Chapter-2

June 22, 2020

1 A gentle introduction to Spark

1.1 Spark Architecture

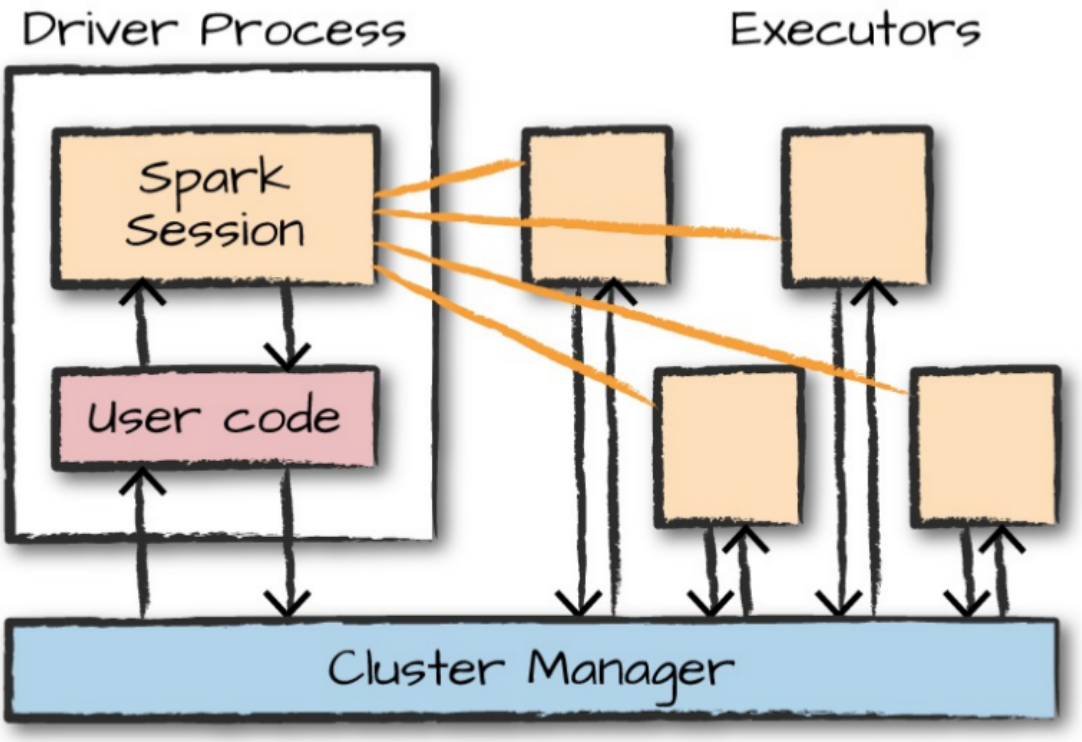

Figure 2-1. The architecture of a Spark Application

Each language API maintains the same core concepts that we described earlier. There is a **SparkSession** object available to the user, which is the entrance point to running Spark code. When using Spark from Python or R, JVM instructions aren't written explicity. The python, or R code is translated by Spark which can be run on the ececutor JVMs.

1.2 Starting Spark

Note: We can start interactive shell using pyspark but there is also a process of rsubmitting standalone applications to Spark called *spark-submit*

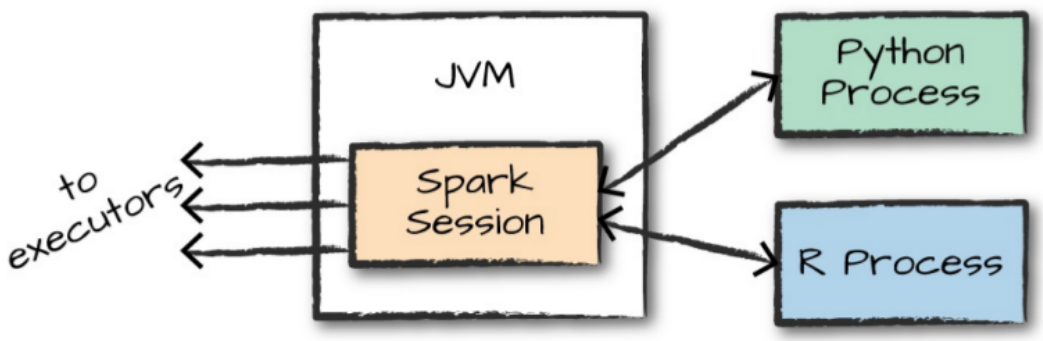

Figure 2-2. The relationship between the SparkSession and Spark's Language API

1.3 SparkSession

- User controls Spark Application through a driver process called the Spark Session
- The SparkSession instance is the way Spark executes user-defined manipulations across the cluster
- **There is a one-to-one correspondence between a SparkSession and a Spark Application**

