

Análisis aerolínea

Práctica obligatoria

Antonio Cabrera

Trabajo para el doble grado de
Ingeniería del Software y Matemática Computacional



Asignatura de procesamiento de datos

U-tad

España

Mayo 2024

Contents

| | | |
|----------|---|-----------|
| 1 | Preparación del entorno de la máquina virtual | 4 |
| 1.1 | Preparación de Cassandra | 4 |
| 1.2 | Desplegar HDFS | 6 |
| 1.3 | Desplegar PostgreSQL | 7 |
| 1.4 | Desplegar Cassandra | 7 |
| 1.5 | Desplegar clúster de Spark Standalone | 8 |
| 1.6 | Desplegar una shell de Spark | 11 |
| 2 | Ingesta de los datos en el lago de datos | 14 |
| 2.1 | Countries | 14 |
| 2.2 | Airlines | 17 |
| 2.3 | Airports | 22 |
| 2.4 | Routes | 26 |
| 3 | Análisis de datos ingestados en el lago de datos | 30 |
| 3.1 | Consultas | 30 |
| 3.1.1 | Preparación de las consultas | 30 |
| 3.1.2 | Consulta 1 | 31 |
| 3.1.3 | Consulta 2 | 31 |
| 3.1.4 | Consulta 3 | 32 |
| 3.1.5 | Consulta 4 | 32 |
| 3.1.6 | Consulta 5 | 33 |
| 3.1.7 | Consulta 6 | 33 |
| 3.2 | Persistencia de datos agregados | 34 |

List of Figures

| | | |
|------|---|----|
| 1.1 | Actualización del sistema operativo | 4 |
| 1.2 | Instalación de Python 2 | 5 |
| 1.3 | Descarga de Cassandra | 5 |
| 1.4 | Descompresión de Cassandra | 5 |
| 1.5 | Eliminación del archivo .tar.gz | 6 |
| 1.6 | Despliegue de HDFS | 6 |
| 1.7 | Comprobación de HDFS | 6 |
| 1.8 | Comando de HDFS | 6 |
| 1.9 | Comprobación de Postgres | 7 |
| 1.10 | Comando de prueba de Postgres | 7 |
| 1.11 | Despliegue de Cassandra | 8 |
| 1.12 | Consola de Cassandra | 8 |
| 1.13 | Creación de keyspace en Cassandra | 8 |
| 1.14 | Cambio de directorio a Spark | 8 |
| 1.15 | Arranque del Master de Spark | 9 |
| 1.16 | URL del Master de Spark | 9 |
| 1.17 | Interfaz web del Master de Spark | 9 |
| 1.18 | URL de los Workers de Spark | 10 |
| 1.19 | Configuración de los Workers de Spark | 10 |
| 1.20 | Arranque de los Workers de Spark | 10 |
| 1.21 | Interfaz web del Master de Spark con los Workers conectados | 11 |
| 1.22 | Descarga de los conectores de Postgres y Cassandra | 11 |
| 1.23 | Arranque de la shell de Spark | 12 |
| 1.24 | Interfaz web del Master de Spark con la aplicación creada | 12 |
| 1.25 | Interfaz web de la shell de Spark | 13 |
| 2.1 | RDD de los países | 15 |
| 2.2 | RDD de los países limpio | 15 |
| 2.3 | RDD de los países con nombres de columnas | 15 |
| 2.4 | Dataframe de los países | 16 |
| 2.5 | Iniciando HDFS | 16 |
| 2.6 | Guardando el dataframe en HDFS | 16 |
| 2.7 | Contenido de la carpeta countries en HDFS | 17 |
| 2.8 | 5 filas de los datos almacenados en HDFS | 17 |
| 2.9 | Importando estructuras de datos | 18 |

| | | |
|------|---|----|
| 2.10 | Creando el esquema de los datos de las aerolíneas | 18 |
| 2.11 | Dataframe de las aerolíneas | 19 |
| 2.12 | Dataframe de las aerolíneas con la columna active convertida a boolean | 19 |
| 2.13 | Dataframe de los países | 20 |
| 2.14 | Dataframe de las aerolíneas con los países | 20 |
| 2.15 | Dataframe de las aerolíneas con las columnas reordenadas | 21 |
| 2.16 | Dataframe de las aerolíneas sin valores nulos en la columna country | 21 |
| 2.17 | 5 filas de los datos de las aerolíneas en HDFS | 22 |
| 2.18 | Creando el esquema de los datos de los aeropuertos | 23 |
| 2.19 | Dataframe de los aeropuertos | 23 |
| 2.20 | Creando la UDF feetToMeters | 23 |
| 2.21 | Dataframe de los aeropuertos con la columna altitud convertida a metros | 24 |
| 2.22 | Entrando en la consola de Postgres | 24 |
| 2.23 | Creando la tabla airports en Postgres | 25 |
| 2.24 | Dando una contraseña al usuario postgres | 25 |
| 2.25 | Guardando los datos en la tabla airports de Postgres | 26 |
| 2.26 | 5 filas de la tabla airports de Postgres | 26 |
| 2.27 | Creando el esquema de los datos de las rutas | 27 |
| 2.28 | Dataframe de las rutas | 27 |
| 2.29 | Dataframe de las rutas con la columna route_id | 28 |
| 2.30 | Entrando en la shell de Cassandra | 28 |
| 2.31 | Creando la tabla routes en Cassandra | 29 |
| 2.32 | Guardando los datos en la tabla routes de Cassandra | 29 |
| 2.33 | 5 filas de los datos en la tabla routes de Cassandra | 29 |
| 3.1 | Creación de las vistas temporales de los dataframes | 31 |
| 3.2 | Resultado de la consulta de aeropuertos con mayor altitud | 31 |
| 3.3 | Resultado de la consulta de aeropuertos en España | 32 |
| 3.4 | Resultado de la consulta de países con aeropuertos en horario de verano de Europa | 32 |
| 3.5 | Resultado de la consulta de aerolíneas en EEUU con particionado por país | 33 |
| 3.6 | Resultado de la consulta de aerolíneas en EEUU | 33 |
| 3.7 | Resultado de la consulta de países con más aerolíneas inactivas | 33 |
| 3.8 | Resultado de la consulta de países con aerolíneas activas y aeropuertos con latitud mayor a 80 | 34 |
| 3.9 | Resultado de la consulta de rutas sin paradas a destinos con altitud mayor a 200 metros por país con aerolíneas inactivas | 34 |
| 3.10 | Guardado del dataframe en formato parquet en la ruta /práctica/aggregations/ de HDFS | 35 |
| 3.11 | Resultado de la consulta de la tabla de agregaciones | 35 |

Chapter 1

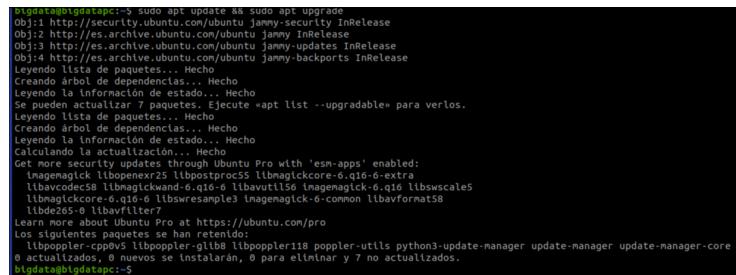
Preparación del entorno de la máquina virtual

Partiremos de la máquina virtual proporcionada por el profesor, la cual tiene instalado el sistema operativo Ubuntu 22.04.3 LTS.

1.1 Preparación de Cassandra

Primero instalaremos Cassandra, para ello primero actualizaremos el sistema operativo. El comando `sudo apt update` actualiza la lista de paquetes disponibles y sus versiones, mientras que el comando `sudo apt upgrade` instala las actualizaciones disponibles.

```
1 sudo apt update && sudo apt upgrade
```



```
[bigdata@bigdatapc:~] $ sudo apt update && sudo apt upgrade
[sudo] password for bigdata: 
Get:1 http://security.ubuntu.com/ubuntu jammy-security InRelease
Get:2 http://archive.ubuntu.com/ubuntu jammy InRelease
Get:3 http://es.archive.ubuntu.com/ubuntu jammy-updates InRelease
Get:4 http://es.archive.ubuntu.com/ubuntu jammy-backports InRelease
Leyendo lista de paquetes... Hecho
Creando árbol de dependencias... Hecho
Leyendo la información de estado... Hecho
Se han actualizado 7 paquetes. Ejecute «apt list --upgradable» para verlos.
Leyendo lista de paquetes... Hecho
Creando árbol de dependencias... Hecho
Leyendo la información de estado... Hecho
Calculando la actualización... Hecho
Get: more security updates through Ubuntu Pro with 'esm-apps' enabled;
Get:10 more security updates through Ubuntu Pro with 'esm-apps' enabled;
Get: libavcodec58 libmagickwand-6.q16-6 libavutil156 libopenmagick-6.q16 libswscale58
Get: libmagickcore-6.q16-6 libwsresample3 imagemagick-6-common libavformat58
Get: libde265-0 libavfilter7
Learn more about Ubuntu Pro at https://ubuntu.com/pro
Los siguientes paquetes se han retenido:
libpoppler3ppa1 libpoppler-glib6 libpoppler198 poppler-utils python3-update-manager update-manager update-manager-core
0 actualizados, 0 nuevos se instalarán, 0 para eliminar y 7 no actualizados.
[bigdata@bigdatapc:~]
```

Figure 1.1: Actualización del sistema operativo

Ahora instalaremos Python 2, ya que Cassandra requiere esta versión de Python. Para ello, ejecutamos el siguiente comando:

```
1 sudo apt install python2
```

```
bigdata@bigdatapc:~$ sudo apt install python2
Leyendo lista de paquetes... Hecho
Creando árbol de dependencias... Hecho
Leyendo la información de estado... Hecho
python2 ya está en su versión más reciente (2.7.18-3).
0 actualizados, 0 nuevos se instalarán, 0 para eliminar y 7 no actualizados.
bigdata@bigdatapc:~$
```

Figure 1.2: Instalación de Python 2

Después, descargaremos el archivo .tar.gz de Cassandra desde la página oficial de Apache. Para ello, ejecutamos el siguiente comando:

```
1 wget https://dlcdn.apache.org/cassandra/3.11.16/apache-cassandra
-3.11.16-bin.tar.gz
```

```
bigdata@bigdatapc:~$ wget https://dlcdn.apache.org/cassandra/3.11.16/apache-cassandra-3.11.16-bin.tar.gz
--2024-05-10 20:03:05 - https://dlcdn.apache.org/cassandra/3.11.16/apache-cassandra-3.11.16-bin.tar.gz
Resolviendo dlcndn.apache.org (dlcdn.apache.org)... 151.101.2.132, 2004:4e42:5644
Conectando con dlcndn.apache.org (dlcdn.apache.org)[151.101.2.132]:443... conectado.
Petición HTTP enviada, esperando respuesta... 200 OK
Longitud: 31111361 (30M) [application/x-gzip]
Guardando como: 'apache-cassandra-3.11.16-bin.tar.gz.1'
apache-cassandra-3.11.16-bin.tar.gz 100%[=====] 29,67M 9,33MB/s   en 3,2s
2024-05-10 20:03:08 (9,33 MB/s) - 'apache-cassandra-3.11.16-bin.tar.gz.1' guardado [31111361/31111361]
bigdata@bigdatapc:~$
```

Figure 1.3: Descarga de Cassandra

Descomprimimos el archivo .tar.gz con el siguiente comando:

```
1 tar -xvzf apache-cassandra-3.11.16-bin.tar.gz
```

```
bigdata@bigdatapc:~$ tar -zxfv apache-cassandra-3.11.16-bin.tar.gz
apache-cassandra-3.11.16/bin/
apache-cassandra-3.11.16/conf/
apache-cassandra-3.11.16/conf/triggers/
apache-cassandra-3.11.16/doc/
apache-cassandra-3.11.16/doc/cql3/
apache-cassandra-3.11.16/interface/
apache-cassandra-3.11.16/lib/
apache-cassandra-3.11.16/lib/sigar-bin/
apache-cassandra-3.11.16/pylib/
apache-cassandra-3.11.16/pylib/cqlshlib/
apache-cassandra-3.11.16/pylib/cqlshlib/test/
apache-cassandra-3.11.16/pylib/cqlshlib/test/config/
apache-cassandra-3.11.16/tools/
apache-cassandra-3.11.16/tools/bin/
apache-cassandra-3.11.16/tools/lib/
apache-cassandra-3.11.16/CASSANDRA-14092.txt
apache-cassandra-3.11.16/CHANGES.txt
apache-cassandra-3.11.16/LICENSE.txt
apache-cassandra-3.11.16/NEWS.txt
```

