada tarea.
- *spill size total*: nos indica el tamaño total de los datos que se escriben en disco temporalmente debido a que no caben en memoria. También, al igual que el resto nos proporciona información sobre el tamaño mínimo, medio y máximo de los datos escritos en memoria en cada tarea.

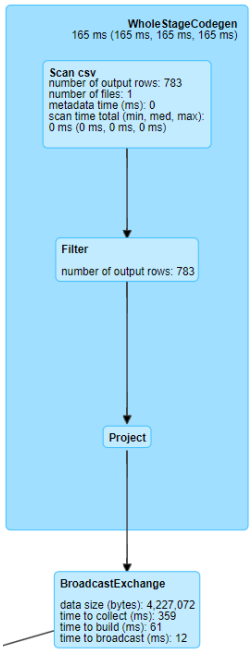
A continuación, se ejecutarán en paralelo varias operaciones. Una de ellas será la lectura del fichero, a través del cual obtendremos la relación entre las estaciones y su ubicación geográfica. Por otro lado, se continuará con la ejecución correspondiente a la consulta realizada.

Comenzaremos comentando la parte correspondiente a la lectura del fichero.

Ejecución en el clúster



Ejecución en local



En esta se realiza un trabajo denominado *Scan csv* el cual nos proporciona información como:

- *number of output rows*: El número de filas de salida una vez leída la información.
- *number of files*: Podremos saber el número de ficheros que se hen leído.
- *metadata time*: El tiempo necesitado para obtener los metadatos.

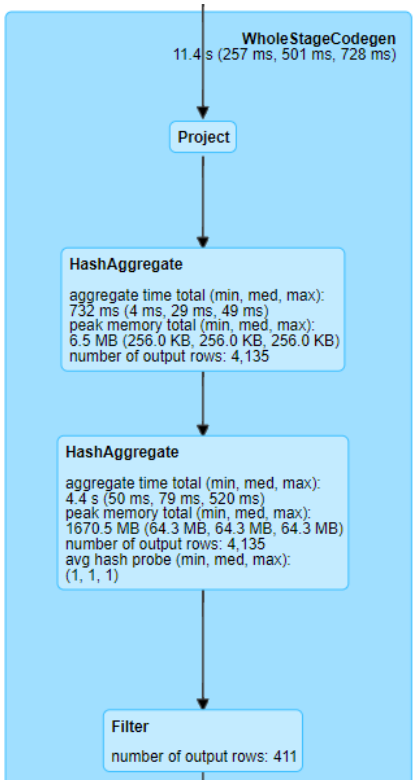
Una vez obtenida la información, se procede a realizar una operación de filtrado sobre los datos. Sin embargo, en este caso, como no se especificó ningún tipo de filtrado, se mantiene el número de filas originales.

Por último, se realiza una operación denominada *BroadcastExchange*. Esta es un mecanismo utilizado para mejorar el rendimiento de las consultas al reducir el tráfico en la red. En lugar de enviar los datos a través de la red para cada tarea, los datos se envían solo una sola vez almacenandose en memoria en cada nodo. Esta tarea proporciona información como:

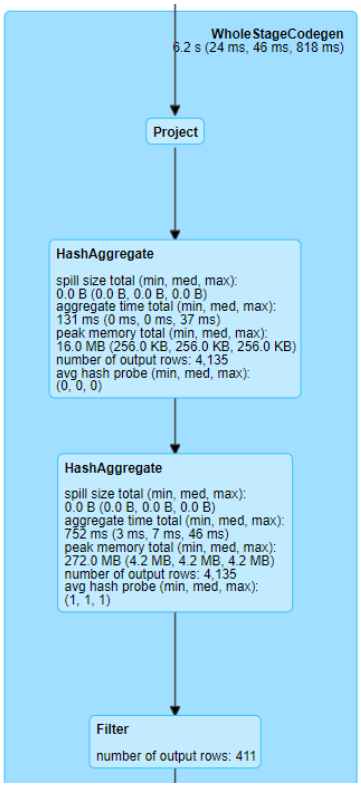
- *data size*: Indicándonos el tamaño de los datos que se están transfiriendo.
- *time to collect*: El tiempo necesario para recolectar todos los datos.
- *time to build*: Tiempo en construir la transmisión.
- *time to broadcast*: Tiempo necesario para realizar la transmisión de los datos.