[1]:
```
import pyspark
```

[2]:
```
from pyspark.sql import SparkSession
```

[3]:
```
spark = SparkSession.builder.appName("introduction").getOrCreate()
```

[4]:
```
spark
```

[4]:
```
<pyspark.sql.session.SparkSession at 0x7fed00dc7588>
```

1.3.1 Create Dataframe with one column containing 1000 rows with values from 0 to 999

[5]:
```
myRange = spark.range(1000)
myRange.show()
```

```
+---+
| id|
+---+
|  0|
|  1|
|  2|
|  3|
```

```
|  4|
|  5|
|  6|
|  7|
|  8|
|  9|
| 10|
| 11|
| 12|
| 13|
| 14|
| 15|
| 16|
| 17|
| 18|
| 19|
+---+
only showing top 20 rows
```

[6]:
```
myRange = myRange.toDF("number") # returns a new dataframe with specified␣
 ↪column name
myRange.show()
```

```
+------+
|number|
+------+
|     0|
|     1|
|     2|
|     3|
|     4|
|     5|
|     6|
|     7|
|     8|
|     9|
|    10|
|    11|
|    12|
|    13|
|    14|
|    15|
|    16|
|    17|
|    18|
|    19|
+------+
```

```
only showing top 20 rows
```

1.4 DataFrames

- Structured API and simply represents a table of data with rows and columns
- The list that defines the columns and the types within those columns is called the _schema-
- **Difference with spreadsheet**: a spreadsheet sits on one computer in one specific location, whereas a Spark DataFrame can span on thousands of computers.
- **Reason for putting data on multiple computers**: Either data is too large to fit or it would take too long to perform computation on it on a single machine

Note: Spark has several core abstractions: Datasets, DataFrames, SQL Tables, and Resilient Distributed Datasets(RDDs). These different abstractions all represent distributed collections of data

1.5 Partitions

- To allow every executor to perform work in parallel, Spark breaks up the data into chunks called **partition**.
- A partition is a collection of rows that sit on one physical machine in acluster
- A DataFrame's partitions represent how the data is physically distirbuted across the cluster of machines during execution
- **For parallelism, both multiple partitions and multiple executors are needed**
- **For DataFrame, we do not (for the most part) manipulate partitions manually or individually. We simply specify high-level transformations of data in the physical partitions, and Spark determines how this work will actually execute on the cluster.**

1.6 Transformations

- In Spark, the core data structures are immutable
- To "change" a DataFrame, we need to instruct Spark how it should be modified which are called **transformations**
- **NOTE:** Spark will not act on transformations until we call an action

[7]:
```
divisBy2 = myRange.where("number % 2 = 0") # Doesn't return output. This is␣
↪because we specified only an abstract transformation, and Spark will not act␣
↪on transformations until we call an action
divisBy2.show() # only executed here when show action is called
```

```
+------+
|number|
+------+
|     0|
|     2|
```

```
|     4|
|     6|
|     8|
|    10|
|    12|
|    14|
|    16|
|    18|
|    20|
|    22|
|    24|
|    26|
|    28|
|    30|
|    32|
|    34|
|    36|
|    38|
+------+
only showing top 20 rows
```

[8]:
```
divisBy2 = myRange.filter("number % 2 = 0")
```

[9]:
```
divisBy2.show()
```

```
+------+
|number|
+------+
|     0|
|     2|
|     4|
|     6|
|     8|
|    10|
|    12|
|    14|
|    16|
|    18|
|    20|
|    22|
|    24|
|    26|
|    28|
|    30|
|    32|
|    34|
|    36|
```

```
|    38|
+------+
only showing top 20 rows
```