Figure 1.4: Descompresión de Cassandra

Por último, eliminamos el archivo .tar.gz con el siguiente comando:

```
1 rm apache-cassandra-3.11.16-bin.tar.gz
```

```
bigdata@bigdatapc:~$ rm apache-cassandra-3.11.16-bin.tar.gz  
bigdata@bigdatapc:~$
```

Figure 1.5: Eliminación del archivo .tar.gz

1.2 Desplegar HDFS

HDFS ya está instalado por defecto en la máquina virtual en la carpeta /hadoop-3.3.6. Para desplegar HDFS ejecutamos el siguiente comando:

```
1 ~/hadoop-3.3.6/sbin/start-dfs.sh
```

```
bigdata@bigdatapc:~$ ./hadoop-3.3.6/sbin/start-dfs.sh  
Starting namenodes on [localhost]  
Starting datanodes  
Starting secondary namenodes [bigdatapc]
```

Figure 1.6: Despliegue de HDFS

Ahora para comprobar que HDFS se ha desplegado correctamente, ejecutamos el siguiente comando:

```
1 jps
```

```
bigdata@bigdatapc:~$ jps  
81604 DataNode  
82038 Jps  
81834 SecondaryNameNode  
81466 NameNode  
bigdata@bigdatapc:~$
```

Figure 1.7: Comprobación de HDFS

Ahora ya podemos ejecutar comandos de HDFS, como por ejemplo el siguiente comando que muestra los archivos en el sistema de archivos HDFS:

```
1 ~/hadoop-3.3.6/bin/hdfs dfs -ls /
```

```
bigdata@bigdatapc:~$ ~/hadoop-3.3.6/bin/hdfs dfs -ls /  
Found 1 items  
drwx-wx-wx  - bigdata supergroup          0 2024-01-29 01:29 /tmp
```

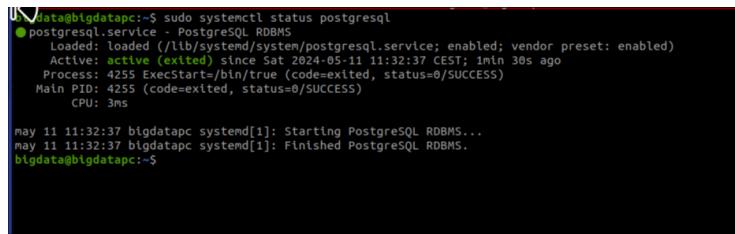
Figure 1.8: Comando de HDFS

1.3 Desplegar PostgreSQL

Postgres también está instalado por defecto en la máquina virtual, además se arranca por defecto al iniciar la sesión en la máquina virtual. El motivo por el que arranca por defecto es que se ha configurado como un servicio de systemd, por lo que se inicia automáticamente al arrancar el sistema.

Para comprobar que Postgres se ha desplegado correctamente, ejecutamos el siguiente comando:

```
1 sudo systemctl status postgresql
```



```
[bigdata@bigdatapc:~]$ sudo systemctl status postgresql
● postgresql.service - PostgreSQL RDBMS
   Loaded: loaded (/lib/systemd/system/postgresql.service; enabled; vendor preset: enabled)
   Active: active (exited) since Sat 2024-05-11 11:32:37 CEST; 1min 30s ago
     Process: 4255 ExecStart=/bin/true (code=exited, status=0/SUCCESS)
   Main PID: 4255 (code=exited, status=0/SUCCESS)
      CPU: 3ms

may 11 11:32:37 bigdatapc systemd[1]: Starting PostgreSQL RDBMS...
may 11 11:32:37 bigdatapc systemd[1]: Finished PostgreSQL RDBMS.
[bigdata@bigdatapc:~]$
```

Figure 1.9: Comprobación de Postgres

Ahora para asegurarnos de que todo funciona correctamente, nos conectamos a la consola de Postgres y ejecutamos un comando de prueba. Para ello, ejecutamos el siguiente comando:

```
1 sudo -u postgres psql
```

```
1 SELECT version();
```



```
[bigdata@bigdatapc:~]$ sudo -u postgres psql
psql: could not change directory to "/home/bigdata": Permission denied
psql (14.11 (Ubuntu 14.11-0ubuntu0.22.04.1))
Type "help" for help.

postgres=# SELECT version();
              version
-----
 PostgreSQL 14.11 (Ubuntu 14.11-0ubuntu0.22.04.1) on x86_64-pc-linux-gnu, compiled by gcc (Ubuntu 11.4.0-1ubuntu1-22.04) 11.4.0, 64-bit
(1 row)

postgres=#
```

Figure 1.10: Comando de prueba de Postgres

1.4 Desplegar Cassandra

Primero nos moveremos a la carpeta de Cassandra con el siguiente comando:

```
1 cd ~/apache-cassandra-3.11.16
```

Acto seguido, arrancamos el servicio de Cassandra con el siguiente comando:

```
1 bin/cassandra
```

```

bigdata@bigdatapc:~$ cd ~/apache-cassandra-3.11.16/bin/cassandra
bigdata@bigdatapc:~/apache-cassandra-3.11.16 OpenJDK 64-Bit Server VM warning: Cannot open file bin/../logs/gc.log due to No such file or
directory
CompilerOracle: dontinline org/apache/cassandra/db/columns$Serializer.deserializeLargeSubset (Lorg/apache/cassandra/io/util/DataInputPlus;
I)org/apache/cassandra/db/columns$1)Lorg/apache/cassandra/db/columns;
CompilerOracle: dontinline org/apache/cassandra/db/columns$Serializer.serializeLargeSubset (Ljava/util/Collection;I)org/apache/cassandra/
db/columns$1)org/apache/cassandra/io/util/DataOutputPlus)V
CompilerOracle: dontinline org/apache/cassandra/db/columns$Serializer.serializeLargeSubsetSize (Ljava/util/Collection;I)org/apache/cassandra/
db/columns$1)
CompilerOracle: dontinline org/apache/cassandra/db/commitlog/AbstractCommitLogSegmentManager.advanceAllocatingFrom (Lorg/apache/cassandra/
db/commitlog/AbstractCommitLogSegmentManager;Lorg/apache/cassandra/db/commitlog/CommitLogSegment;Lorg/apache/cassandra/db/commitlog/CommitLogSegment$1)V

```

Figure 1.11: Despliegue de Cassandra

Ahora inciamos la consola de Cassandra con el siguiente comando:

```
1 bin/cqlsh
```

Figure 1.12: Consola de Cassandra

Por último, tendremos que generar un keyspace que nos servirá más adelante.

```
1 CREATE KEYSPACE IF NOT EXISTS practica WITH REPLICATION = {'class':
'SimpleStrategy', 'replication_factor': 1};
```

Salimos de la consola de Cassandra con el siguiente comando:

```
1 exit
```

Figure 1.13: Creación de keyspace en Cassandra

1.5 Desplegar clúster de Spark Standalone

Vamos a ver como configurar y arrancar un despliegue de 1 nodo Master y 2 nodos Worker de Spark Standalone.

Primero, nos movemos a la carpeta de Spark con el siguiente comando:

```
1 cd ~/spark-3.3.3-bin-hadoop3
```

Figure 1.14: Cambio de directorio a Spark

Una vez que estamos en la carpeta de Spark, arrancamos el Master con el siguiente comando:

```
1 sbin/start-master.sh
```

```
bigdata@bigdatapc:~/spark-3.3.3-bin-hadoop3$ sbin/start-master.sh
starting org.apache.spark.deploy.master.Master, logging to /home/bigdata/spark-3.3.3-bin-hadoop3/logs/spark-bigdata-org.apache.spark.depl
oy.master.Master-1-bigdatapc.out
bigdata@bigdatapc:~/spark-3.3.3-bin-hadoop3$
```

Figure 1.15: Arranque del Master de Spark

En la ejecución del comando anterior, se nos muestra el archivo de los logs del Master, en este caso el archivo es `/home/bigdata/spark-3.3.3-bin-hadoop3/logs/spark-bigdata-org.apache.spark.deploy.master.Master-1-bigdatapc.out`. Con un par de comandos sacaremos la URL del Master, que es la dirección que usaremos para conectarnos a la interfaz web del Master.

```
1 cat /home/bigdata/spark-3.3.3-bin-hadoop3/logs/spark-bigdata-org.
apache.spark.deploy.master.Master-1-bigdatapc.out | grep 'http://'
| awk '{print $NF}'
```

```
bigdata@bigdatapc:~/spark-3.3.3-bin-hadoop3$ cat /home/bigdata/spark-3.3.3-bin-hadoop3/logs/spark-bigdata-org.apache.spark.deploy.master.
Master-1-bigdatapc.out | grep 'http://' | awk '{print $NF}'
http://10.0.2.15:8080
bigdata@bigdatapc:~/spark-3.3.3-bin-hadoop3$
```

Figure 1.16: URL del Master de Spark

En nuestro caso, si nos conectamos a la URL `http://10.0.2.15:8080/` podremos ver la interfaz web del Master de Spark.

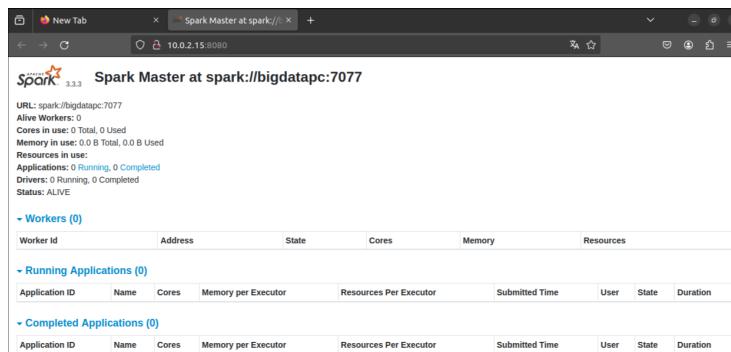


Figure 1.17: Interfaz web del Master de Spark

Para los workers, también necesitaremos una URL que encontraremos en los logs del Master. Para ello, ejecutamos el siguiente comando:

```
1 cat /home/bigdata/spark-3.3.3-bin-hadoop3/logs/spark-bigdata-org.
apache.spark.deploy.master.Master-1-bigdatapc.out | grep 'spark://'
| awk '{print $NF}'
```

```
bigdata@bigdatapc:~/spark-3.3.3-bin-hadoop3$ cat /home/bigdata/spark-3.3.3-bin-hadoop3/logs/spark-bigdata-org.apache.spark.deploy.master.* | grep 'spark://' | awk '{print $NF}'
spark://bigdatapc:7077
bigdata@bigdatapc:~/spark-3.3.3-bin-hadoop3$
```

Figure 1.18: URL de los Workers de Spark

Antes de desplegar los Workres, para poder tener dos Workers en una misma máquina vamos a modificar la configuración del archivo `conf/spark-env.sh`. Para ello, ejecutamos el siguiente comando:

```
1 vim conf/spark-env.sh
```

Y añadimos las 3 siguientes líneas al final del archivo:

```
1 SPARK_WORKER_INSTANCES=2
2 SPARK_WORKER_CORES=2
3 SPARK_WORKER_MEMORY=1g
```

```
SPARK_WORKER_CORES=2
SPARK_WORKER_MEMORY=1g
SPARK_WORKER_INSTANCES=2
"conf/spark-env.sh.template" 81L, 4576B escritos
```

Figure 1.19: Configuración de los Workers de Spark

Ahora arrancaremos dos Workers con 1GB de memoria y 2 cores (la configuración que hemos especificado). Es necesario especificar la memoria y los cores ya que por defecto los Workers usarán toda la memoria y todos los cores disponibles.

```
1 sbin/start-slave.sh spark://bigdatapc:7077
```

```
bigdata@bigdatapc:~/spark-3.3.3-bin-hadoop3$ sbin/start-slave.sh spark://bigdatapc:7077
This script is deprecated, use start-worker.sh
starting org.apache.spark.deploy.worker.Worker, logging to /home/bigdata/spark-3.3.3-bin-hadoop3/logs/spark-bigdata-org.apache.spark.depl
oy.worker.worker-2-bigdatapc.out
starting org.apache.spark.deploy.worker.Worker, logging to /home/bigdata/spark-3.3.3-bin-hadoop3/logs/spark-bigdata-org.apache.spark.depl
oy.worker.worker-3-bigdatapc.out
starting org.apache.spark.deploy.worker.Worker, logging to /home/bigdata/spark-3.3.3-bin-hadoop3/logs/spark-bigdata-org.apache.spark.depl
oy.worker.worker-4-bigdatapc.out
bigdata@bigdatapc:~/spark-3.3.3-bin-hadoop3$
```

Figure 1.20: Arranque de los Workers de Spark

Si nos vamos a la interfaz web del Master de Spark, podremos ver los Workers conectados. En esta interfaz se nos muestra el id del nodo Worker, la dirección IP, el número de cores, la memoria disponible, la carga de trabajo, la memoria utilizada y el estado del nodo. Además, arriba se nos muestra un resumen de todos los recursos usados y de las aplicaciones en ejecución.