Mientras se lleva a cabo la lectura del fichero, la consulta continúa con una etapa que resulta ser bastante extensa.

Ejecución en el clúster



Ejecución en local



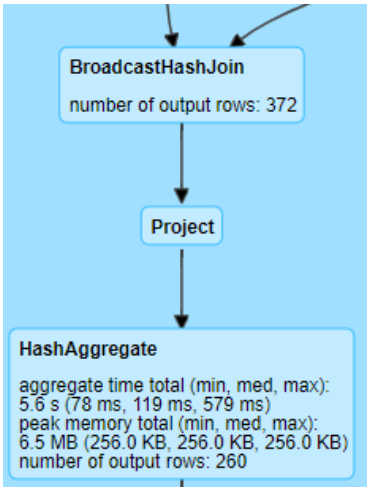
Los primeros pasos de la etapa se corresponden con dos tareas *HashAggregate*, las cuales son responsables de llevar a cabo las funciones de agrupación y agregación de los datos en la consulta. En la primera de ellas se realizara de manera parcial y en la segunda completa mejorando así el rendimiento del sistema.

- *Aggregate time total*: Se corresponde al tiempo total que tardó la tarea en completarse, con el tiempo mínimo, mediano y máximo que tardó cada una de las particiones de datos.
- *Peak memory total*: La cantidad de memoria utilizada por la tarea en su punto más alto, mostrando el uso mínimo, medio y máximo de memoria de cada una de las particiones de datos.
- *Number of output rows*: El número de filas en el DataFrame resultante después de la agrupación o agregación.
- *Avg hash probe*: Indica el promedio de veces que se tuvo que buscar un valor específico en una tabla hash.

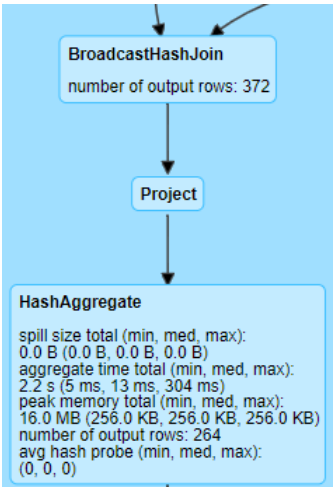
Por último, se realiza el filtro correspondiente a que el número de días de la ola de calor debe ser mayor a 3, quedándonos únicamente con 411 filas.

A continuación, se realiza la unión de los datos correspondientes a la consulta con los del fichero CSV que habíamos leído a la par.

Ejecución en el clúster



Ejecución en local

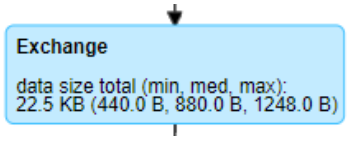


La primera de las tareas a ejecutar en esta sección es un *BroadcastHashJoin*, el cual se utiliza para unir dos conjuntos de datos en Spark, especialmente cuando uno de los conjuntos de datos es mucho más pequeño que el otro. Además, nos muestra información como el numero de filas resultantes una vez realizada la unión.

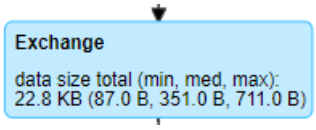
La última tarea por ejecutar en esta etapa es *HashAggregate*, la cual realiza la última agrupación parcial de los datos de la consulta antes de guardarlos.

A continuación, se produce una tarea *Exchange* la cual mide el tamaño total de los datos intercambiados entre los diferentes nodos del clúster.