- Transormations Types:
  - **Those that specify narrow dependencies**: For these, each input partition will contribute to only one output partition. Above example of "number % 2 = 0) is an example of narrow transformations
  - **Those that specify wide dependencies**: These will have input partitions contributing to many ouput partitions. This is often referred to as a **shuffle** whereby Spark will exchange partiions across the cluster.
- **NOTE:** With narrow transformations, Spark will automatically perform and operation called **piplining**, meaning that if we specify multiple filters on DataFrames, they'll all be performed in-memory. The same cannot be said for shuffles. When a shuffle is performed, Spark writes the results to disk

1.7 Lazy Evaluation

- It means that Spark will wait until the very last moment to execute the graph of computation instructions. In Spark, instead of modifying the data immediately when we express some operation, we build up a plan of transformations that you would like to apply to source data.
- By waiting until the last minute to execute the code, Spark compiles this paln from raw DataFrame transformations to a streamlined physical plan that will run as efficiently as possible across the cluster.
- **This provides immense benefits because Spark can optimize the entire data flow from end to end.**

Example Predicate Pushdown: If we build a large Spark job but specify a filter at the end that requires us to fetch one row from our source data, the most efficient way to execute this is to access the single record that we need. Spark will actually optimize this for us by pushning the filter down automatically.

1.8 Actions

- Transformation allow us to build up our logical transformation plan. **To trigger the computation we run an action which instructs Spark to compute a result from a series of transformation**

[10]:
```
divisBy2.count() # returns the number of rows which is an action
```

[10]: 500

1.8.1 Types of action

- Actions to view data in the console
- Actions to collect data to native objects in the respective language