The screenshot shows the Spark Master web interface at `spark://bigdatapc:7077`. It displays the following information:

- URL:** `spark://bigdatapc:7077`
- Alive Workers:** 2
- Cores in use:** 4 Total, 0 Used
- Memory in use:** 2.0 GiB Total, 0.0 B Used
- Resources in use:** 0
- Applications:** 0 Running, 0 Completed
- Drivers:** 0 Running, 0 Completed
- Status:** ALIVE

Workers (2)

| Worker ID | Address | State | Cores | Memory | Resources |
|---------------------------------------|-----------------|-------|------------|-------------------------|-----------|
| worker-20240511124347-10-0.2.15-42451 | 10.0.2.15:42451 | ALIVE | 2 (0 Used) | 1024.0 MiB (0.0 B Used) | |
| worker-20240511124350-10-0.2.15-33493 | 10.0.2.15:33493 | ALIVE | 2 (0 Used) | 1024.0 MiB (0.0 B Used) | |

Running Applications (0)

| Application ID | Name | Cores | Memory per Executor | Resources Per Executor | Submitted Time | User | State | Duration |
|----------------|------|-------|---------------------|------------------------|----------------|------|-------|----------|
|----------------|------|-------|---------------------|------------------------|----------------|------|-------|----------|

Completed Applications (0)

| Application ID | Name | Cores | Memory per Executor | Resources Per Executor | Submitted Time | User | State | Duration |
|----------------|------|-------|---------------------|------------------------|----------------|------|-------|----------|
|----------------|------|-------|---------------------|------------------------|----------------|------|-------|----------|

Figure 1.21: Interfaz web del Master de Spark con los Workers conectados

1.6 Desplegar una shell de Spark

El primer paso será descargar los conectores de Postgres y Cassandra. Primero nos moveremos a la carpeta `spark-3.3.3-bin-hadoop3/jars` y a continuación descargaremos los conectores con los siguientes comandos:

```
1 cd ~/spark-3.3.3-bin-hadoop3/jars
2 wget https://jdbc.postgresql.org/download/postgresql-42.7.3.jar
3 wget https://repo1.maven.org/maven2/com/datastax/spark/spark-
cassandra-connector_2.12/3.3.0/spark-cassandra-connector_2
.12-3.3.0.jar
```

```
[bigdata@bigdatapc:~/sparklines]$ cd ~/spark-3.3.3-bin-hadoop3/jars
[bigdata@bigdatapc:~/sparklines]$ wget https://jdbc.postgresql.org/download/postgresql-42.7.3.jar
[bigdata@bigdatapc:~/sparklines]$ wget https://repo1.maven.org/maven2/com/datastax/spark/spark-
cassandra-connector_2.12/3.3.0/spark-cassandra-connector_2
.12-3.3.0.jar
Resolviendo jdbc.postgresql.org [jdbc.postgresql.org]:72.32.157.228, 2001:4880:Se1:1::228
Conectando jdbc.postgresql.org [jdbc.postgresql.org][72.32.157.228]:443... conectado.
Petición HTTP enviada, esperando respuesta... 200 OK
Longitud: 1089312 (1,0M) [application/java-archive]
Guardando Conexión [postgresql-42.7.3.jar]
Guardando Conexión [postgresql-42.7.3.jar]
postgresql-42.7.3.jar 100%[=====]> 1,04M 1,29MB/s en 0,8s
2024-05-11 16:01:55 {1,29 MB/s} - postgresql-42.7.3.jar guardado [1089312/1089312]

[bigdata@bigdatapc:~/sparklines]$ wget https://repo1.maven.org/maven2/com/datastax/spark/spark-
cassandra-connector_2.12/3.3.0/spark-cassandra-connector_2
.12-3.3.0.jar
Resolviendo repo1.maven.org [repo1.maven.org]... 151.101.132.209, 2a04:4e42:1:f:1000:1000:1000:1000
Conectando con repo1.maven.org [repo1.maven.org][151.101.132.209]:443... conectado.
Petición HTTP enviada, esperando respuesta... 200 OK
Longitud: 1636559 (1,0M) [application/java-archive]
Guardando Conexión [spark-cassandra-connector_2.12-3.3.0.jar]
spark-cassandra-connector_2.12-3.3.0.jar 100%[=====]> 1,50M 8,19MB/s en 0,2s
2024-05-11 16:01:55 {8,19 MB/s} - spark-cassandra-connector_2.12-3.3.0.jar guardado [1636559/1636559]
```

Figure 1.22: Descarga de los conectores de Postgres y Cassandra

Ahora, para arrancar una shell de Spark, ejecutaremos el siguiente comando. En este comando especificamos la memoria que queremos que use el driver y los executors, así como el número de cores que queremos que use el driver y los executors. Además, especificamos los conectores que hemos descargado anteriormente.

```
1 bin/spark-shell --master spark://bigdatapc:7077 --driver-memory 1G --  
    executor-memory 1G --total-executor-cores 2 --executor-cores 2 --  
    jars jars/postgresql-42.7.3.jar,jars/spark-cassandra-connector_2  
    .12-3.3.0.jar
```

```
bigdata@bigdatapc: ~
```

```
bin/spark-shell -master spark://bigdatapc:7077 --driver-memory 1G --executor-memory 1G --total-executor-cores 2 --executor-cores 2 --jars jars/postgresql-42.7.3.jar,jars/spark-cassandra-connector_2.12-3.3.0.jar
```

```
24/05/11 23:50:19 WARN Util: Your hostname, bigdataresol, has datagram resolvers
```

```
24/05/11 23:50:19 WARN Util: Set SPARK_LOCAL_IP if you need to bind to another address
```

```
24/05/11 23:50:20 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
```

```
Setting default log level to "WARN".
```

```
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
```

```
Spark context Web UI available at http://192.0.2.15:4040
```

```
Spark context available as 'sc' (master = spark://bigdatapc:7077, app id = app-20240511235029-0006).
```

```
Spark session available as 'spark'.
```

```
Welcome to
```

```
version 3.3.3
```

```
Using Scala version 2.12.15 (OpenJDK 64-Bit Server VM, Java 1.8.0_402)
```

```
Type th expressions to have them evaluated.
```

```
Type help for more information.
```

```
scala: |
```

Figure 1.23: Arranque de la shell de Spark

Si vamos a la web, veremos que se ha creado una aplicación en la interfaz web del Master de Spark.

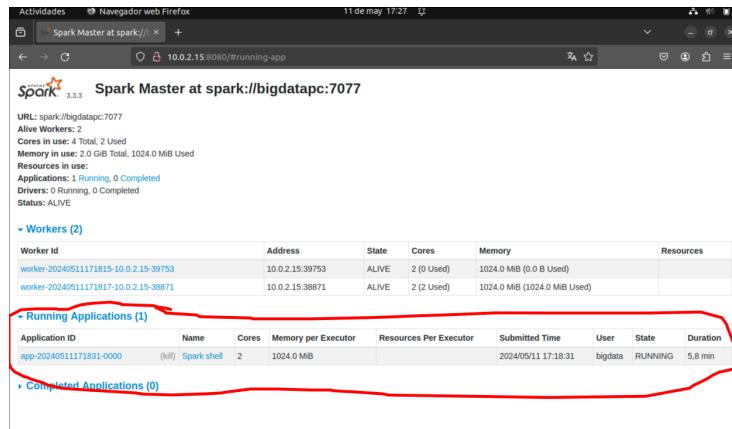


Figure 1.24: Interfaz web del Master de Spark con la aplicación creada

También, al arrancar la shell de Spark, se nos muestra la URL de la interfaz web de la shell de Spark:

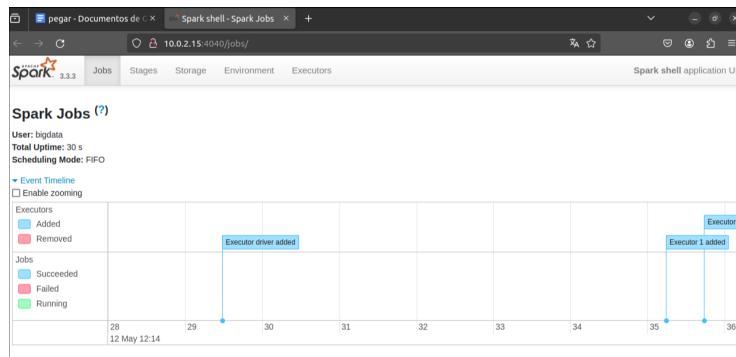


Figure 1.25: Interfaz web de la shell de Spark

Chapter 2

Ingesta de los datos en el lago de datos

Los datos que se van a analizar en este trabajo han sido proporcionados por el profesor en el archivo `datos_practica.tar.gz`. El primer paso es descomprimir el archivo y ver qué contiene. Para ello, se ejecuta el siguiente comando en la terminal:

```
1 tar -xvzf datos_practica.tar.gz
```

Ahora desde la consola de Spark iremos cargando los datos en el lago de datos.

2.1 Countries

Los datos sobre los países en el archivo `countries.txt` está en un formato no estándar: `name::<name> ## iso_code::<iso_code> ## dafif_code::<dafif_code>`, por lo que se va a tener que tratar como datos desestructurados (RDDs) y realizar una serie de transformaciones con el objetivo de realizar una limpieza de los datos y darle una estructura `name: string, iso_code: string, dafif_code: string` para obtener un DataFrame. Una vez obtenido el DataFrame, se debe utilizar el conector csv para guardar los datos en el path `/practica/countries/` de HDFS.

Primero introducimos nuestro archivo en un RDD y lo mostramos:

```
1 val rdd = sc.textFile("/home/bigdata/practica/countries.txt")
2 rdd.take(5).foreach(println)
```

```

scala> val rdd = sc.textFile("/home/bigdata/practica/countries.txt")
rdd: org.apache.spark.rdd.RDD[String] = /home/bigdata/practica/countries.txt MapPartitionsRDD[3] at textFile at <console>:23
scala> rdd.take(5).foreach(println)
name::Bonaire, Saint Eustatius and Saba ## iso_code::BQ ## dafif_code::BQ
name::Aruba ## iso_code::AW ## dafif_code::AW
name::Curaçao ## iso_code::CW ## dafif_code::CW
name::United Arab Emirates ## iso_code::AE ## dafif_code::AE
name::Afghanistan ## iso_code::AF ## dafif_code::AF
scala>

```

Figure 2.1: RDD de los países

Ahora vamos a realizar una serie de transformaciones para limpiar los datos y darles una estructura. Para ello, vamos a utilizar el método map para dividir las líneas por el separador ##, eliminar los espacios en blanco y eliminar los elementos vacíos. A continuación, filtramos las líneas que no tengan 3 elementos y mostramos el resultado:

```

1 val lines = rdd.map(line => line.split("##").map(_.trim).filter(_.nonEmpty)).filter(_.size == 3)
2 lines.collect()(0)

```

```

scala> val lines = rdd.map(line => line.split("##").map(_.trim).filter(_.nonEmpty)).filter(_.size == 3)
lines: org.apache.spark.rdd.RDD[Array[String]] = MapPartitionsRDD[7] at filter at <console>:23
scala> lines.collect()(0)
res18: Array[String] = Array(name::Bonaire, Saint Eustatius and Saba, iso_code::BQ, dafif_code::BQ)
scala>

```

Figure 2.2: RDD de los países limpio

Crearemos la variable columns en la que quitaremos el prefijo de las columnas (ej: name:::) y mostraremos el resultado:

```

1 val columns = lines.map(line => line.map(_.split(":")(1)))
2 columns.collect()(0)

```

```

scala> val columns = lines.map(line => line.map(_.split(":")(1)))
columns: org.apache.spark.rdd.RDD[Array[String]] = MapPartitionsRDD[8] at map at <console>:23
scala> columns.collect()(0)
res19: Array[String] = Array(Bonaire, Saint Eustatius and Saba, BQ, BQ)
scala>