Ejecución en el clúster



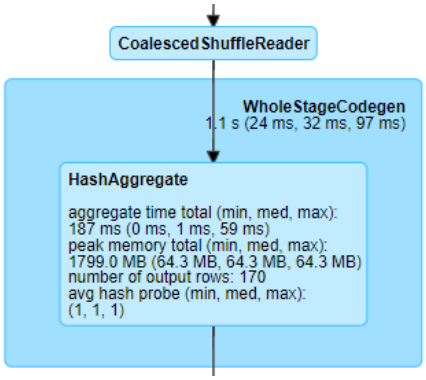
Ejecución en local



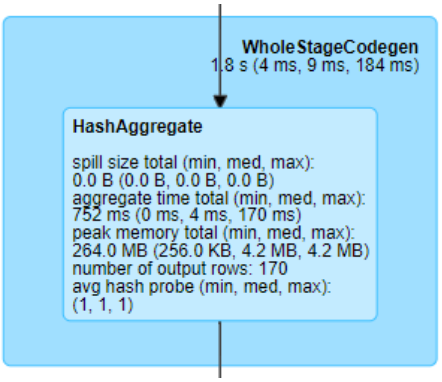
Se nos proporciona información valiosa acerca del intercambio de datos realizado durante el proceso. En particular, se nos informa acerca del tamaño total de los datos intercambiados y acerca del tamaño mínimo, medio y máximo de los paquetes de datos intercambiados.

Para finalizar, se realiza otra tarea de HashAggregate, la cual realiza la última agrupación total de los datos de la consulta antes de guardarlos

Ejecución en el clúster



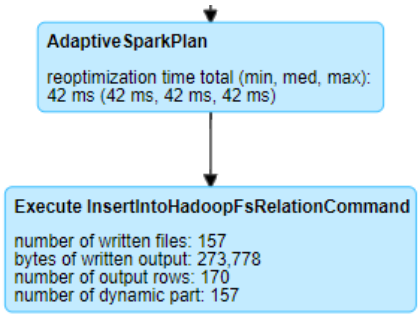
Ejecución en local



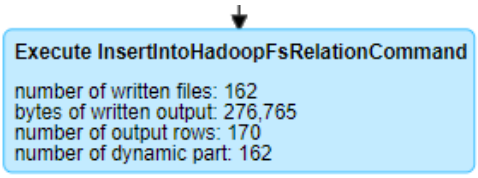
Como se puede observar en el clúster, se realiza una tarea denominada *CoalescedShuffleReader* antes de realizar la otra tarea. Esta tarea se encarga de optimizar el proceso de intercambio de datos entre nodos del clúster. En particular, se agrupan los datos que se van a enviar entre los nodos, con el objetivo de reducir el número de conexiones y la cantidad de datos transmitidos.

Para finalizar, se realiza una tarea de escritura de los resultados.

Ejecución en el clúster



Ejecución en local



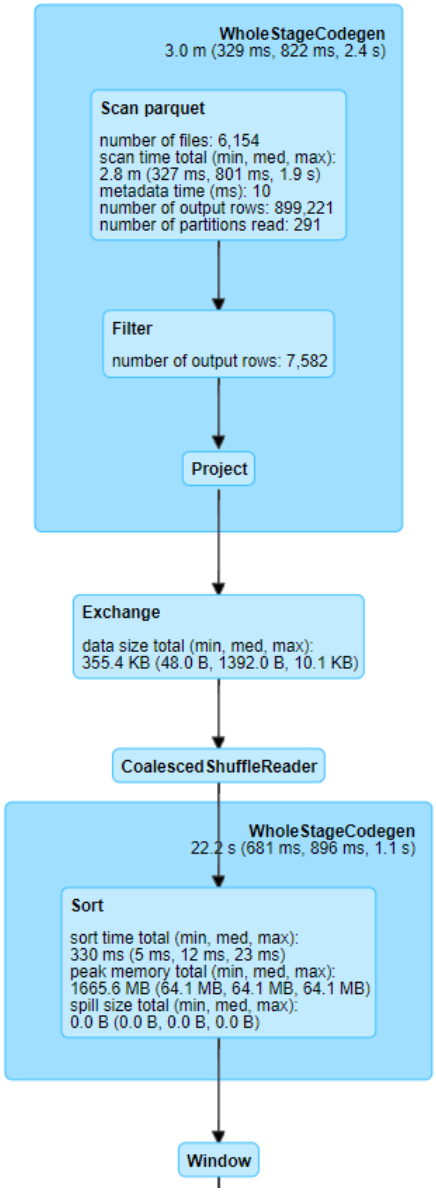
La tarea *Execute InsertIntoHadoopFsRelationCommand* se utiliza para insertar datos en una tabla en un sistema de archivos Hadoop. Nos muestra información como:

- *Number of written files*: Corresponde al número de archivos escritos.
- *Bytes of written output*: Tamaño total de bytes de salida.
- *Number of output rows*: Nos informa del número de filas escritas.
- *Number of dinamic part*: El número de particiones generadas al guardar los datos.

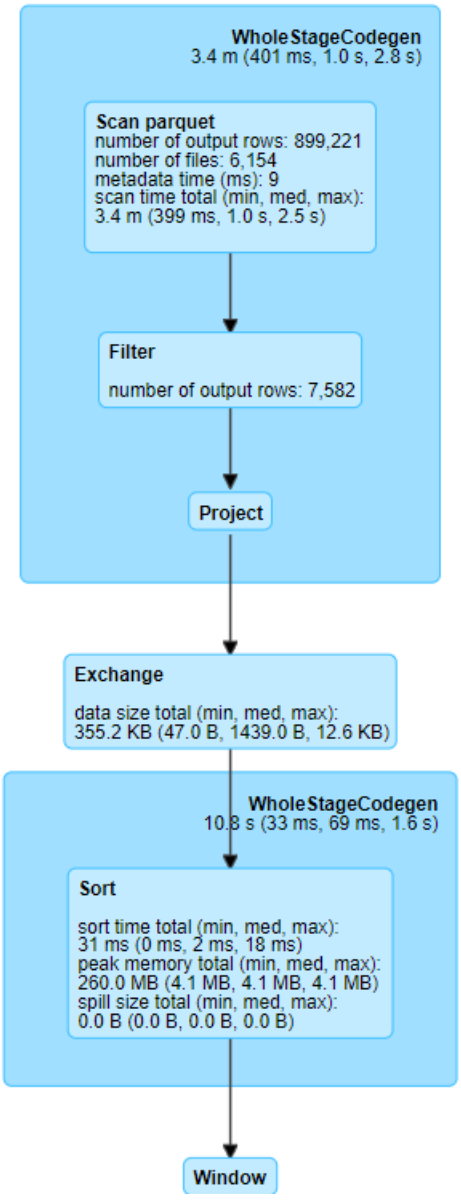
Además, en el clúster se ejecuta la tarea AdaptiveSparkPlan, la cual se refiere a la optimización de un plan de ejecución de Spark de manera dinámica. Esta tarea se ejecuta al final de la consulta para poder utilizar los datos y resultados de la ejecución anterior para optimizar el plan de ejecución de la siguiente consulta en caso de que la hubiera, lo que mejora el rendimiento y la eficiencia de la aplicación.

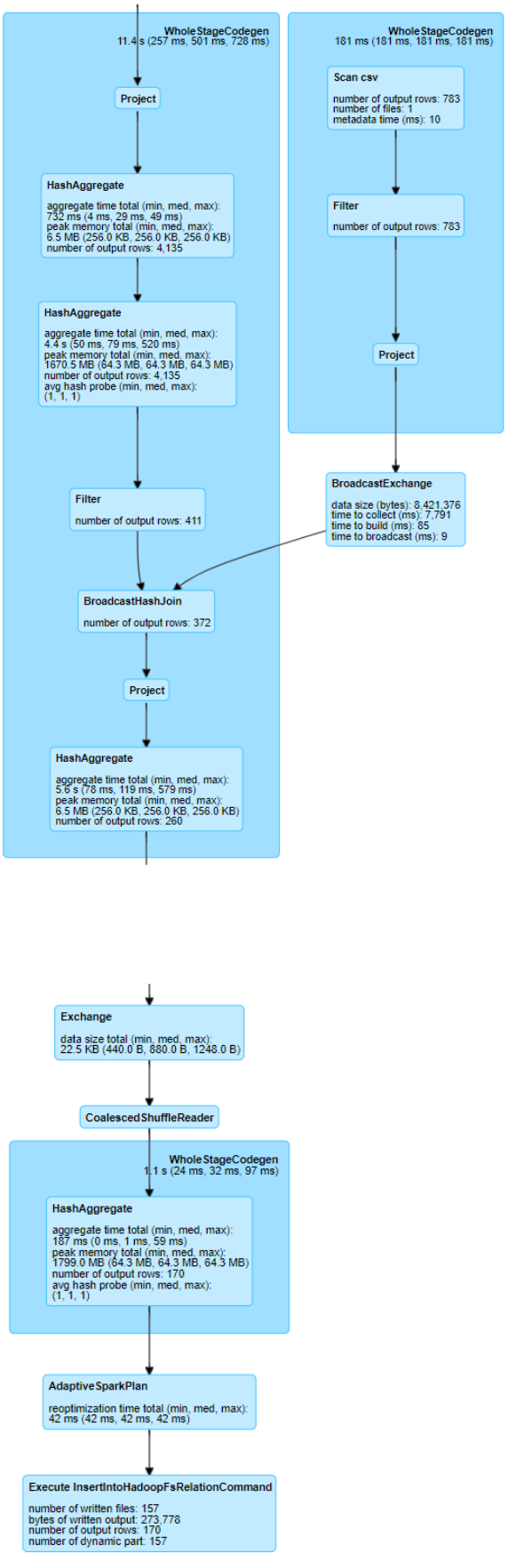
A continuación, se muestra el plan de consulta de manera completa:

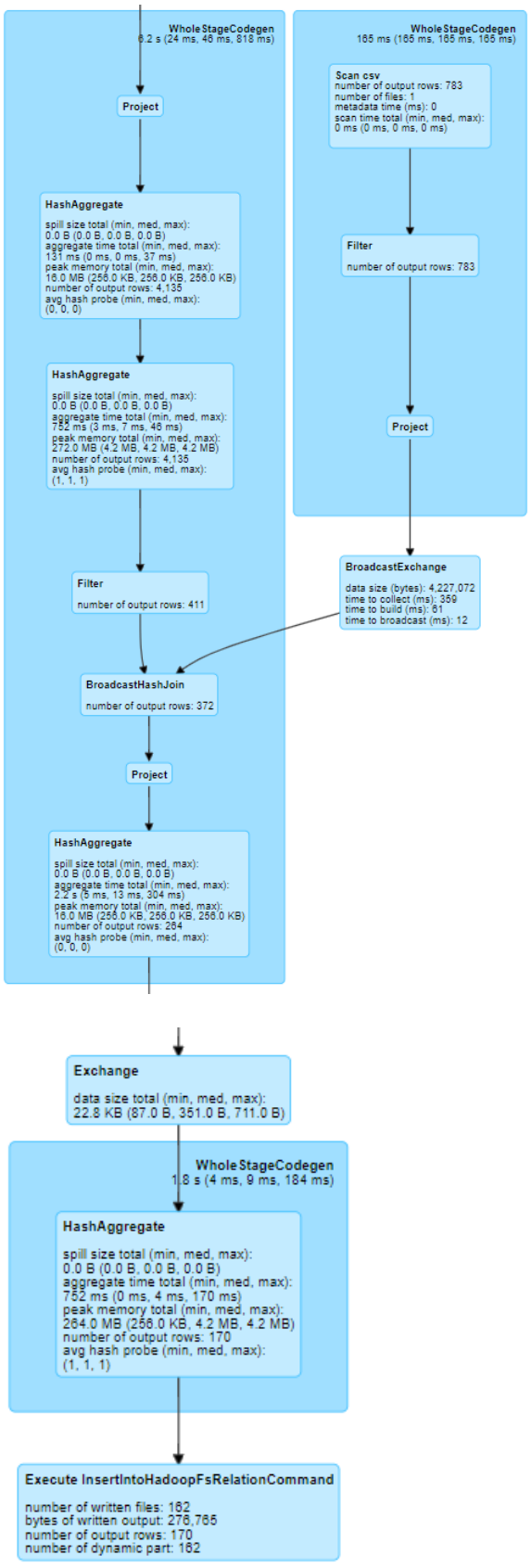
Ejecución en el clúster



Ejecución en local




WholeStageCodegen
11.4 s (257 ms, 501 ms, 728 ms)
Project
HashAggregate
aggregate time total (min, med, max):
732 ms (4 ms, 29 ms, 49 ms)
peak memory total (min, med, max):
6.5 MB (256.0 KB, 256.0 KB, 256.0 KB)
number of output rows: 4,135
HashAggregate
aggregate time total (min, med, max):
4.4 s (50 ms, 79 ms, 520 ms)
peak memory total (min, med, max):
1670.5 MB (64.3 MB, 64.3 MB, 64.3 MB)
number of output rows: 4,135
avg hash probe (min, med, max):
(1, 1, 1)
Filter
number of output rows: 411
BroadcastHashJoin
number of output rows: 372
Project
HashAggregate
aggregate time total (min, med, max):
5.6 s (78 ms, 119 ms, 579 ms)
peak memory total (min, med, max):
6.5 MB (256.0 KB, 256.0 KB, 256.0 KB)
number of output rows: 260
WholeStageCodegen
181 ms (181 ms, 181 ms, 181 ms)
Scan csv
number of output rows: 783
number of files: 1
metadata time (ms): 10
Filter
number of output rows: 783
Project
BroadcastExchange
data size (bytes): 8,421,376
time to collect (ms): 7,791
time to build (ms): 85
time to broadcast (ms): 9
Exchange
data size total (min, med, max):
22.5 KB (440.0 B, 880.0 B, 1248.0 B)
CoalescedShuffleReader
WholeStageCodegen
1.1 s (24 ms, 32 ms, 97 ms)
HashAggregate
aggregate time total (min, med, max):
187 ms (0 ms, 1 ms, 59 ms)
peak memory total (min, med, max):
1799.0 MB (64.3 MB, 64.3 MB, 64.3 MB)
number of output rows: 170
avg hash probe (min, med, max):
(1, 1, 1)
AdaptiveSparkPlan
reoptimization time total (min, med, max):
42 ms (42 ms, 42 ms, 42 ms)
Execute InsertIntoHadoopFsRelationCommand
number of written files: 157
bytes of written output: 273,778
number of output rows: 170
number of dynamic part: 157


WholeStageCodegen
6.2 s (24 ms, 48 ms, 818 ms)
Project
HashAggregate