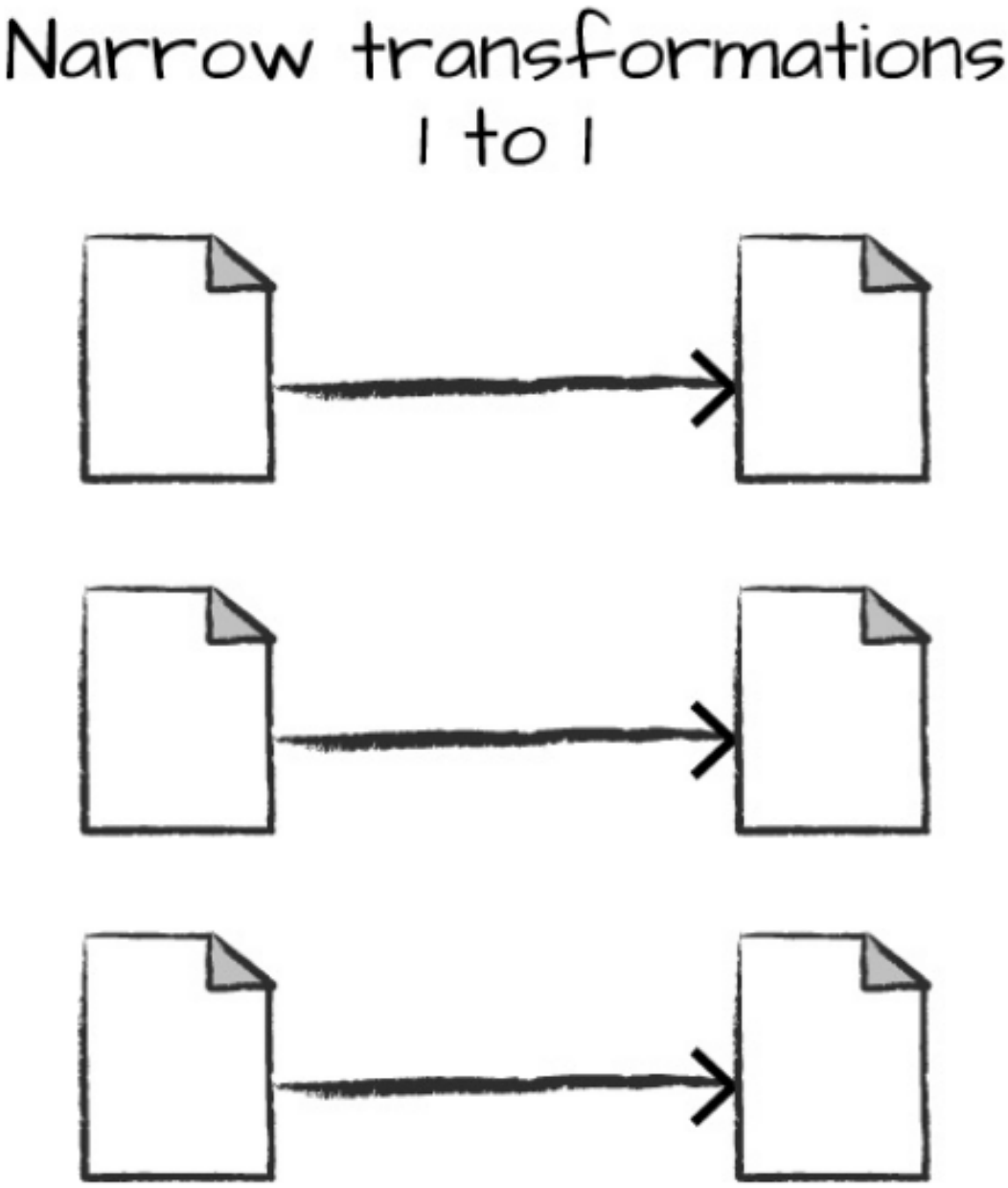

Figure 2-4. A narrow dependency

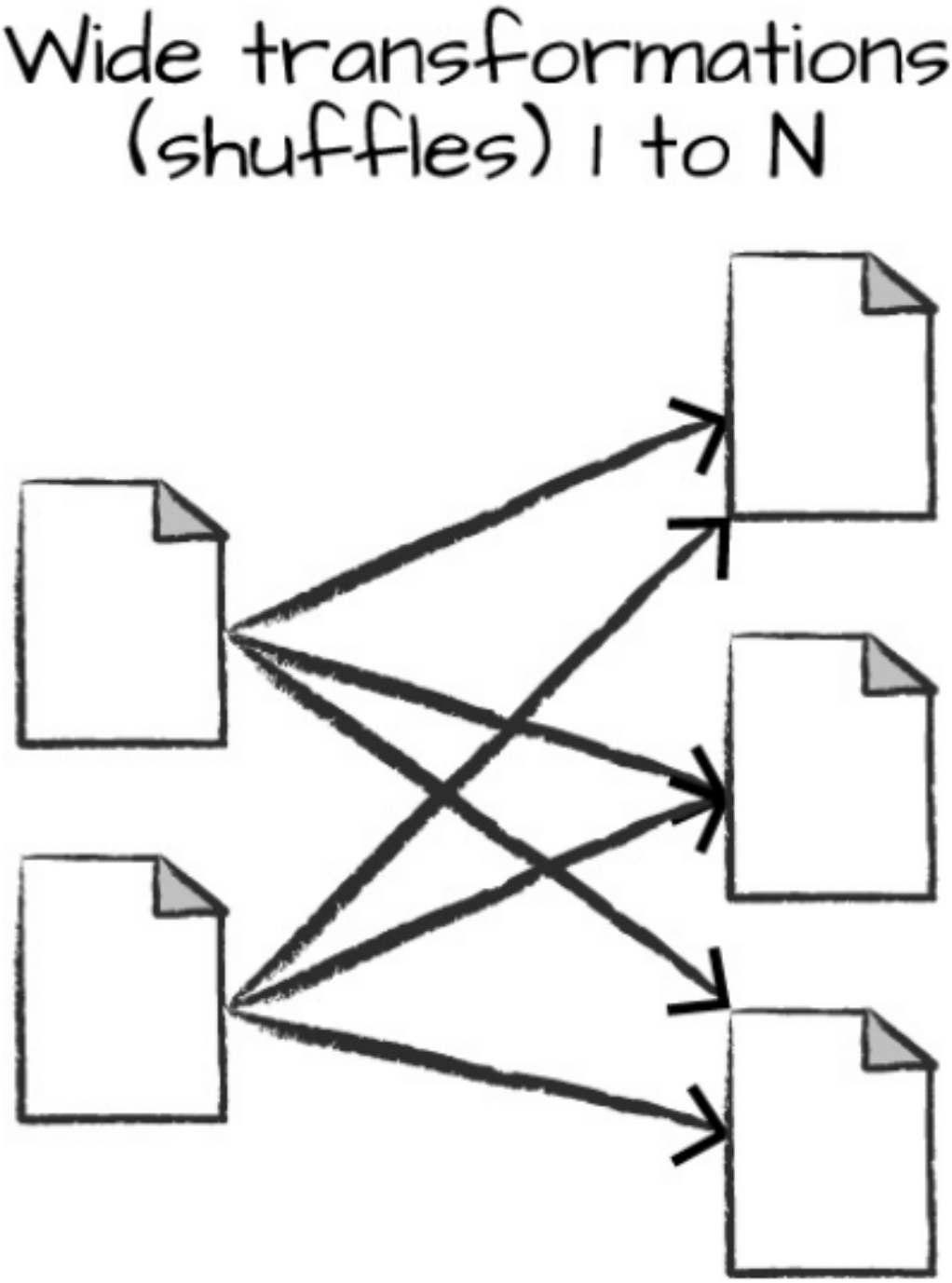

Figure 2-5. A wide dependency

- Actions to write to output data sources

1.9 Spark UI

- Runs at 4040 port of the driver node
- Runs at *http://localhost:4040* if running in local mode

1.10 An End-to-End Example

[13]:
```
file_path = "../data/flight-data/csv/2015-summary.csv"
```

[14]:
```
flightData2015 = spark.read.option("inferSchema","true").
↪option("header","true").csv(file_path)
```

This DataFrame have a set of columns with an unspecified number of rows. The reason the number of rows is unspecified is because reading data is a transformation, and is therefore a lazy operation. Spark peeked at only a couple of rows of data to try to guess what types each column should be.

Figure 2-7. Reading a CSV file into a DataFrame and converting it to a local array or list of rows

Take action

[15]:
```
flightData2015.take(3)
```

[15]:
```
[Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Romania',
count=15),
 Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Croatia', count=1),
 Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Ireland',
count=344)]
```

Sort