```

Figure 2.3: RDD de los países con nombres de columnas

Ahora ya si crearemos el dataframe de countries:

```

1 val df_countries = columns.map(row => (row(0), row(1), row(2))).toDF("name", "iso_code", "dafif_code")
2 df_countries.show()

```

```

scala> val df_countries = columns.map(row => (row(0), row(1), row(2))).toDF("name", "iso_code", "daftf_code")
df_countries: org.apache.spark.sql.DataFrame = [name: string, iso_code: string ... 1 more field]
scala> df_countries.show()
+-----+-----+-----+
|      name|iso_code|daftf_code|
+-----+-----+-----+
|Bonaire, Sint Eustatius| BQ|     BQ|
|Aruba| AW|     AW|
|Antigua and Barbuda| AG|     AG|
|United Arab Emirates| AE|     AE|
|Afghanistan| AF|     AF|
|Algeria| DZ|     AG|
|Azerbaijan| AZ|     AZ|
|Albania| AL|     AL|
|Armenia| AM|     AM|
|Angola| AO|     AO|
|American Samoa| AS|     AQ|
|Argentina| AR|     AR|
|Australia| AU|     AS|
|Ashmore and Cartier Islands| AW|     AT|
|Austria| AT|     AU|
|Anguilla| AI|     AV|
|Antarctica| AQ|     AY|
|Bahrain| BH|     BA|
|Barbados| BB|     BB|
|Bermuda| BM|     BC|
+-----+-----+-----+
only showing top 20 rows

scala>

```

Figure 2.4: Dataframe de los países

Ahora en otra terminal inciamos HDFS.

```
1 hadoop-3.3.6/sbin/start-dfs.sh
```

```

bigdata@bigdatapc:~$ hadoop-3.3.6/sbin/start-dfs.sh
Starting namenodes on [localhost]
Starting datanodes
Starting secondary namenodes [bigdatapc]
bigdata@bigdatapc:~$ 

```

Figure 2.5: Iniciando HDFS

Y guardamos el dataframe en HDFS desde la consola de scala:

```
1 df_countries.write.format("csv").option("header", "true").save("hdfs://localhost:9000/practica/countries/")
```

```

scala> df_countries.write.format("csv").option("header", "true").save("hdfs://localhost:9000/practica/countries/")
scala>

```

Figure 2.6: Guardando el dataframe en HDFS

Mostramos el contenido de la carpeta /practica/countries/ en HDFS:

```
1 hadoop-3.3.6/bin/hdfs dfs -ls /practica/countries/
```

```

bigdata@bigdatapc:~$ hadoop-3.3.6/bin/hdfs dfs -ls /practica/countries/
-rw-r--r-- 3 bigdata supergroup 0 2024-05-11 18:25 /practica/countries/_SUCCESS
-rw-r--r-- 3 bigdata supergroup 2291 2024-05-11 18:25 /practica/countries/part-00000-16afa48f-9c98-45ff-a572-2614ced3c572-c000.cs
-rw-r--r-- 3 bigdata supergroup 2226 2024-05-11 18:25 /practica/countries/part-00001-16afa48f-9c98-45ff-a572-2614ced3c572-c000.cs
bigdata@bigdatapc:~$
```

Figure 2.7: Contenido de la carpeta countries en HDFS

Por último, mostraremos 5 filas de los datos almacenados en HDFS:

```

1 spark.read.format("csv").option("header", "true").option("inferSchema",
    "true").load("hdfs://localhost:9000/practica/countries/").limit(5).
    show()
```

```

scala> spark.read.format("csv").option("header", "true").option("inferSchema", "true").load("hdfs://localhost:9000/practica/countries/").
  limit(5).show()
+-----+-----+
| name |iso_code|dafff_code|
+-----+-----+
| Bonaire, Saint E...| BQ| BQ|
| Aruba| AW| AA|
| Antigua and Barbuda| AG| AA|
| United Arab Emirates| AE| AE|
| Afghanistan| AF| AF|
+-----+-----+
```

Figure 2.8: 5 filas de los datos almacenados en HDFS

2.2 Airlines

Los datos sobre las aerolíneas están en el archivo `airlines.dat` y está en formato CSV. Por lo tanto, al ser datos estructurados, se puede cargar directamente en un DataFrame. Una de las columnas requiere una transformación ya que debería ser booleana en vez de string. Además, se nos pide añadir una columna nueva llamada `country_iso` que corresponde al `iso_code` de los datos CSV que guardamos en HDFS. Una vez terminado, se debe utilizar el conector `parquet` para guardar los datos particionados por la columna `country` en el path `/practica/airlines/` de HDFS.

Primero cargamos los datos de `airlines.dat` en un Dataframe a través del conector de csv y las propiedades necesarias, para ello comenzaremos importando algunas estructuras de datos:

```

1 import org.apache.spark.sql.types.StructType
2 import org.apache.spark.sql.types.StructField
3 import org.apache.spark.sql.types.StringType
4 import org.apache.spark.sql.types.IntegerType
5 import org.apache.spark.sql.types.FloatType
```

```

scala> import org.apache.spark.sql.types.StructType
import org.apache.spark.sql.types.StructType

scala> import org.apache.spark.sql.types.StructField
import org.apache.spark.sql.types.StructField

scala> import org.apache.spark.sql.types.StringType
import org.apache.spark.sql.types.StringType

scala> import org.apache.spark.sql.types.IntegerType
import org.apache.spark.sql.types.IntegerType

scala> import org.apache.spark.sql.types.FloatType
import org.apache.spark.sql.types.FloatType

scala>

```

Figure 2.9: Importando estructuras de datos

Después creamos un esquema para los datos de las aerolíneas:

```

1 val airlines_schema = StructType(Array(StructField("id", IntegerType,
  true), StructField("name", StringType, true), StructField("alias",
  StringType,true), StructField("IATA", StringType, true),StructField(
  "ICAO", StringType, true),StructField("callsign", StringType, true
  ), StructField("country", StringType, true), StructField("active",
  StringType, true)))

```

```

scala> val airlines_schema = StructType(Array(StructField("id", IntegerType, true), StructField("name", StringType, true), StructField("callsign", StringType, true), StructField("alias", StringType, true), StructField("IATA", StringType, true), StructField("ICAO", StringType, true), StructField("country", StringType, true)))
airlines_schema: org.apache.spark.sql.types.StructType = StructType(StructField(id,IntegerType,true),StructField(name,StringType,true),StructField(callsign,StringType,true),StructField(alias,StringType,true),StructField(IATA,StringType,true),StructField(ICAO,StringType,true),StructField(country,StringType,true))
scala>

```

Figure 2.10: Creando el esquema de los datos de las aerolíneas

Cargamos los datos en un DataFrame utilizando el esquema creado:

```

1 var df_airlines = spark.read.format("csv").option("header", "false").
  schema(airlines_schema).load("/home/bigdata/practica/airlines.dat")
2 df_airlines.show()

```

```

scala> val df_airlines = spark.read.format("csv").option("header", "false").schema(airlines_schema).load("/home/bigdata/practica/airlines.csv")
df_airlines: org.apache.spark.sql.DataFrame = [id: int, name: string ... 6 more fields]

scala> df_airlines.show()
+-----+-----+-----+-----+-----+
| id | name| alias| IATA| ICAO| callsign| country| active|
+-----+-----+-----+-----+-----+
| -1 | Unknown| [N] - [N/A] | [N] | [N] | null| [N] | Y |
| 1 | Private Flight| [N] - [N/A] | null| null| null| null| Y |
| 2 | 135 Airlines| [N] null| [GENERAL] | United States| [true] | Y |
| 3 | 1tme Airline| [N] ITI RNX| [NEXTIME] | South Africa| [Y] |
| 4 | 2 Sqn No 1 Element...| [N]null| WYT| null| [United Kingdom] | N |
| 5 | 213 Flight Unit| [N]null| TFI| null| null| Russia| N |
| 6 | 223 Flight Unit| [N]null| CHKALOVSKY-K| null| null| Russia| N |
| 7 | 224th Flight Unit| [N]null| TTF| CARGO UNIT| Russia| N |
| 8 | 247 Jet Ltd| [N]null| TMF| CLOUD RUNNER| [United Kingdom] | N |
| 9 | 30 Aviation| [N]null| SEC| SECUREX| United States| N |
| 10 | 40-MTR| [N]null| MTR| MILITARY| United States| Y |
| 11 | 4D Airl| [N]null| QBT| QUARTET| Thailand| N |
| 12| 611897 Alberta Lt...| [N]null| THD| DONUT| Canada| N |
| 13 | Ansett Australia| [N] ANI AAA| ANSETT| Australia| Y |
| 14 | Abacus Airline| [N] null| null| null| null| Singapore| Y |
| 15 | Abelag Aviation| [N] W9J AAB| ABG| Belgium| N |
| 16 | Army Atr Corps| [N]null| AAC| ARMYAIR| [United Kingdom] | N |
| 17| Aero Aviation Cen...| [N]null| AAD| SUNRISE| Canada| N |
| 18| Aero Servicios El...| [N]null| SII| ASEISA| Mexico| false |
| 19 | Aero Buzua| [N]null| BZS| BINIZA| Mexico| N |
+-----+-----+-----+-----+-----+
only showing top 20 rows

scala>

```

Figure 2.11: Dataframe de las aerolíneas

Ahora realizaremos las transformaciones necesarias para convertir los datos y el tipo de la columna active de String a Boolean

```

1 df_airlines = df_airlines.withColumn("active", when(col("active") =
== "Y", true).otherwise(false))

```

```

scala> df_airlines = df_airlines.withColumn("active", when(col("active") == "Y", true).otherwise(false))
df_airlines: org.apache.spark.sql.DataFrame = [id: int, name: string ... 6 more fields]

scala> df_airlines.show()
+-----+-----+-----+-----+-----+
| id | name| alias| IATA| ICAO| callsign| country| active|
+-----+-----+-----+-----+-----+
| -1 | Unknown| [N] - [N/A] | [N] | [N] | null| [N] | true |
| 1 | Private Flight| [N] - [N/A] | null| null| null| null| true |
| 2 | 135 Airlines| [N] null| [GENERAL] | United States| [true] | true |
| 3 | 1tme Airline| [N] ITI RNX| [NEXTIME] | South Africa| [Y] |
| 4 | 2 Sqn No 1 Element...| [N]null| WYT| null| [United Kingdom] | false |
| 5 | 213 Flight Unit| [N]null| TFI| null| null| Russia| false |
| 6 | 223 Flight Unit| [N]null| CHKALOVSKY-K| null| null| Russia| false |
| 7 | 224th Flight Unit| [N]null| TTF| CARGO UNIT| Russia| false |
| 8 | 247 Jet Ltd| [N]null| TMF| CLOUD RUNNER| [United Kingdom] | false |
| 9 | 30 Aviation| [N]null| SEC| SECUREX| United States| false |
| 10 | 40-MTR| [N]null| MTR| MILITARY| United States| true |
| 11 | 4D Airl| [N]null| QBT| QUARTET| Thailand| false |
| 12| 611897 Alberta Lt...| [N]null| THD| DONUT| Canada| false |
| 13 | Ansett Australia| [N] ANI AAA| ANSETT| Australia| true |
| 14 | Abacus Airline| [N] null| null| null| null| Singapore| true |
| 15 | Abelag Aviation| [N] W9J AAB| ABG| Belgium| false |
| 16 | Army Atr Corps| [N]null| AAC| ARMYAIR| [United Kingdom] | false |
| 17| Aero Aviation Cen...| [N]null| AAD| SUNRISE| Canada| false |
| 18| Aero Servicios El...| [N]null| SII| ASEISA| Mexico| false |
| 19 | Aero Buzua| [N]null| BZS| BINIZA| Mexico| false |
+-----+-----+-----+-----+-----+
only showing top 20 rows

scala>

```

Figure 2.12: Dataframe de las aerolíneas con la columna active convertida a boolean

Ahora cargaremos los datos de los países en un DataFrame para poder unirlo con el DataFrame de las aerolíneas:

```

1 var df_countries = spark.read.format("csv").option("header", "true").
    option("inferSchema", "true").load("hdfs://localhost:9000/practica/
    countries/")
2 countries.show()