spill size total (min, med, max):
0.0 B (0.0 B, 0.0 B, 0.0 B)
aggregate time total (min, med, max):
131 ms (0 ms, 0 ms, 37 ms)
peak memory total (min, med, max):
16.0 MB (256.0 KB, 256.0 KB, 256.0 KB)
number of output rows: 4,135
avg hash probe (min, med, max):
(0, 0, 0)
HashAggregate
spill size total (min, med, max):
0.0 B (0.0 B, 0.0 B, 0.0 B)
aggregate time total (min, med, max):
752 ms (3 ms, 7 ms, 48 ms)
peak memory total (min, med, max):
272.0 MB (4.2 MB, 4.2 MB, 4.2 MB)
number of output rows: 4,135
avg hash probe (min, med, max):
(1, 1, 1)
Filter
number of output rows: 411
BroadcastHashJoin
number of output rows: 372
Project
HashAggregate
spill size total (min, med, max):
0.0 B (0.0 B, 0.0 B, 0.0 B)
aggregate time total (min, med, max):
2.2 s (5 ms, 13 ms, 304 ms)
peak memory total (min, med, max):
16.0 MB (256.0 KB, 256.0 KB, 256.0 KB)
number of output rows: 264
avg hash probe (min, med, max):
(0, 0, 0)
WholeStageCodegen
165 ms (165 ms, 165 ms, 165 ms)
Scan csv
number of output rows: 783
number of files: 1
metadata time (ms): 0
scan time total (min, med, max):
0 ms (0 ms, 0 ms, 0 ms)
Filter
number of output rows: 783
Project
BroadcastExchange
data size (bytes): 4,227,072
time to collect (ms): 359
time to build (ms): 61
time to broadcast (ms): 12
Exchange
data size total (min, med, max):
22.8 KB (87.0 B, 351.0 B, 711.0 B)
WholeStageCodegen
1.8 s (4 ms, 9 ms, 184 ms)
HashAggregate
spill size total (min, med, max):
0.0 B (0.0 B, 0.0 B, 0.0 B)
aggregate time total (min, med, max):
752 ms (0 ms, 4 ms, 170 ms)
peak memory total (min, med, max):
264.0 MB (256.0 KB, 4.2 MB, 4.2 MB)
number of output rows: 170
avg hash probe (min, med, max):
(1, 1, 1)
Execute InsertIntoHadoopFsRelationCommand
number of written files: 162
bytes of written output: 276,765
number of output rows: 170
number of dynamic part: 162