- It does not modify the DataFrame. It returns a new DataFrame by transforming the previous DataFrame
- Nothing happens to the data when we call sort because it's just a transformation.
- We can see that Spark is building up a plan for how it will execute this across the cluster by looking at the **explain** plan.

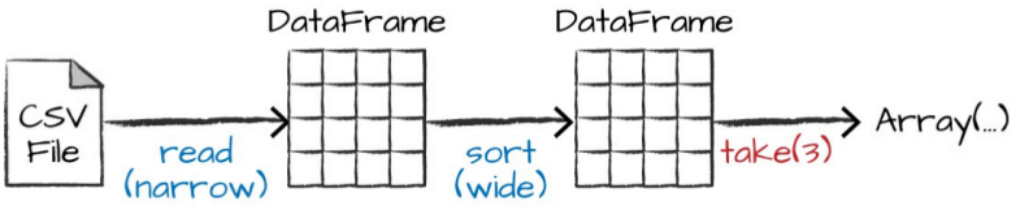

Figure 2-8. Reading, sorting, and collecting a DataFrame

[16]:
```
flightData2015.sort("count").explain()
```

```
== Physical Plan ==
*(1) Sort [count#48 ASC NULLS FIRST], true, 0
+- Exchange rangepartitioning(count#48 ASC NULLS FIRST, 200), true, [id=#113]
   +- FileScan csv [DEST_COUNTRY_NAME#46,ORIGIN_COUNTRY_NAME#47,count#48]
Batched: false, DataFilters: [], Format: CSV, Location:
InMemoryFileIndex[file:/home/raj/online-courses/pyspark/Spark-The-Definitive-
Guide/data/flight-da…, PartitionFilters: [], PushedFilters: [], ReadSchema:
struct<DEST_COUNTRY_NAME:string,ORIGIN_COUNTRY_NAME:string,count:int>
```

- We can read plans from top to bottom, **the top being the end result, and the bottom bein gthe source(s) of data.**
- By default, when we perform a shuffle **(wide_transformation)**, Sparks outputs **200 shuffle partitions.**