```

```

scala> val countries = spark.read.format("csv").option("header", "true").option("inferSchema", "true").load("hdfs://localhost:9000/practica/countries/")
countries: org.apache.spark.sql.DataFrame = [name: string, iso_code: string ... 1 more field]

scala> countries.show()
+-----+-----+
|      name|iso_code|dafff_code|
+-----+-----+
|[Bonaire, Sint Eustatius and Saba|BQ|      |
|[Aruba|AW|      AA|
|[Antigua and Barbuda|AG|      AC|
|[United Arab Emirates|AE|      AE|
|[Afghanistan|AF|      AF|
|[Algeria|DZ|      AG|
|[Azerbaijan|AZ|      AJ|
|[Albania|AL|      AL|
|[American Samoa|AS|      AK|
|[Angola|AO|      AO|
|[American Samoa|AS|      AQ|
|[Argentina|AR|      AR|
|[Australia|AU|      AS|
|[Ashmore and Cartier Islands|N|      AT|
|[Austria|AT|      AU|
|[Agharta|AT|      AV|
|[Antarctica|AQ|      AY|
|[Bahrain|BH|      BA|
|[Barbados|BB|      BB|
|[Botswana|BW|      BC|
+-----+
only showing top 20 rows

scala>

```

Figure 2.13: Dataframe de los países

Como `countries` y `df_airlines` tienen una columna en común, `country`, tenemos que renombrar la columna `name` de `countries` a `country` para poder unir los dos DataFrames.

```
1 df_countries = df_countries.withColumnRenamed("name", "country")
```

Ahora unimos los dos DataFrames por la columna `country` y renombramos la columna `iso_code` a `country_iso`:

```
1 df_airlines = df_airlines.join(df_countries, Seq("country"), "left")
2 df_airlines = df_airlines.withColumnRenamed("iso_code", "country_iso")
```

```

scala> df_airlines = df_airlines.join(df_countries, Seq("country"), "left")
df_airlines: org.apache.spark.sql.DataFrame = [country: string, id: int ... 8 more fields]

scala> df_airlines = df_airlines.withColumnRenamed("iso_code", "country_iso")
df_airlines: org.apache.spark.sql.DataFrame = [country: string, id: int ... 8 more fields]

scala> df_airlines.show()
+-----+-----+-----+-----+-----+-----+-----+-----+
|      country|       id|  name|alias|IATA|ICAO|  callsign|active|country_iso|dafff_code|
+-----+-----+-----+-----+-----+-----+-----+-----+
|[Unknown| -1| Unknown| -1| N/A|      |      | true| null| null| null|
|[null| 1| Private flight| 1| N/A|      |      | true| null| null| null|
|[United States| 2| 135 Airways| 2| GNL|      GENERAL| false| US|      US|
|[South Africa| 3| 1Time Airline| 3| RNX|      NEXTIME| true| ZA|      SF|
|[United Kingdom| 4| 2 Sqn No 1 Element...| 4| WYT|      |      | false| GB|      UK|
|[Russia| 5| 213 Flight Unit| 5| TFU|      |      | false| RU|      RS|
|[Russia| 6| 223 Flight Unit S...| 6| CHD|      CHD|CHKALOVSK-AVIA| false| RU|      RS|
|[Russia| 7| 224th Flight Unit| 7| TFT|      CARGO UNIT| false| RU|      RS|
|[United Kingdom| 8| 247 Jet Ltd| 8| TWF|      CLOUD RUNNER| false| GB|      UK|
|[United States| 9| 3D Aviation| 9| SEC|      SECUREX| false| US|      US|
|[United States| 10| 40-Mile Atr| 10| Q5|      MLA|      MILE-AIR| true| US|      US|
|[Thailand| 11| 40 Atr| 11| QRT|      QUARTET| false| TH|      TH|
|[Canada| 12| 611897 Alberta Li...| 12| THD|      DONUT| false| CA|      CA|
|[Australia| 13| Ansett Australia| 13| AN| AAA|      ANSETT| true| AU|      AS|
|[Singapore| 14| Abacus International| 14| 1B|      null|      null| true| SG|      SN|
|[Belgium| 15| Abelag Aviation| 15| W9| AAB|      ABG| false| BE|      BE|
|[United Kingdom| 16| Army Air Corps| 16| AAC|      ARMYAIR| false| GB|      UK|
|[Canada| 17| Aero Aviation Cen...| 17| AAD|      SUNRISE| false| CA|      CA|
|[Mexico| 18| Aero Servicios Ej...| 18| SII|      ASEISA| false| MX|      MX|
|[Mexico| 19| Aero Blniza| 19| BZS|      BINIZA| false| MX|      MX|
+-----+
only showing top 20 rows

```

Figure 2.14: Dataframe de las aerolíneas con los países

Vamos a eliminar la columna `dafff_code` ya que no la necesitamos:

```
1 df_airlines = df_airlines.drop("drafif_code")
```

También reordenaremos las columnas para que `id` sea la primera columna:

```
1 df_airlines = df_airlines.select("id", "name", "alias", "IATA", "ICAO",
    "callsign", "country", "country_iso", "active")
```

```
scala> df_airlines = df_airlines.drop("drafif_code")
df_airlines: org.apache.spark.sql.DataFrame = [country: string, id: int ... 7 more fields]
scala> df_airlines.select("id", "name", "alias", "IATA", "ICAO", "callsign", "country", "country_iso", "active")
df_airlines: org.apache.spark.sql.DataFrame = [id: int, name: string ... 7 more fields]

scala> df_airlines.show()
+---+-----+-----+-----+-----+-----+-----+-----+
| id|      name|alias|IATA|ICAO| callsign| country|country_iso|active|
+---+-----+-----+-----+-----+-----+-----+-----+
|  1|Unknown|  N/A|  N/A|  N/A|       |       |       |  true |
|  1|Private flight|  N/A|  N/A|  N/A|       |       |       |  true |
|  2|135 Airways|  N|GNL|GENERAL|United States|       |       |  false|
|  3|1Tme Airline|  N|IT|RNX|NEXTIME|South Africa|       |  true |
|  4|2 Sqn No 1 Elemen...|  N|null|WYT|       |null|United Kingdom|  false|
|  5|213 Flight Unit|  N|null|TFU|       |null|Russia|  false|
|  6|223 Flight Unit S...|  N|null|CHD|CHKALOVSK-AVIA|Russia|  false|
|  7|224th Flight Unit|  N|null|TTF|CARGO UNIT|Russia|  false|
|  8|247 Jet Ltd|  N|null|TWF|CLOUD RUNNER|United Kingdom|  false|
|  9|3D Aviation|  N|null|SEC|SECUREX|United States|  false|
| 10|40-Mile Air|  N|Q5|MLA|MILE-AIR|United States|  true |
| 11|4D Airl|  N|null|QRT|QUARTET|Thailand|  false|
| 12|611897 Alberta Lt...|  N|null|THD|DONUT|Canada|  false|
| 13|Ansett Australia|  N|AN|AAA|ANSETT|Australia|  AU|  true |
| 14|Abacus Internation...|  N|B|null|       |Singapore|SG|  true |
| 15|Abelag Aviation|  N|W9|AAB|ABG|Belgium|BE|  false|
| 16|Army Air Corps|  N|null|AAC|ARMYAIR|United Kingdom|  false|
| 17|Aero Aviation Cen...|  N|null|AAD|SUNRISE|Canada|  CA|  false|
| 18|Aero Servicios Ej...|  N|null|SII|ASEISA|Mexico|  MX|  false|
| 19|Aero Bñiza|  N|null|BZS|BINIZA|Mexico|  MX|  false|
+---+-----+-----+-----+-----+-----+-----+-----+
only showing top 20 rows
```

Figure 2.15: Dataframe de las aerolíneas con las columnas reordenadas

Vamos a eliminar las filas que tengan valores nulos o `\N` en la columna `country`:

```
1 df_airlines = df_airlines.filter(col("country").isNotNull && col("country") != "\\\N")
```

```
scala> df_airlines = df_airlines.filter(col("country").isNotNull && col("country") != "\\\N")
df_airlines: org.apache.spark.sql.DataFrame = [id: int, name: string ... 7 more fields]

scala> df_airlines.show()
+---+-----+-----+-----+-----+-----+-----+
| id|      name|alias|IATA|ICAO| callsign| country|country_iso|active|
+---+-----+-----+-----+-----+-----+-----+
|  2|135 Airways|  N|GNL|GENERAL|United States|       |       |  false|
|  3|1Tme Airline|  N|IT|RNX|NEXTIME|South Africa|       |  true |
|  4|2 Sqn No 1 Elemen...|  N|null|WYT|       |null|United Kingdom|  false|
|  5|213 Flight Unit|  N|null|TFU|       |null|Russia|  false|
|  6|223 Flight Unit S...|  N|null|CHD|CHKALOVSK-AVIA|Russia|  false|
|  7|224th Flight Unit|  N|null|TTF|CARGO UNIT|Russia|  false|
|  8|247 Jet Ltd|  N|null|TWF|CLOUD RUNNER|United Kingdom|  false|
|  9|3D Aviation|  N|null|SEC|SECUREX|United States|  false|
| 10|40-Mile Air|  N|Q5|MLA|MILE-AIR|United States|  true |
| 11|4D Airl|  N|null|QRT|QUARTET|Thailand|  false|
| 12|611897 Alberta Lt...|  N|null|THD|DONUT|Canada|  false|
| 13|Ansett Australia|  N|AN|AAA|ANSETT|Australia|  AU|  true |
| 14|Abacus Internation...|  N|B|null|       |Singapore|SG|  true |
| 15|Abelag Aviation|  N|W9|AAB|ABG|Belgium|BE|  false|
| 16|Army Air Corps|  N|null|AAC|ARMYAIR|United Kingdom|  false|
| 17|Aero Aviation Cen...|  N|null|AAD|SUNRISE|Canada|  CA|  false|
| 18|Aero Servicios Ej...|  N|null|SII|ASEISA|Mexico|  MX|  false|
| 19|Aero Bñiza|  N|null|BZS|BINIZA|Mexico|  MX|  false|
+---+-----+-----+-----+-----+-----+-----+
only showing top 20 rows
```

Figure 2.16: Dataframe de las aerolíneas sin valores nulos en la columna `country`

Por último, guardamos el DataFrame en HDFS particionado por la columna `country`:

```
1 df_airlines.write.format("parquet").partitionBy("country").save("hdfs://localhost:9000/practica/airlines/")
```

Cargramos los datos de las aerolíneas desde HDFS y mostramos 5 filas:

```
1 spark.read.format("parquet").load("hdfs://localhost:9000/practica/airlines/").limit(5).show()
```

| id | name alias IATA ICAO | callsign country isactive | country |
|---|----------------------|---------------------------|---------|
| 2 135 Airways \N null GNL GENERAL US false United States | | | |
| 9 3D Aviation \N null SEC SECUREX US false United States | | | |
| 10 40-Mile Air \N QSI MLA MILE-AIR US true United States | | | |
| 22 Aloha Airlines \N AQI AAH ALOHA US true United States | | | |
| 23 Alaska Island Air \N null AAK ALASKA ISLAND US false United States | | | |

Figure 2.17: 5 filas de los datos de las aerolíneas en HDFS

2.3 Airports

Los datos sobre aeropuertos (fichero airports.dat) están en formato CSV. Por tanto, estos datos se pueden cargar en un Dataframe con el conector correspondiente y las opciones que se requieran. Una de las columnas está en una unidad de medida que no es compatible con los sistemas del cliente. Esa columna es altitude y sus valores están en pies. Para realizar la transformación a metros, se debe desarrollar una UDF y ser aplicada en dicha columna. Una vez obtenido este Dataframe, se debe utilizar el conector JDBC para guardar los datos en la tabla airports de Postgres. Además, el líder técnico de nuestro proyecto indica que, al hacer la lectura de estos datos, se deben añadir las propiedades partitionColumn, lowerBound, upperBound y numPartitions propias del conector JDBC.