We can change this by setting a configuration

[17]:
```
spark.conf.set("spark.sql.shuffle.partitions","5")
```

[18]:
```
flightData2015.sort("count").explain()
# partitions changed to 5 unlike 200 in previous case
```

```
== Physical Plan ==
*(1) Sort [count#48 ASC NULLS FIRST], true, 0
+- Exchange rangepartitioning(count#48 ASC NULLS FIRST, 5), true, [id=#125]
   +- FileScan csv [DEST_COUNTRY_NAME#46,ORIGIN_COUNTRY_NAME#47,count#48]
Batched: false, DataFilters: [], Format: CSV, Location:
InMemoryFileIndex[file:/home/raj/online-courses/pyspark/Spark-The-Definitive-
Guide/data/flight-da…, PartitionFilters: [], PushedFilters: [], ReadSchema:
struct<DEST_COUNTRY_NAME:string,ORIGIN_COUNTRY_NAME:string,count:int>
```

[19]:
```
flightData2015.sort("count").take(2)
```

[19]:
```
[Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Singapore',
count=1),
```

```
Row(DEST_COUNTRY_NAME='Moldova', ORIGIN_COUNTRY_NAME='United States', count=1)]
```

Above whole process The data is partitioned on wide transformation(shuffle) which is sort.

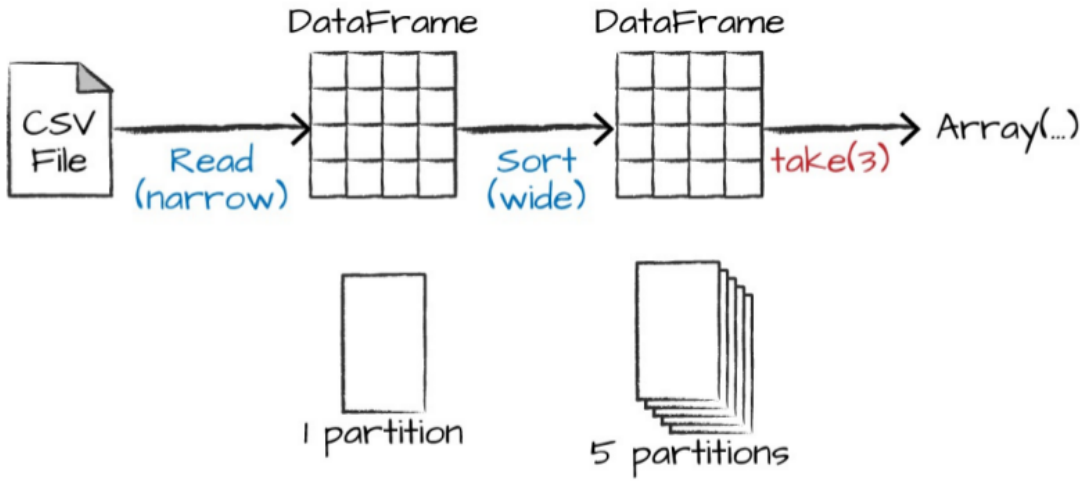

Figure 2-9. The process of logical and physical DataFrame manipulation

1.10.1 DataFrames and SQL

- We can specify logic in SQL or DataFrames and Spark will comple that logic down to an underlying plan before actually executing code
- With Spark SQL, we can **register any DataFrame as a table or view (a temporary table) and query it using pure SQL**
- There is no performance difference between writing SQL queries or writing DataFrame code, they both "compile" to the same underlying plan that we specify in DataFrame code.

Create Table or view

[20]:
```
flightData2015.createOrReplaceTempView("flight_data_2015")
```

Query data in SQL

- We can use spark.sql function that returns a new DataFrame
- A SQL query against a DataFrame returns another DataFrame which is actually quite powerful
- **This makes it possible to specify transformations in the manner most convenient to the user at any given point in time and not sacrifice any efficiency to do so**

[21]:
```
sqlWay = spark.sql("""
select DEST_COUNTRY_NAME, count(1)
from flight_data_2015
group by DEST_COUNTRY_NAME
""")
```

```
#COUNT(1) is basically just counting a constant value 1 column for each row
dataFrameWay = flightData2015.groupBy("DEST_COUNTRY_NAME").count()
```

[22]:
```
sqlWay.explain()
```

```
== Physical Plan ==
*(2) HashAggregate(keys=[DEST_COUNTRY_NAME#46], functions=[count(1)])
+- Exchange hashpartitioning(DEST_COUNTRY_NAME#46, 5), true, [id=#154]
   +- *(1) HashAggregate(keys=[DEST_COUNTRY_NAME#46],
functions=[partial_count(1)])
      +- FileScan csv [DEST_COUNTRY_NAME#46] Batched: false, DataFilters: [],
Format: CSV, Location: InMemoryFileIndex[file:/home/raj/online-
courses/pyspark/Spark-The-Definitive-Guide/data/flight-da…, PartitionFilters:
[], PushedFilters: [], ReadSchema: struct<DEST_COUNTRY_NAME:string>
```

[23]:
```
dataFrameWay.explain()
```

```
== Physical Plan ==
*(2) HashAggregate(keys=[DEST_COUNTRY_NAME#46], functions=[count(1)])
+- Exchange hashpartitioning(DEST_COUNTRY_NAME#46, 5), true, [id=#173]
   +- *(1) HashAggregate(keys=[DEST_COUNTRY_NAME#46],
functions=[partial_count(1)])
      +- FileScan csv [DEST_COUNTRY_NAME#46] Batched: false, DataFilters: [],
Format: CSV, Location: InMemoryFileIndex[file:/home/raj/online-
courses/pyspark/Spark-The-Definitive-Guide/data/flight-da…, PartitionFilters:
[], PushedFilters: [], ReadSchema: struct<DEST_COUNTRY_NAME:string>
```

[24]:
```
sqlWay.show()
```

```
+--------------------+--------+
|   DEST_COUNTRY_NAME|count(1)|
+--------------------+--------+
|             Moldova|       1|
|             Bolivia|       1|
|             Algeria|       1|
|Turks and Caicos …|       1|
|            Pakistan|       1|
|    Marshall Islands|       1|
|            Suriname|       1|
|              Panama|       1|
|         New Zealand|       1|
|             Liberia|       1|
|             Ireland|       1|
|              Zambia|       1|
```

```
|            Malaysia|       1|
|               Japan|       1|
|    French Polynesia|       1|
|           Singapore|       1|
|             Denmark|       1|
|               Spain|       1|
|             Bermuda|       1|
|            Kiribati|       1|
+--------------------+--------+
only showing top 20 rows
```

[25]:
```
dataFrameWay.show()
```

```
+--------------------+-----+
|   DEST_COUNTRY_NAME|count|
+--------------------+-----+
|             Moldova|    1|
|             Bolivia|    1|
|             Algeria|    1|
|Turks and Caicos …|    1|
|            Pakistan|    1|
|    Marshall Islands|    1|
|            Suriname|    1|
|              Panama|    1|
|         New Zealand|    1|
|             Liberia|    1|
|             Ireland|    1|
|              Zambia|    1|
|            Malaysia|    1|
|               Japan|    1|
|    French Polynesia|    1|
|           Singapore|    1|
|             Denmark|    1|
|               Spain|    1|
|             Bermuda|    1|
|            Kiribati|    1|
+--------------------+-----+
only showing top 20 rows
```

[26]:
```
spark.sql("""
select max(count)
from flight_data_2015""").take(1)
```

[26]:
```
[Row(max(count)=370002)]
```

[27]:
```
from pyspark.sql.functions import max
flightData2015.select(max('count').alias("count")).take(1)
```

[27]:
```
[Row(count=370002)]
```

[28]:
```
# Finding top 5 countries
# SQL Code
maxSql = spark.sql("""
select DEST_COUNTRY_NAME, sum(count) as destination_total
from flight_data_2015
group by DEST_COUNTRY_NAME
order by destination_total desc
limit 5""")
```

[29]:
```
maxSql.show()
```

```
+-----------------+-----------------+
|DEST_COUNTRY_NAME|destination_total|
+-----------------+-----------------+
|    United States|           411352|
|           Canada|             8399|
|           Mexico|             7140|
|   United Kingdom|             2025|
|            Japan|             1548|
+-----------------+-----------------+
```

[30]:
```
# Finding top 5 countries
# DataFrame code
from pyspark.sql.functions import desc
flightData2015\
.groupby("DEST_COUNTRY_NAME")\
.sum("count")\
.withColumnRenamed("sum(count)", "destination_total")\
.sort(desc("destination_total"))\
.limit(5)\
.show()
```

```
+-----------------+-----------------+
|DEST_COUNTRY_NAME|destination_total|
+-----------------+-----------------+
|    United States|           411352|
|           Canada|             8399|
|           Mexico|             7140|
|   United Kingdom|             2025|
|            Japan|             1548|
+-----------------+-----------------+
```

The execution plan is a directed acyclic graph(DAG) of transformations, each resulting in a new immutable DataFrame, on which we call an action to generate a result.