Creamos el esquema de los datos de los aeropuertos:

```
1 val airports_schema = StructType(Array(StructField("id", IntegerType, true), StructField("name", StringType, true), StructField("city", StringType, true), StructField("country", StringType, true), StructField("IATA", StringType, true), StructField("ICAO", StringType, true), StructField("latitude", FloatType, true), StructField("longitude", FloatType, true), StructField("altitude", IntegerType, true), StructField("timeZone", FloatType, true), StructField("DST", StringType, true), StructField("tz_database", StringType, true), StructField("tipo", StringType, true), StructField("source", StringType, true)))
```

```

scala> val airports_schema = StructType(Array(StructField("id", IntegerType,
true), StructField("name", StringType, true), StructField("city",
StringType, true), StructField("country", StringType, true),
StructField("IATA", StringType, true), StructField("ICAO",
StringType, true), StructField("latitude", FloatType, true),
StructField("longitude", FloatType, true), StructField("altitude",
IntegerType, true), StructField("tz_database", StringType, true),
StructField("dst", StringType, true), StructField("tz_database",
StringType, true), StructField("tpo", StringType, true),
StructField("source", StringType, true)))
StructType = StructType(StructField(id, IntegerType, true), StructField(name, StringType, true), StructField(city, StringType, true), StructField(country, StringType, true), StructField(IATA, StringType, true), StructField(ICAO, StringType, true), StructField(latitude, FloatType, true), StructField(longitude, FloatType, true), StructField(altitude, IntegerType, true), StructField(tz_database, StringType, true), StructField(dst, StringType, true), StructField(tz_database, StringType, true), StructField(tpo, StringType, true), StructField(source, StringType, true))

scala> airports = spark.read.schema(airports_schema).load("/home/bigdata/practica/airports.dat")

```

Figure 2.18: Creando el esquema de los datos de los aeropuertos

Cargamos los datos en un DataFrame utilizando el esquema creado:

```

1 var df_airports = spark.read.format("csv").option("header", "false").
schema(airports_schema).load("/home/bigdata/practica/airports.dat")

```

```

scala> var db_airports = spark.read.format("csv").option("header", "false").schema(airports_schema).load("/home/bigdata/practica/airports.dat")
db_airports: org.apache.spark.sql.DataFrame = [id: int, name: string ... 12 more fields]

scala> db_airports.show()
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | name | city | country|IATA|ICAO|latitude|longitude|altitude|timeZone|dst|tz_database| tpo| source|
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | Goroka Airport | Goroka|Papua New Guinea| GKA|AVGA|-6.08169| 145.392| 5282| 10.0| U|Pacific/Port_Moresby|alrpo|rtOurAirports
| 2 | Madang Airport | Madang|Papua New Guinea| MAG|AYWD|-5.20708| 145.789| 28| 10.0| U|Pacific/Port_Moresby|alrpo|rtOurAirports
| 3 | Mount Hagen Kagan...| Mount Hagen|Papua New Guinea| HGU|AVWH|-5.82879| 144.296| 5388| 10.0| U|Pacific/Port_Moresby|alrpo|rtOurAirports
| 4 | Nadzab Airport | Nadzab|Papua New Guinea| LAE|AYNZ|-6.56988| 146.72598| 239| 10.0| U|Pacific/Port_Moresby|alrpo|rtOurAirports
| 5 | Port Moresby Jack...| Port Moresby|Papua New Guinea| POM|AYPY|-9.44338| 147.22| 146| 10.0| U|Pacific/Port_Moresby|alrpo|rtOurAirports
| 6 | Newak Internation...| Newak|Papua New Guinea| WMK|AVWK|-3.58383| 143.669| 19| 10.0| U|Pacific/Port_Moresby|alrpo|rtOurAirports
| 7 | Narssarssuaq Airport | Narssarssuaq|Greenland| UAK|BGBW| 61.1605| -45.426| 112| -3.0| E| America/Godthab|alrpo|rtOurAirports
| 8 | Godthaab / Nuuk A...| Godthaab |Greenland| GOM|BGHH| 64.1969| -51.6781| 283| -3.0| E| America/Godthab|alrpo|rtOurAirports
| 9 | Kangerlussuaq Air...| Sondrestrom|Greenland| SFJ|BGSF| 67.01222|-50.711605| 165| -3.0| E| America/Thule|alrpo|rtOurAirports
| 10 | Thule At Base | Thule |Greenland| THU|BGTL| 76.5312| -68.7032| 251| -4.0| E| America/Thule|alrpo|rtOurAirports
| 11 | Akureyri Airport | Akureyri |Iceland| AEY|BIAR| 65.66| -18.7227| 6| 0.0| N| Atlantic/Reykjavik|alrpo|rtOurAirports
| 12 | Egilstadir Airport | Egilstadir |Iceland| EGS|BIEG| 65.2833| -14.4014| 76| 0.0| N| Atlantic/Reykjavik|alrpo|rtOurAirports
| 13 | Hornafjordur Airport | Hofn |Iceland| HFN|BIHN| 64.2956| -15.2272| 24| 0.0| N| Atlantic/Reykjavik|alrpo|rtOurAirports
| 14 | Husavik Airport | Husavik |Iceland| HZK|BIHU| 65.9523| -17.426| 48| 0.0| N| Atlantic/Reykjavik|alrpo|rtOurAirports
| 15 | Isafjordur Airport | Isafjordur |Iceland| IFJ|BIIS| 66.0581| -23.1553| 8| 0.0| N| Atlantic/Reykjavik|alrpo|rtOurAirports
| 16 | Keflavik Internat...| Keflavik |Iceland| KEF|BIKF| 63.985| -22.6056| 171| 0.0| N| Atlantic/Reykjavik|alrpo|rtOurAirports
| 17 | Patreksfjordur Al...| Patreksfjordur |Iceland| PFF|BIPA| 65.5558| -23.965| 11| 0.0| N| Atlantic/Reykjavik|alrpo|rtOurAirports
| 18 | Reykjavik Airport | Reykjavik |Iceland| RKV|BIRK| 64.13| -21.9406| 48| 0.0| N| Atlantic/Reykjavik|alrpo|rtOurAirports
| 19 | Siglufjordur Airport | Siglufjordur |Iceland| SJF|BISI| 66.1333| -18.9167| 10| 0.0| N| Atlantic/Reykjavik|alrpo|rtOurAirports
| 20 | Vestmannaeyjar Al...| Vestmannaeyjar |Iceland| VEY|BIVM| 63.4243| -20.2789| 326| 0.0| N| Atlantic/Reykjavik|alrpo|rtOurAirports
+-----+-----+-----+-----+-----+-----+-----+-----+
only showing top 20 rows

```

Figure 2.19: Dataframe de los aeropuertos

Ahora haremos la conversión de la columna `altitude` de pies a metros, primero creamos la variable `feetToMeters` que será nuestra UDF:

```

1 val feetToMeters = udf((x: Float) => x / 3.281)

```

```

scala> val feetToMeters = udf((x: Float) => x / 3.281)
feetToMeters: org.apache.spark.sql.expressions.UserDefinedFunction = SparkUserDefinedFunction($Lambda54439/1889740874@44bf174,doubl
eType,List(Some(class[value@0: float]),Some(class[value@0: double]),None,false,true)

```

Figure 2.20: Creando la UDF `feetToMeters`

Y aplicamos la UDF a la columna `altitude`:

```

1 df_airports = df_airports.withColumn("altitude", feetToMeters(col("altitude")))

```

```

scala> db_airports = db_airports.withColumn("altitude", feetToMeters(col("altitude")))
db_airports: org.apache.spark.sql.DataFrame = [id: int, name: string ... 12 more fields]

scala> db_airports.show()
+-----+-----+-----+-----+-----+-----+-----+
| id | name | city | country | IATA | ICAO | latitude | longitude | altitude | timeZone|DST| tz_data |
+-----+-----+-----+-----+-----+-----+-----+
| 1 | Goroka Airport | Goroka|Papua New Guinea| GKA|AYGA | -6.08169 | 145.392 | 1099.8750380981407 | 10.0 | UIPacific|Port_Mor |
| 2 | Madang Airport | Madang|Papua New Guinea| MAG|AYMD | -5.20708 | 145.789 | 6.09570252971655 | 10.0 | UIPacific|Port_Mor |
| 3 | Mount Hagen Kagan...| Mount Hagen|Papua New Guinea| HGU|AYMH | -5.82679 | 144.296 | 1642.1822615056385 | 10.0 | UIPacific|Port_Mor |
| 4 | Nadzab Airport | Nadzab|Papua New Guinea| LAE|AYNZ | -6.569883 | 146.72598 | 72.84364523011277 | 10.0 | UIPacific|Port_Mor |
| 5 | Port Moresby Jack...| Port Moresby|Papua New Guinea| POM|AYPY | -9.44338 | 147.221 | 44.498628466939881 | 10.0 | UIPacific|Port_Mor |
| 6 | Wewak International...| Wewak|Papua New Guinea| WKK|AYWK | -3.58383 | 143.669 | 5.790917403230722 | 10.0 | UIPacific|Port_Mor |
| 7 | Narssarssuaq | Narssarssuaq|Greenland| UAK|BGW | 61.1605 | -45.426 | 34.135934166412675 | -3.0 | E| America/God |
| 8 | Godthaab | Godthaab|Greenland| GOH|BGHH | 64.1989 | -51.6781 | 86.25419079548918 | -3.0 | E| America/God |
| 9 | Kangerlussuaq Air...| Sondrestrom|Greenland| SFJ|GSF | 67.01222 | -50.710651 | 50.28954587916153 | -3.0 | E| America/God |
| 10 | Thule Air Base | Thule|Greenland| THU|BGT | 76.5312 | -68.7032 | 76.5010667479427 | -4.0 | E| America/T |
| 11 | Akureyri Airport | Akureyri|Iceland| AEY|BIAR | 65.661 | -18.07271 | 1.828710758914965 | 0.0 | NI| Atlantic/Reykj |
| 12 | Egilsstadir Airport | Egilsstadir|Iceland| EG5|BTIG | 65.2833 | -14.4914 | 23.163669612922888 | 0.0 | NI| Atlantic/Reykj |
| 13 | Hornafjordur Airport | Hofn|Iceland| HFN|BIHN | 64.2956 | -15.2272 | 7.31484383656986 | 0.0 | NI| Atlantic/Reykj |
| 14 | Husavik Airport | Husavik|Iceland| HZK|BIHU | 65.9523 | -17.428 | 14.62968607131972 | 0.0 | NI| Atlantic/Reykj |
| 15 | Isafjordur Airport | Isafjordur|Iceland| IFJ|BIIIS | 66.0581 | -23.1353 | 2.4382810118866196 | 0.0 | NI| Atlantic/Reykj |
| 16 | Keflavik Internat...| Keflavik|Iceland| KEF|BIKF | 63.985 | -22.0056 | 52.11825662290765 | 0.0 | NI| Atlantic/Reykj |
| 17 | Patreksfjordur Air...| Patreksfjordur|Iceland| PFI|BIPA | 65.5558 | -23.965 | 3.3526363913441024 | 0.0 | NI| Atlantic/Reykj |
| 18 | Reykjavik Airport | Reykjavik|Iceland| RKV|BIRK | 64.13 | -21.9406 | 14.62968607131972 | 0.0 | NI| Atlantic/Reykj |
| 19 | Siglufjordur Airport | Siglufjordur|Iceland| SIJ|BISI | 66.1333 | -18.9167 | 3.047851204858275 | 0.0 | NI| Atlantic/Reykj |
| 20 | Vestmannayjar Air...| Vestmannayjar|Iceland| VEY|BIVM | 63.4243 | -20.2789 | 99.35995123437976 | 0.0 | NI| Atlantic/Reykj |
+-----+-----+-----+-----+-----+-----+-----+
only showing top 20 rows

```

Figure 2.21: Dataframe de los aeropuertos con la columna altitude convertida a metros

Para poder guardar los datos en la tabla airports de Postgres, primero la tendremos que crear. Primero nos metemos en la consola de Postgres:

```
1 sudo -u postgres psql
```

```

bigdata@bigdatapc:~/spark-3.3.3-bin-hadoop3$ sudo -u postgres psql
[sudo] contraseña para bigdata:
could not change directory to "/home/bigdata/spark-3.3.3-bin-hadoop3": Permiso denegado
psql (14.11 (Ubuntu 14.11-0ubuntu0.22.04.1))
Type "help" for help.

postgres=#

```

Figure 2.22: Entrando en la consola de Postgres

Ahora creamos la tabla airports:

```

1 CREATE TABLE airports(id INT, name VARCHAR(255), city VARCHAR(255),
    country VARCHAR(255), IATA VARCHAR(255), ICAO VARCHAR(255),
    latitude FLOAT, longitude FLOAT, altitude DOUBLE PRECISION,
    timeZone INT, DST VARCHAR(255), tz_database VARCHAR(255), tipo
    VARCHAR(255), source VARCHAR(255));

```

```
postgres=# CREATE TABLE airports(id INT, name VARCHAR(255), city VARCHAR(255),
country VARCHAR(255), IATA VARCHAR(255), ICAO VARCHAR(255),
latitude FLOAT, longitude FLOAT, altitude DOUBLE PRECISION,
timeZone INT, DST VARCHAR(255), tz_database VARCHAR(255), tipo
VARCHAR(255), source VARCHAR(255));
CREATE TABLE
postgres=#
```

Figure 2.23: Creando la tabla airports en Postgres

También deberemos darle una contraseña al usuario postgres:

```
1 ALTER USER postgres WITH PASSWORD 'postgres';
```

```
postgres=# ALTER USER postgres WITH PASSWORD 'postgres';
ALTER ROLE
postgres#
```

Figure 2.24: Dando una contraseña al usuario postgres

Guardamos los datos en la tabla airports de Postgres:

```
1 import java.util.Properties
2 val properties = new Properties()
3
4 properties.setProperty("user", "postgres")
5 properties.setProperty("driver", "org.postgresql.Driver")
6 properties.setProperty("password", "postgres")
7 properties.setProperty("partitionColumn", "altitude")
8 properties.setProperty("lowerBound", "0")
9 properties.setProperty("upperBound", "2000")
10 properties.setProperty("numPartitions", "10")
11 properties.setProperty("password", "postgres")
12
13 df_airports.write.mode("overwrite").jdbc("jdbc:postgresql://localhost
:5432/postgres", "airports", properties)
```

```

scala> import java.util.Properties
import java.util.Properties
scala> val properties = new Properties()
properties: java.util.Properties = {}
scala>
scala> properties.setProperty("user", "postgres")
res1: Object = null
scala> properties.setProperty("driver", "org.postgresql.Driver")
res1: Object = null
scala> properties.setProperty("password", "postgres")
res1: Object = null
scala> properties.setProperty("partitionColumn", "altitude")
res1: Object = null
scala> properties.setProperty("lowerBound", "0")
res2: Object = null
scala> properties.setProperty("upperBound", "2000")
res2: Object = null
scala> properties.setProperty("numPartitions", "10")
res2: Object = null
scala> properties.setProperty("password", "postgres")
res2: Object = postgres
scala>
scala> df_airports.write.mode("overwrite").jdbc("jdbc:postgresql://localhost:5432/postgres", "airports_schema", properties)
scala> 

```

Figure 2.25: Guardando los datos en la tabla airports de Postgres

Mostramos las 5 filas de la tabla airports de Postgres añadiendo los parámetros necesarios. Los parámetros lowerBound y upperBound se utilizan para definir el rango de valores de la columna altitude y numPartitions para definir el número de particiones. El parámetro partitionColumn se utiliza para definir la columna por la que se particionarán los datos. Es importante utilizar estos parámetros para mejorar el rendimiento de la lectura de los datos.

```

1 spark.read.option("partitionColumn", "altitude").option("lowerBound",
0).option("upperBound", 2000).option("numPartitions", 10).jdbc(""
jdbc:postgresql://localhost:5432/postgres", "airports", properties)
.limit(5).show()

```

| id | name | city | country | IATA | ICAO | latitude | longitude | altitude | timeZone | dst | tz_database |
|----|-------------------------------|--------------|------------------|------|------|-----------|------------|--------------------|----------|-----|------------------------|
| 1 | Madang Airport | Madang | Papua New Guinea | MAG | AVMD | -5.20708 | 145.789 | 6.09570252971655 | 10.0 | 0 | UJPacific/Port_Moresby |
| 2 | Nadzab Airport | Nadzab | Papua New Guinea | LAE | AVNZ | -6.569803 | 146.72598 | 72.84364523011277 | 10.0 | 0 | UJPacific/Port_Moresby |
| 3 | Hagorok Airport | Hagorok | Papua New Guinea | HAG | AVYH | -6.569803 | 146.72598 | 72.84364523011277 | 10.0 | 0 | UJPacific/Port_Moresby |
| 4 | Port Moresby Jack... | Port Moresby | Papua New Guinea | POM | AVPY | -9.44338 | 147.22 | 44.998628466939881 | 10.0 | 0 | UJPacific/Port_Moresby |
| 5 | Port Moresby International... | Port Moresby | Papua New Guinea | WIK | AYWK | -3.58383 | 143.0691 | 5.790917403230722 | 10.0 | 0 | UJPacific/Port_Moresby |
| 6 | Hewak Airport | Hewak | Papua New Guinea | HAK | AVYK | -3.58383 | 143.0691 | 5.790917403230722 | 10.0 | 0 | UJPacific/Port_Moresby |
| 7 | Narsarsuaq Airport Narsarsuaq | Narsarsuaq | Greenland | UAK | BGBW | 61.1605 | -45.426134 | 13.15934166412675 | -3.0 | E | America/Godthaab |

Figure 2.26: 5 filas de la tabla airports de Postgres

2.4 Routes

Antes de comenzar, es necesario utilizar un comando diferente para levantar la shell de spark. El JAR del conector de Cassandra no funciona correctamente, por lo que habrá que cargarlo mediante el paquete:

```

1 bin/spark-shell --master spark://bigdatapc:7077 --executor-memory 2g --
  executor-cores 2 --jars jars/postgresql-42.7.3.jar --packages com.
  datastax.spark:spark-cassandra-connector_2.12:3.3.0 --conf spark.
  cassandra.connection.host=localhost

```

Los datos sobre rutas (fichero routes.dat) están en formato CSV. Por tanto, estos datos se pueden cargar en un Dataframe con el conector correspondiente y las opciones que se requieran. Una vez obtenido este Dataframe, se debe utilizar el conector de Cassandra para guardar los datos en la tabla routes del keyspace practica.

Creamos el esquema de los datos de las rutas:

```

1 val routes_schema = StructType(Array(StructField("airline", StringType,
  true), StructField("airline_id", IntegerType, true), StructField("source_airport",
  StringType, true), StructField("source_airport_id", IntegerType, true), StructField("destination_airport",
  StringType, true), StructField("destination_airport_id", IntegerType, true),
  StructField("codeshare", StringType, true), StructField("stops",
  IntegerType, true), StructField("equipment", StringType, true)))

```

```

scala> val routes_schema = StructType(Array(StructField("airline", StringType, true), StructField("airlineID", IntegerType, true), StructField("sourceAirport", StringType, true), StructField("sourceAirportID", IntegerType, true), StructField("destinationAirport", StringType, true), StructField("destinationAirportID", IntegerType, true), StructField("codeshare", StringType, true), StructField("stops", IntegerType, true), StructField("equipment", StringType, true)))

```

Figure 2.27: Creando el esquema de los datos de las rutas

Cargamos los datos en un DataFrame utilizando el esquema creado:

```

1 val df_routes = spark.read.format("csv").option("header", "false").
  schema(routes_schema).load("/home/bigdata/practica/routes.dat")

```

```

scala> val df_routes = spark.read.format("csv").option("header", "false").schema(routes_schema).load("/home/bigdata/practica/routes.dat")
df_routes: org.apache.spark.sql.DataFrame = [airline: string, airlineID: int ... 7 more fields]
scala> df_routes.show()
+-----+-----+-----+-----+-----+-----+-----+-----+
|airline|airlineID|sourceAirport|sourceAirportID|destinationAirport|destinationAirportID|codeshare|stops|equipment|
+-----+-----+-----+-----+-----+-----+-----+-----+
| 2B| 410| AER| 2965| KZN| 2990| null| 0| CR2|
| 2B| 410| ASF| 2964| KZN| 2991| null| 0| CR2|
| 2B| 410| ASF| 2966| HRV| 2962| null| 0| CR2|
| 2B| 410| CEK| 2968| KZN| 2990| null| 0| CR2|
| 2B| 410| CEK| 2968| OVB| 4078| null| 0| CR2|
| 2B| 410| DME| 4029| KZN| 2991| null| 0| CR2|
| 2B| 410| DME| 4029| NBC| 6969| null| 0| CR2|
| 2B| 410| DME| 4029| TGK| null| null| 0| CR2|
| 2B| 410| DME| 4029| UUA| 6160| null| 0| CR2|
| 2B| 410| EGO| 6151| KZN| 2990| null| 0| CR2|
| 2B| 410| EGO| 6156| KZN| 2990| null| 0| CR2|
| 2B| 410| GVD| 2922| NBC| 6969| null| 0| CR2|
| 2B| 410| KGD| 2952| EGO| 6156| null| 0| CR2|
| 2B| 410| KZN| 2990| ASI| 3041| null| 0| CR2|
| 2B| 410| KZN| 2990| ASF| 2966| null| 0| CR2|
| 2B| 410| KZN| 2990| CEX| 2968| null| 0| CR2|
| 2B| 410| KZN| 2990| DME| 4029| null| 0| CR2|
| 2B| 410| KZN| 2990| LED| 2940| null| 0| CR2|
| 2B| 410| KZN| 2990| SVX| 2975| null| 0| CR2|
+-----+-----+-----+-----+-----+-----+-----+-----+
only showing top 20 rows

```

Figure 2.28: Dataframe de las rutas

Vamos a añadir una columna route_id que será la primary key de la tabla routes de Cassandra. Para ello a cada fila le añadiremos un identificador único:

```

1 import org.apache.spark.sql.functions.monotonically_increasing_id
2 df_routes = df_routes.withColumn("route_id",
  monotonically_increasing_id())

```

Reordenamos para que la columna route_id sea la primera:

```
1 df_routes = df_routes.select("route_id", "airline", "airline_id", "source_airport", "source_airport_id", "destination_airport", "destination_airport_id", "codeshare", "stops", "equipment")
```

```
scala> import org.apache.spark.sql.functions.monotonically_increasing_id
import org.apache.spark.sql.functions.monotonically_increasing_id
scala> df_routes = df_routes.withColumn("route_id", monotonically_increasing_id())
df_routes: org.apache.spark.sql.DataFrame = [route_id: bigint, airline: string ... 8 more fields]
scala>
scala> df_routes.select("route_id", "airline", "airline_id", "source_airport", "source_airport_id", "destination_airport", "destination_airport_id", "codeshare", "stops", "equipment")
df_routes: org.apache.spark.sql.DataFrame = [route_id: bigint, airline: string ... 8 more fields]
scala> df_routes.show(5)
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| route_id | airline | airline_id | source_airport | source_airport_id | destination_airport | destination_airport_id | codeshare | stops | equipment |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 0 | 2B | 410 | AER | 2965 | KZN | 2990 | null | 0 | CR2 |
| 1 | 2B | 410 | ASFI | 2966 | KZN | 2990 | null | 0 | CR2 |
| 2 | 2B | 410 | ASFI | 2967 | MMV | 2990 | null | 0 | CR2 |
| 3 | 2B | 410 | CEK | 2968 | KZN | 2990 | null | 0 | CR2 |
| 4 | 2B | 410 | CEK | 2969 | OVJ | 4078 | null | 0 | CR2 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
only showing top 5 rows
```

Figure 2.29: Dataframe de las rutas con la columna route_id

Creamos una nueva terminal y ejecutamos el siguiente comando para conectarnos a la shell de Cassandra:

```
1 ~/apache-cassandra-3.11.16/bin/cqlsh
```

```
bigdata@bigdatapc:~/spark-3.3.3-bin-hadoop3$ ~/apache-cassandra-3.11.16/bin/cqlsh
Connected to Test Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 3.11.16 | CQL spec 3.4.4 | Native protocol v4]
Use HELP for help.
cqlsh>
```

Figure 2.30: Entrando en la shell de Cassandra

Creamos la tabla routes en el keyspace practica:

```
1 USE practica;
2
3 CREATE TABLE routes (
4     route_id int,
5     airline text,
6     airline_id int,
7     source_airport text,
8     source_airport_id int,
9     destination_airport text,
10    destination_airport_id int,
11    codeshare text,
12    stops int,
13    equipment text,
14    PRIMARY KEY (route_id)
15 );
```

```
cqlsh:practica> CREATE TABLE routes (
    route_id int,
    airline text,
    airline_id int,
    source_airport text,
    source_airport_id int,
    destination_airport text,
    destination_airport_id int,
    codeshare text,
    stops int,
    equipment text,
    PRIMARY KEY (route_id)
);
cqlsh:practica>
```

Figure 2.31: Creando la tabla routes en Cassandra

Ahora ya podemos guardar los datos en la tabla routes del keyspace practica de Cassandra:

```
1 df_routes.write.format("org.apache.spark.sql.cassandra").option("spark.
  cassandra.connection.host", "127.0.0.1").option("spark.cassandra.
  connection.port", "9042").option("keyspace", "practica").option("table",
  "routes").option("ttl", "1000").mode("append").save()
```

```
scala> df_routes.write.format("org.apache.spark.sql.cassandra").option("spark.cassandra.connection.host", "127.0.0.1").option("spark.cassandra.connection.port", "9042").option("keyspace", "practica").option("table", "routes").option("ttl", "1000").mode("append").save()
```

Figure 2.32: Guardando los datos en la tabla routes de Cassandra

Por último, mostraremos 5 filas de los datos en spark, almacenados en Cassandra:

```
1 spark.read.format("org.apache.spark.sql.cassandra").option("spark.
  cassandra.connection.host", "127.0.0.1").option("spark.cassandra.
  connection.port", "9042").option("keyspace", "practica").option("table",
  "routes").load().limit(5).show()
```

```
scala> spark.read.format("org.apache.spark.sql.cassandra").option("spark.cassandra.connection.host", "127.0.0.1").option("spark.cassandra.connection.port", "9042").option("keyspace", "practica").option("table", "routes").load().limit(5).show()
+-----+-----+-----+-----+-----+-----+-----+-----+
| route_id | airline | airline_id | codeshare | destination_airport | destination_airport_id | equipment | source_airport | stops |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 37968 | LH | 3320 | null | MUC | 346 | 319 | 321 | 320 | ADB | 1786 | 0 |
| 15391 | BE | 2470 | Y | EDL | 538 | 1 | 2WV | 538 | 0 |
| 53329 | WF | 5439 | null | EVE | 641 | DH3 | 770 | 655 | 0 |
| 17382 | CG | 1308 | null | MMH | 5431 | DHB | 71Z | 5433 | 0 |
| 66200 | YN | null | null | YKQ | 5507 | DHB | VKU | 5472 | 0 |
+-----+-----+-----+-----+-----+-----+-----+-----+
```

Figure 2.33: 5 filas de los datos en la tabla routes de Cassandra

Chapter 3

Análisis de datos ingestados en el lago de datos

En este apartado, se busca realizar una serie de consultas analíticas para demostrar al cliente cómo poder extraer información relevante para su caso de uso. Además, también se busca demostrar cómo persistir datos agregados en una nueva tabla para poder ser consultados de manera más rápida por herramientas de BI o de Dashboarding.

3.1 Consultas

3.1.1 Preparación de las consultas

Para poder ejecutar las consultas vamos a cargar los dataframes que hemos guardado en el lago de datos:

```
1 import java.util.Properties
2 val properties = new Properties()
3
4 properties.setProperty("user", "postgres")
5 properties.setProperty("driver", "org.postgresql.Driver")
6 properties.setProperty("password", "postgres")
7 properties.setProperty("partitionColumn", "altitude")
8 properties.setProperty("lowerBound", "0")
9 properties.setProperty("upperBound", "2000")
10 properties.setProperty("numPartitions", "10")
11 properties.setProperty("password", "postgres")
12
13 val df_airports = spark.read.option("partitionColumn", "altitude").
14     option("lowerBound", 0).option("upperBound", 2000).option(
15         "numPartitions", 10).jdbc("jdbc:postgresql://localhost:5432/postgres",
16         "airports", properties)
17
18 val df_airlines = spark.read.format("parquet").load("hdfs://localhost:
19 :9000/practica/airlines/")
```

```

16 val df_countries = spark.read.format("csv").option("header", "true").
  option("inferSchema", "true").load("hdfs://localhost:9000/practica/
  countries/")
17
18 val df_routes=spark.read.format("org.apache.spark.sql.cassandra").
  option("spark.cassandra.connection.host","127.0.0.1").option("spark
  .cassandra.connection.port","9042").option("keyspace", "practica").
  option("table", "routes").load()

```

Ahora pasaremos los dataframes a TempViews para poder realizar consultas SQL sobre ellos:

```

1 df_airports.createOrReplaceTempView("airports")
2 df_airlines.createOrReplaceTempView("airlines")
3 df_countries.createOrReplaceTempView("countries")
4 df_routes.createOrReplaceTempView("routes")

```

```

scala> df_airports.createOrReplaceTempView("airports")
scala> df_airlines.createOrReplaceTempView("airlines")
scala> df_countries.createOrReplaceTempView("countries")
scala> df_routes.createOrReplaceTempView("routes")

```

Figure 3.1: Creación de las vistas temporales de los dataframes

3.1.2 Consulta 1

La primera consulta que se nos pide es obtener los aeropuertos con mayor altitud. Para ello, se ha realizado la siguiente consulta:

```

1 spark.sql("SELECT * FROM airports ORDER BY altitude DESC LIMIT 1").show
  ()

```

| id | name | city country IATA ICAO | latitude | longitude | altitude timeZone DST | tz_database | tpe | sourc |
|------|----------------------|---|----------|-----------|-----------------------|-------------|------------|-------|
| 9310 | Daocheng Yading A... | Daocheng China DCY ZUOC 29.323055 106.05333 4410.850350502696 | 8.0 | N | Asia/Shanghai | airport | OurAirport | |

Figure 3.2: Resultado de la consulta de aeropuertos con mayor altitud

3.1.3 Consulta 2

La segunda consulta que se nos pide es obtener el número de aeropuertos que hay en España. Para ello, se ha realizado la siguiente consulta:

```

1 spark.sql("SELECT COUNT(country) AS airport_count FROM airports WHERE
  country == 'Spain'").show()

```

```
scala> spark.sql("SELECT COUNT(country) AS airport_count FROM airports WHERE country =='Spain'").show()
+-----+
| airport_count|
+-----+
|          64|
+-----+
```

Figure 3.3: Resultado de la consulta de aeropuertos en España

3.1.4 Consulta 3

La tercera consulta que se nos pide es obtener los países que tienen aeropuertos cuyo horario de verano sea el de Europa. Para ello, se ha realizado la siguiente consulta:

```
1 spark.sql("SELECT DISTINCT country FROM airports WHERE dst == 'E'").
    show()
```

```
scala> spark.sql("SELECT DISTINCT country FROM airports WHERE dst == 'E'" ).show()
+-----+
| country|
+-----+
| Russia|
| Sweden|
| Jersey|
| Turkey|
| Germany|
| Jordan|
| France|
| Greece|
| Slovakia|
| Belgium|
| Albania|
| Finland|
| Guernsey|
| United States|
| Belarus|
| Malta|
| Isle of Man|
| Croatia|
| Italy|
| Lithuania|
+-----+
only showing top 20 rows
```

Figure 3.4: Resultado de la consulta de países con aeropuertos en horario de verano de Europa

3.1.5 Consulta 4

La cuarta consulta que se nos pide es obtener el número de aerolíneas que tiene EEUU. Como habíamos particionado los datos por país, cargaremos los datos de las aerolíneas de EEUU y contaremos el número de filas:

```
1 val df_airlines_usa = spark.read.format("parquet").option("header", "true").
    option("inferSchema", "true").load("hdfs://localhost:9000/
    practica/airlines/country=United States")
2 df_airlines_usa.count()
```

```

scala> val df_airlines_usa = spark.read.format("parquet").option("header","true").option("inferSchema", "true").load("hdfs://localhost:9000/practica/airlines/country=United States")
df_airlines_usa: org.apache.spark.sql.DataFrame = [id: int, name: string ... 6 more fields]
scala> df_airlines_usa.count()
res44: Long = 1099

```

Figure 3.5: Resultado de la consulta de aerolíneas en EEUU con particionado por país

También se puede realizar la consulta directamente con SQL:

```

1 spark.sql("SELECT COUNT(*) AS airline_count FROM airlines WHERE country
    == 'United States').show()

```

```

scala> spark.sql("SELECT COUNT(*) AS airline_count FROM airlines WHERE country == 'United States').show()
+-----+
|airline_count|
+-----+
|      1099|
+-----+

```

Figure 3.6: Resultado de la consulta de aerolíneas en EEUU

3.1.6 Consulta 5

La quinta consulta que se nos pide es obtener los 10 países con más aerolíneas inactivas. Para ello, se ha realizado la siguiente consulta:

```

1 spark.sql("SELECT country, COUNT(*) AS inactive_airlines FROM airlines
    WHERE active == 'f' GROUP BY country ORDER BY inactive_airlines
    DESC LIMIT 10").show()

```

```

scala> spark.sql("SELECT country, COUNT(*) AS inactive_airlines FROM airlines WHERE active == 'f' GROUP BY country ORDER BY inactive_airlines
    DESC LIMIT 10").show()
+-----+-----+
|country|inactive_airlines|
+-----+-----+
| United States|        943|
|     Mexico|        427|
|United Kingdom|        369|
|     Canada|        201|
|     Russia|        158|
|     Spain|        142|
|     France|        98|
|     Germany|        97|
| South Africa|        81|
|     Nigeria|        80|
+-----+-----+

```

Figure 3.7: Resultado de la consulta de países con más aerolíneas inactivas

3.1.7 Consulta 6

La sexta y última consulta que se nos pide es obtener los países que tienen aerolíneas en activo y aeropuertos con una latitud mayor a 80. Para ello, se ha realizado la siguiente consulta:

```

1 spark.sql("SELECT DISTINCT a.country FROM airports a JOIN airlines b ON
    a.country = b.country WHERE a.latitude > 80 AND b.active == 't'").
    show()

```

```

scala> spark.sql("SELECT DISTINCT a.country FROM airports a JOIN airlines b ON a.country = b.country WHERE a.latitude > 80 AND b.active =
+-----+
| country|
+-----+
| Russia|
| Canada|
| New Zealand|
+-----+

```

Figure 3.8: Resultado de la consulta de países con aerolíneas activas y aeropuertos con latitud mayor a 80

3.2 Persistencia de datos agregados

Por último, también queremos mostrar al cliente la capacidad que tiene nuestra plataforma para guardar datos agregados en tablas nuevas para poder realizar informes o consultar datos específicos con menor procesamiento y menor latencia.

Para ello, se necesita crear una tabla llamada aggregations que guarde sus datos en formato parquet en la ruta /practica/aggregations/ de HDFS y que responda a la siguiente pregunta:

- ¿Cuántas rutas sin paradas (stops) a destinos con una altitud (altitude) mayor a 200 metros se hicieron por país (country) con aerolíneas que ya no están activas (active)?

Para ello, se ha realizado la siguiente consulta:

```

1 val df_aggregations = spark.sql("SELECT a.country, COUNT(*) AS
  routes_count FROM airports a JOIN routes r ON a.IATA = r.
  destination_airport JOIN airlines b ON a.country = b.country WHERE
  r.stops = 0 AND a.altitude > 200 AND b.active = 'f' GROUP BY a.
  country")

```

```

scala> val df_aggregations = spark.sql("SELECT a.country, COUNT(*) AS routes_count FROM airports a JOIN routes r ON a.IATA = r.destination_airport JOIN airlines b ON a.country = b.country WHERE r.stops = 0 AND a.altitude > 200 AND b.active = 'f' GROUP BY a.country")
df_aggregations: org.apache.spark.sql.DataFrame = [country: string, routes_count: bigint]
scala> df_aggregations.show()
+-----+-----+
| country|routes_count|
+-----+
| Russia| 65096|
| Sweden| 1485|
| Maldives| 35|
| Turkey| 3404|
| Germany| 66251|
| France| 27930|
| Greece| 260|
| Algeria| 260|
| Argentina| 952|
| Angola| 885|
| Peru| 720|
| India| 11952|
| China| 105165|
| United States| 4438617|
| Chile| 33841|
| Tajikistan| 490|
| Croatia| 8|
| Nigeria| 3440|
| Saudi Arabia| 570|
| Norway| 4347|
+-----+
only showing top 20 rows
scala>

```

Figure 3.9: Resultado de la consulta de rutas sin paradas a destinos con altitud mayor a 200 metros por país con aerolíneas inactivas

Ahora ya podemos guardar el dataframe en formato parquet en la ruta /practica/aggregations/ de HDFS:

```

1 df_aggregations.write.format("parquet").save("hdfs://localhost:9000/
  practica/aggregations/")

```

```
scala> df_aggregations.write.format("parquet").save("hdfs://localhost:9000/practica/aggregations/")
scala>
```

Figure 3.10: Guardado del dataframe en formato parquet en la ruta /practica/aggregations/ de HDFS

Ahora ya podemos consultar la tabla de agregaciones:

```
1 spark.read.format("parquet").load("hdfs://localhost:9000/practica/
    aggregations/").show()
```

```
scala> spark.read.format("parquet").load("hdfs://localhost:9000/practica/aggregations/").show()
+-----+-----+
| country|routes_count|
+-----+-----+
| Russia|      65096|
| Sweden|       1485|
| Malaysia|        35|
| Turkey|       3484|
| Germany|      66251|
| France|      27930|
| Greece|        266|
| Algeria|        260|
| Argentina|      952|
| Angola|       885|
| Peru|        720|
| Iran|       11523|
| China|      105165|
| United States| 4638617|
| Chile|       3384|
| Tajikistan|       490|
| Croatia|         8|
| Nigeria|      3440|
| Bolivia|       570|
| Norway|       4347|
+-----+-----+
only showing top 20 rows
scala>
```

Figure 3.11: Resultado de la consulta de la tabla de agregaciones