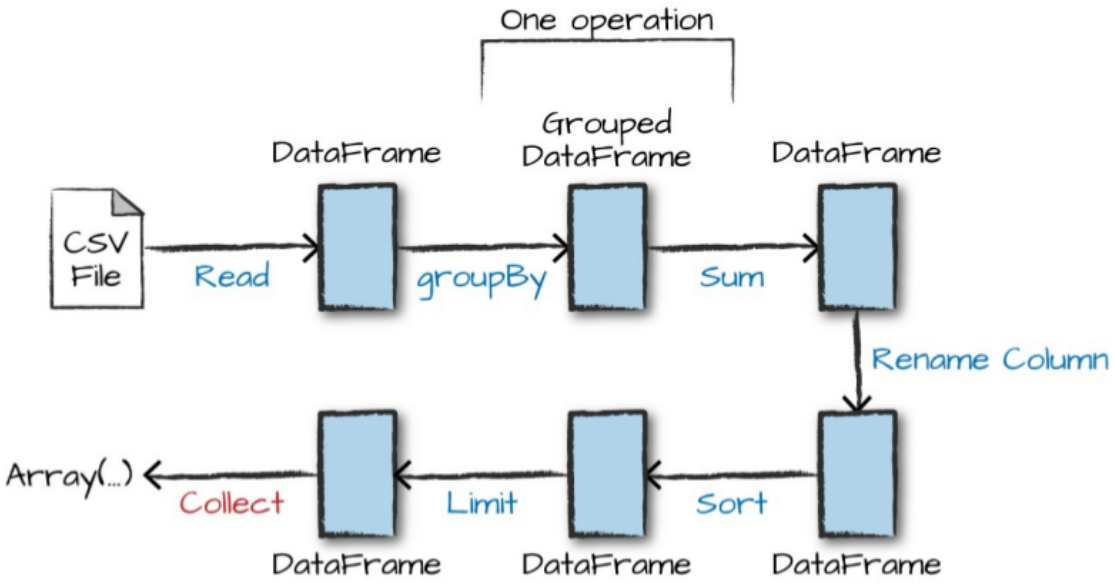

Figure 2-10. The entire DataFrame transformation flow

- **The first step is to read in the data. We defined the DataFrame previously but, as a reminder, Spark does not actually read it in until an action is calle don that DataFrame or one derived from the original DataFrame**
- In general, may DataFRame methods will accept strings (as column names) or Column types or expressions. Columns and expressions are actually the exact same thing.

[31]:
```
from pyspark.sql.functions import desc
flightData2015\
.groupby("DEST_COUNTRY_NAME")\
.sum("count")\
.withColumnRenamed("sum(count)", "destination_total")\
.sort(desc("destination_total"))\
.limit(5)\
.explain()
```

```
== Physical Plan ==
TakeOrderedAndProject(limit=5, orderBy=[destination_total#153L DESC NULLS LAST],
output=[DEST_COUNTRY_NAME#46,destination_total#153L])
+- *(2) HashAggregate(keys=[DEST_COUNTRY_NAME#46], functions=[sum(cast(count#48
as bigint))])
   +- Exchange hashpartitioning(DEST_COUNTRY_NAME#46, 5), true, [id=#368]
      +- *(1) HashAggregate(keys=[DEST_COUNTRY_NAME#46],
functions=[partial_sum(cast(count#48 as bigint))])
         +- FileScan csv [DEST_COUNTRY_NAME#46,count#48] Batched: false,
DataFilters: [], Format: CSV, Location: InMemoryFileIndex[file:/home/raj/online-
courses/pyspark/Spark-The-Definitive-Guide/data/flight-da…, PartitionFilters:
[], PushedFilters: [], ReadSchema: struct<DEST_COUNTRY_NAME:string,count:int>
```

[ ]:

[ ]: