Six Spark Exercises

Some challenging Spark SQL questions, easy to lift-and-shift on many real-world problems (with solutions)

Spark SQL is very easy to use, period. You might already know that it's also quite difficult to master. To be proficient in Spark, one must have three fundamental skills:

1. The ability to manipulate and understand the data
2. The knowledge on how to bend the tool to the programmer's needs
3. The art of finding a balance among the factors that affect Spark jobs executions

I crafted the following six exercises that will resemble some typical situations that Spark developers face daily when building their pipelines: these will help to assess the skills above.

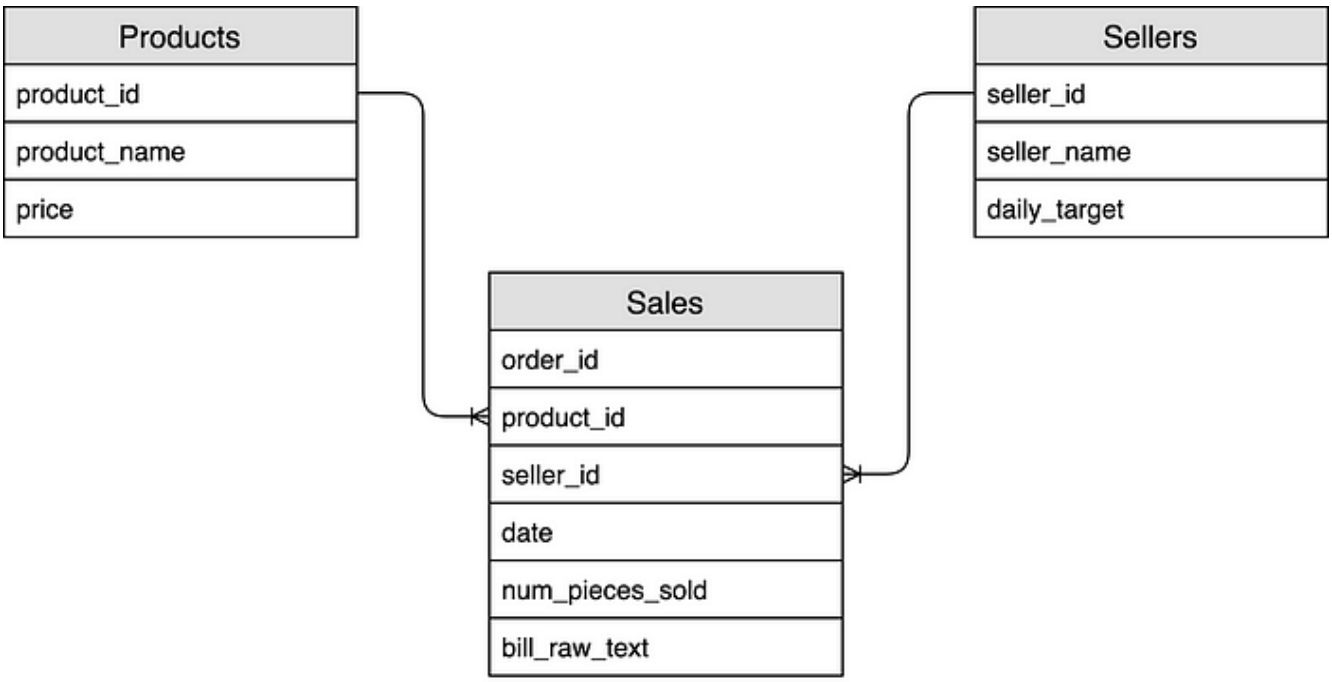
You can find the proposed solutions at the end of the article!

The Dataset

Let's describe briefly the dataset that we are going to use: it consists of three tables coming from the database of a shop, with products, sales and sellers. Data is available in Parquet files that you can download at the link below. Note that to exploit the exercises 100% you'll need to read from the files provided! (.zip, ~6GB, if you can't download the data, you can find the generator script clicking here)

DatasetToCompleteTheSixSparkExercises.zip
Dataset containing the three tables to do the six exercises
drive.google.com

The following diagram shows how the tables can be connected:



Sales Table

Each row in this table is an order and every order can contain only one product. Each row stores the following fields:

- order_id: The order ID
- product_id: The single product sold in the order. All orders have exactly one product)
- seller_id: The selling employee ID that sold the product
- num_pieces_sold: The number of units sold for the specific product in the order
- bill_raw_text: A string that represents the raw text of the bill associated with the order
- date: The date of the order.

Here's a sample of the table:

Search this file...

	order_id	product_id	seller_id	date	num_pieces_sold	bill_raw_text
1	1	0	0	2020-07-07	12	bdgwqyboczbitqzsxwmtjhehnxyvdfsddlftaiPmDbzqnta
2	2	0	0	2020-07-01	71	wmewueqbyoqivcpjfsmadtgxsvxjaretdoitcymxqnfWfby
3	3	0	0	2020-07-07	7	qhweohgmqkxxlzxzitjgbntpqswakfihqywypyzuijdmfcq
4	4	0	0	2020-07-07	85	bxwiyotlyldofwovdmyrzxujhlpgvjhwshlxpmhvjjylivlGfo
5	5	0	0	2020-07-09	53	qoSlyuuqmndmjleivcxijoqfcnftzaxuqkabwdbgcgmapu
6	6	0	0	2020-07-07	17	yvjkecsvwxmlmivopwbbiplmgkdmklzvmeibbrkvhgcxyh
7	7	0	0	2020-07-03	16	iqkfxtfmznbtrcqbqqmiepvpwfmqrkamlfceijbepfnmlcxy
8	8	0	0	2020-07-05	18	euwaghljdvrkwigxdwqvjegefiagyaygjemjpnxdsworxeh
9	9	0	0	2020-07-05	4	nqtyldjxhvkgxzptwvmlkucgdyoboyhzijxkqotgczutgnpc
10	10	0	0	2020-07-03	22	zoymbvgfcjrrrnvepxlwwezhkakaihxlllvnsremiealsgzwq
11	11	0	0	2020-07-02	78	tuvzrbtnmjtnxyrsrowmoxampugfqlalopxcfizxpmixiljcox
12	12	0	0	2020-07-08	42	hjrxxdfcnnzkurdxcvgmbvekchntuzpijxqeanqfvywsabqc
13	13	0	0	2020-07-02	65	dzccfnampspplsvnkfcchtskqlfpfkglbjzhqwnzvpdhiqxef

Products Table

Each row represents a distinct product. The fields are:

- product_id: The product ID
- product_name: The product name
- price: The product price

Search this file...

product_id	product_name	price
0	product_0	22
1	product_1	85
2	product_2	109
3	product_3	100
4	product_4	49
5	product_5	102
6	product_6	101
7	product_7	147
8	product_8	85
9	product_9	106
10	product_10	147
11	product_11	141
12	product_12	66
13	product_13	102

Sellers Table

This table contains the list of all the sellers:

- seller_id: The seller ID
- seller_name: The seller name
- daily_target: The number of items (regardless of the product type) that the seller needs to hit his/her quota. For example, if the daily target is 100,000, the employee needs to sell 100,000 products he can hit the quota by selling 100,000 units of product_0, but also selling 30,000 units of product_1 and 70,000 units of product_2

Search this file...

seller_id	seller_name	daily_target
0	seller_0	2500000
1	seller_1	1187414
2	seller_2	938318
3	seller_3	1322049
4	seller_4	1543722
5	seller_5	1476659
6	seller_6	51443
7	seller_7	492968
8	seller_8	437790
9	seller_9	1777256

Exercises

The best way to exploit the exercises below is to download the data and implement a working code that solves the proposed problems, ideally in a distributed environment! I would advise doing so before reading the solutions that are available at the end of the page!

Tip: I built the dataset to allow working on a single machine: when writing the code, imagine what would happen with a dataset 100 times bigger.

Even if you'd know how to solve them, my advice is not to skip the warm-up questions! (if you know Spark they'll take a few seconds).

If you are going to do the exercise with Python, you'll need the following packages:

```
# Pyspark
pip install pyspark# Pyspark stubs
pip install pyspark-stubs
```

Warm-up #1

Find out how many orders, how many products and how many sellers are in the data?

How many products have been sold at least once?

Which is the product contained in more orders?

Create the Spark session using the following code :

```
spark = SparkSession.builder \
    .master("local") \
    .config("spark.sql.autoBroadcastJoinThreshold", -1) \
    .config("spark.executor.memory", "500mb") \
    .appName("Exercise1") \
    .getOrCreate()
```

Warm-up #2

How many distinct products have been sold in each day?

Create the Spark session using the following code :

```
spark = SparkSession.builder \
    .master("local") \
    .config("spark.sql.autoBroadcastJoinThreshold", -1) \
    .config("spark.executor.memory", "500mb") \
    .appName("Exercise1") \
    .getOrCreate()
```

Exercise #1

What is the average revenue of the orders?

Create the Spark session using the following code :

```
spark = SparkSession.builder \
    .master("local") \
    .config("spark.sql.autoBroadcastJoinThreshold", -1) \
    .config("spark.executor.memory", "500mb") \
    .appName("Exercise1") \
    .getOrCreate()
```

Exercise #2

For each seller, what is the average % contribution of an order to the seller's daily quota?

 # Example

If Seller_0 with `quota=250` has 3 orders:

Order 1: 10 products sold
Order 2: 8 products sold
Order 3: 7 products sold

The average % contribution of orders to the seller's quota would be:

Order 1: 10/105 = 0.04
Order 2: 8/105 = 0.032
Order 3: 7/105 = 0.028

Average % Contribution = (0.04+0.032+0.028)/3 = 0.03333

Create the Spark session using the following code :

```
spark = SparkSession.builder \
    .master("local") \
    .config("spark.sql.autoBroadcastJoinThreshold", -1) \
    .config("spark.executor.memory", "500mb") \
    .appName("Exercise1") \
    .getOrCreate()
```

Exercise #3

Who are the second most selling and the least selling persons (sellers) for each product?

Who are those for product with `product_id = 0`

Create the Spark session using the following code :

```
spark = SparkSession.builder \
    .master("local") \
    .config("spark.sql.autoBroadcastJoinThreshold", -1) \
    .config("spark.executor.memory", "3gb") \
    .appName("Exercise1") \
    .getOrCreate()
```

Exercise #4

Create a new column called "hashed_bill" defined as follows:

- **if the order_id is even:** apply MD5 hashing iteratively to the bill_raw_text field, once for each 'A' (capital 'A') present in the text. E.g. if the bill text is 'nbAAnllA', you would apply hashing three times iteratively (only if the order number is even)
- **if the order_id is odd:** apply SHA256 hashing to the bill text

Finally, check if there are any duplicate on the new column

Create the Spark session using the following code :

```
spark = SparkSession.builder \
    .master("local") \
    .config("spark.sql.autoBroadcastJoinThreshold", -1) \
    .config("spark.executor.memory", "3gb") \
    .appName("Exercise1") \
    .getOrCreate()
```

Solutions

Let's dive into the solutions. First, you should have noted that the warm-up questions are handy to solve the exercises:

Warm-up #1

The solution to this exercise is quite easy. First, we simply need to `count` how many rows we have in every dataset:

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import *

#   Initialize the Spark session
spark = SparkSession.builder \
    .master("local") \
    .config("spark.sql.autoBroadcastJoinThreshold", -1) \
    .config("spark.executor.memory", "500mb") \
    .appName("Exercise1") \
    .getOrCreate()

#   Read the source tables in Parquet format
products_table = spark.read.parquet("./data/products_parquet")
sales_table = spark.read.parquet("./data/sales_parquet")
sellers_table = spark.read.parquet("./data/sellers_parquet")

#   Print the number of orders
print("Number of Orders: {}".format(sales_table.count()))

#   Print the number of sellers
print("Number of sellers: {}".format(sellers_table.count()))
```

```
#   Print the number of products
print("Number of products: {}".format(products_table.count()))
```

We get the following output:

```
Number of Orders: 20000040
Number of sellers: 10
Number of products: 75000000
```

As you can see, we have 75,000,000 products in our dataset and 20,000,040 orders: since each order can only have a single product, some of them have never been sold. Let's find out how many products appear at least once and which is the product contained in more orders:

```
#   Output how many products have been actually sold at least once
print("Number of products sold at least once")
sales_table.agg(countDistinct(col("product_id"))).show()

#   Output which is the product that has been sold in more orders
print("Product present in more orders")
sales_table.groupBy(col("product_id")).agg(
    count("*").alias("cnt")).orderBy(col("cnt").desc()).limit(1).show()
```

The first query is counting how many distinct products we have in the sales table, while the second block is pulling the `product_id` that has the highest count in the sales table.

The output is the following:

```
Number of products sold at least once
+-------------------------+
|count(DISTINCT product_id)|
+-------------------------+
|                   993429|
+-------------------------+
Product present in more orders
+----------+--------+
|product_id|     cnt|
+----------+--------+
|         0|19000000|
+----------+--------+
```

Let's have a closer look at the second result: 19,000,000 orders out of 20 M are selling the product with `product_id = 0`: this is a powerful information that we should use later!

Warm-up #2

Having some knowledge of Spark this should be straightforward: we simply need to find out "how many distinct products have been sold in each date":

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import *

# Create Spark session
spark = SparkSession.builder \
    .master("local") \
    .config("spark.sql.autoBroadcastJoinThreshold", -1) \
```

```
    .config("spark.executor.memory", "500mb") \
    .appName("Exercise1") \
    .getOrCreate()

# Read Source tables
products_table = spark.read.parquet("./data/products_parquet")
sales_table = spark.read.parquet("./data/sales_parquet")
sellers_table = spark.read.parquet("./data/sellers_parquet")

sales_table.groupby(col("date")).agg(countDistinct(col("product_id")).alias("distinct
_products_sold")).orderBy(
    col("distinct_products_sold").desc()).show()
```

Nothing much to say here, the output is the following:

```
+----------+----------------------+
|      date|distinct_products_sold|
+----------+----------------------+
|2020-07-06|                100765|
|2020-07-09|                100501|
|2020-07-01|                100337|
|2020-07-03|                100017|
|2020-07-02|                 99807|
|2020-07-05|                 99796|
|2020-07-04|                 99791|
|2020-07-07|                 99756|
|2020-07-08|                 99662|
|2020-07-10|                 98973|
+----------+----------------------+
```

Exercise #1

Let's work out the hard stuff! The first exercise is simply asking "What is the average revenue of the orders?"

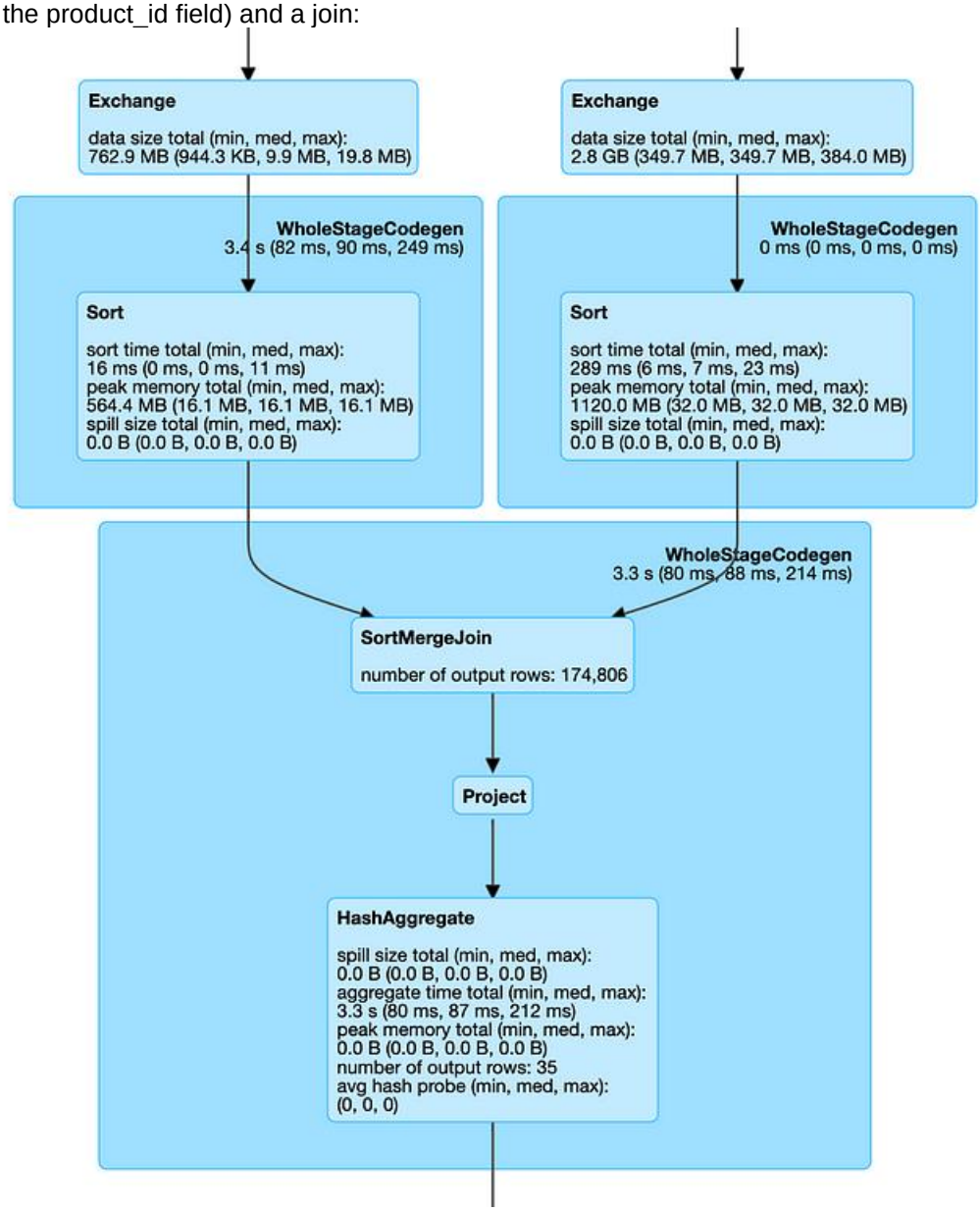
In theory, this is simple: we first need to calculate the revenue for each order and then get the average. Remember that `revenue = price * quantity`. Pretty easy: the `product_price` is in the products table, while the amount is in the sales table.

A first approach could be to simply join the two tables, create a new column and do the average:

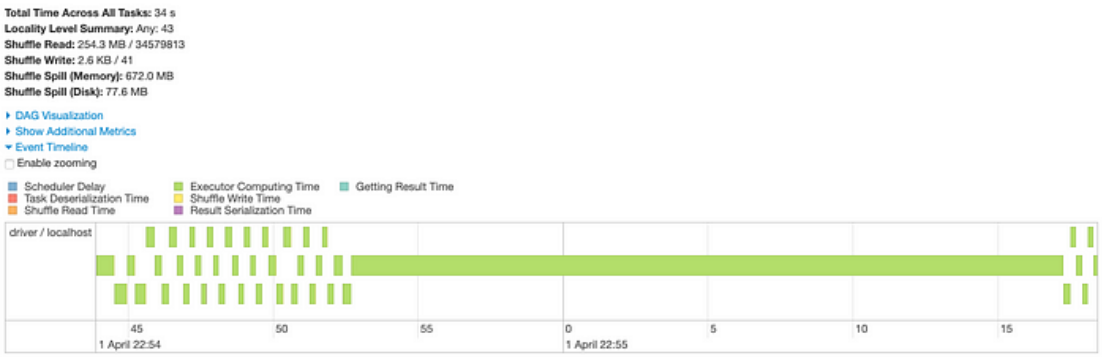
```
#  What is the average revenue of the orders?

from pyspark.sql import SparkSession
from pyspark.sql.functions import *
from pyspark.sql import Row
from pyspark.sql.types import IntegerType

# Create the Spark session
spark = SparkSession.builder \
    .master("local") \
    .config("spark.sql.autoBroadcastJoinThreshold", -1) \
    .config("spark.executor.memory", "500mb") \
    .appName("Exercise1") \
```

```
    .getOrCreate()

# Read the source tables
products_table = spark.read.parquet("./data/products_parquet")
sales_table = spark.read.parquet("./data/sales_parquet")
sellers_table = spark.read.parquet("./data/sellers_parquet")

# Do the join and print the results
print(sales_table.join(products_table, sales_table["product_id"] ==
products_table["product_id"], "inner").
      agg(avg(products_table["price"] * sales_table["num_pieces_sold"])).show())
```

The above is correct, and it probably works quite well (especially if you are working on a local environment). But let's have a look at the execution plan DAG: at some point we will have a repartitioning (on the product_id field) and a join:



Let's see what happens when Spark performs the join (on the Spark UI):



Oops! One task is taking much more time than the others!
This is a typical case of a skewed join, where one task takes a long time to execute since the join is skewed on a very small number of keys (in this case, product_id = 0).

I covered Spark joins in my Medium article, “The Art of Joining in Spark”, if you want to know more about it you could have a look there!

Note that this is not a huge problem in case you are running Spark on a local system. On a distributed environment (and with more data), though, this join could take an incredible amount of time to complete (maybe never complete at all!).

Let’s fix this issue using a technique known as “key salting”. I won’t describe in detail since I have already covered the topic in the article linked above. As a summary, what we are going to do is the following:

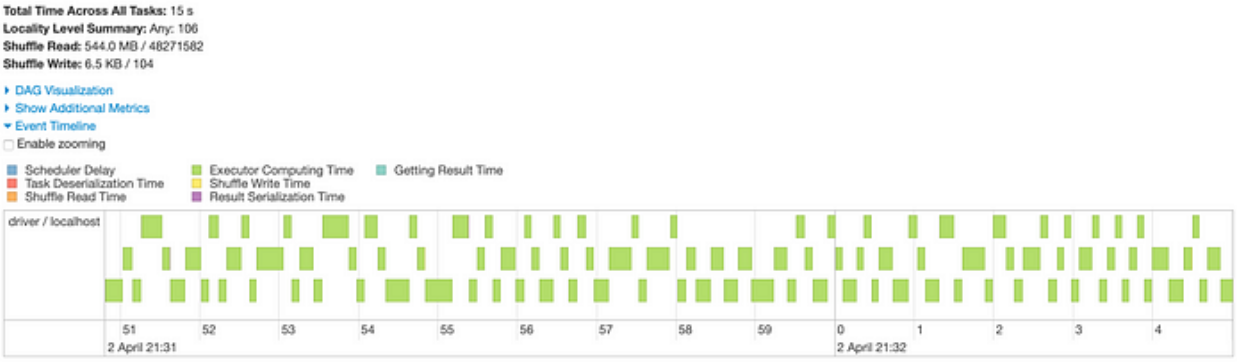
1. Duplicate the entries that we have in the dimension table for the most common products, e.g. product_0 will be replicated creating the IDs: product_0–1, product_0–2, product_0–3 and so on.
2. On the sales table, we are going to replace “product_0” with a random replica (e.g. some of them will be replaced with product_0–1, others with product_0–2, etc.) Using the new “salted” key will un-skew the join:

The important thing to observe here is that we are NOT salting ALL the products, but only those that drive skewness (in the example we are getting the 100 most frequent products). Salting the whole dataset would be problematic since the number of rows would grow linearly on the “salting factor”:

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import *
from pyspark.sql import Row
from pyspark.sql.types import IntegerType

# Create the Spark session
spark = SparkSession.builder \
    .master("local") \
    .config("spark.sql.autoBroadcastJoinThreshold", -1) \
    .config("spark.executor.memory", "500mb") \
    .appName("Exercise1") \
    .getOrCreate()

# Read the source tables
products_table = spark.read.parquet("./data/products_parquet")
sales_table = spark.read.parquet("./data/sales_parquet")
sellers_table = spark.read.parquet("./data/sellers_parquet")

# Step 1 - Check and select the skewed keys
```

```
# In this case we are retrieving the top 100 keys: these will be the only
salted keys.
results =
sales_table.groupby(sales_table["product_id"]).count().sort(col("count").desc
()).limit(100).collect()

# Step 2 - What we want to do is:
#  a. Duplicate the entries that we have in the dimension table for the most
common products, e.g.
#       product_0 will become: product_0-1, product_0-2, product_0-3 and so
on
#  b. On the sales table, we are going to replace "product_0" with a random
duplicate (e.g. some of them
#     will be replaced with product_0-1, others with product_0-2, etc.)
# Using the new "salted" key will unskew the join

# Let's create a dataset to do the trick
REPLICATION_FACTOR = 101
l = []
replicated_products = []
for _r in results:
    replicated_products.append(_r["product_id"])
    for _rep in range(0, REPLICATION_FACTOR):
        l.append((_r["product_id"], _rep))
rdd = spark.sparkContext.parallelize(l)
replicated_df = rdd.map(lambda x: Row(product_id=x[0],
replication=int(x[1])))
replicated_df = spark.createDataFrame(replicated_df)

#   Step 3: Generate the salted key
products_table = products_table.join(broadcast(replicated_df),
                                     products_table["product_id"] ==
replicated_df["product_id"], "left"). \
    withColumn("salted_join_key", when(replicated_df["replication"].isNull(),
products_table["product_id"]).otherwise(
    concat(replicated_df["product_id"], lit("-"),
replicated_df["replication"])))

sales_table = sales_table.withColumn("salted_join_key",
when(sales_table["product_id"].isin(replicated_products),

concat(sales_table["product_id"], lit("-"),

round(rand() * (REPLICATION_FACTOR - 1), 0).cast(

IntegerType()))).otherwise(
    sales_table["product_id"]))

#   Step 4: Finally let's do the join
print(sales_table.join(products_table, sales_table["salted_join_key"] ==
products_table["salted_join_key"],
                       "inner").
      agg(avg(products_table["price"] *
sales_table["num_pieces_sold"])).show())

print("Ok")
```

Looking at the stages when we execute the above:



The result of the query should be the following

```
+------------------------------+
|avg((price * num_pieces_sold))|
+------------------------------+
|            1246.1338560822878|
+------------------------------+
```

Using this technique in a local environment could lead to an increase of the execution time; in the real world, though, this trick can make the difference between completing and not completing the join.

Exercise #2

Question number two was: “for each seller, what is the average % contribution of an order to the sellers’ daily quota?”.

This is similar to the first exercise: we can join our table with the sellers table, we calculate the percentage of the quota hit thanks to a specific order and we do the average, grouping by the seller_id.

Again, this could generate a skewed join, since even the sellers are not evenly distributed. In this case, though, the solution is much simpler! Since the sellers table is very small, we can broadcast it, making the operations much much faster!

“Broadcasting” simply means that a copy of the table is sent to every executor, allowing to “localize” the task. We need to use this operator carefully: when we broadcast a table, we need to be sure that this will not become too-big-to-broadcast in the future, otherwise we’ll start to have Out Of Memory errors later in time (as the broadcast dataset gets bigger).

```
#   For each seller find the average % of the target amount brought by each
order

from pyspark.sql import SparkSession
from pyspark.sql.functions import *
from pyspark.sql import Row
from pyspark.sql.types import IntegerType

# Create the Spark session
spark = SparkSession.builder \
    .master("local") \
```

```
    .config("spark.sql.autoBroadcastJoinThreshold", -1) \
    .config("spark.executor.memory", "3g") \
    .appName("Exercise1") \
    .getOrCreate()

# Read the source tables
products_table = spark.read.parquet("./data/products_parquet")
sales_table = spark.read.parquet("./data/sales_parquet")
sellers_table = spark.read.parquet("./data/sellers_parquet")

#   Wrong way to do this - Skewed
#   (Note that Spark will probably broadcast the table anyway, unless we
forbid it throug the configuration paramters)
print(sales_table.join(sellers_table, sales_table["seller_id"] ==
sellers_table["seller_id"], "inner").withColumn(
    "ratio", sales_table["num_pieces_sold"]/sellers_table["daily_target"]
).groupBy(sales_table["seller_id"]).agg(avg("ratio")).show())

#   Correct way through broarcasting
print(sales_table.join(broadcast(sellers_table), sales_table["seller_id"] ==
sellers_table["seller_id"], "inner").withColumn(
    "ratio", sales_table["num_pieces_sold"]/sellers_table["daily_target"]
).groupBy(sales_table["seller_id"]).agg(avg("ratio")).show())
```

Exercise #3

Question: “Who are the second most selling and the least selling persons (sellers) for each product? Who are those for the product with product_id = 0”.

This sounds like window functions! Let’s analyze the question: for each product, we need the second most selling and the least selling employees (sellers): we are probably going to need two rankings, one to get the second and the other one to get the last in the sales chart. We also need to handle some edge cases:

- If a product has been sold by only one seller, we’ll put it into a special category (category: Only seller or multiple sellers with the same quantity).
- If a product has been sold by more than one seller, but all of them sold the same quantity, we are going to put them in the same category as if they were only a single seller for that product (category: Only seller or multiple sellers with the same quantity).
- If the “least selling” is also the “second selling”, we will count it only as “second seller”

Let’s draft a strategy:

- We get the sum of sales for each product and seller pairs.
- We add two new ranking columns: one that ranks the products’ sales in descending order and another one that ranks in ascending order.
- We split the dataset obtained in three pieces: one for each case that we want to handle (second top selling, least selling, single selling).

1. When calculating the “least selling”, we exclude those products that have a single seller and those where the least selling employee is also the second most selling
2. We merge the pieces back together.

```
from pyspark.sql import SparkSession
```

```
from pyspark.sql.functions import *
from pyspark.sql import Row, Window
from pyspark.sql.types import IntegerType

spark = SparkSession.builder \
    .master("local") \
    .config("spark.sql.autoBroadcastJoinThreshold", -1) \
    .config("spark.executor.memory", "3g") \
    .appName("Exercise1") \
    .getOrCreate()

# Read the source tables
products_table = spark.read.parquet("./data/products_parquet")
sales_table = spark.read.parquet("./data/sales_parquet")
sellers_table = spark.read.parquet("./data/sellers_parquet")

# Calcuate the number of pieces sold by each seller for each product
sales_table = sales_table.groupby(col("product_id"), col("seller_id")). \
    agg(sum("num_pieces_sold").alias("num_pieces_sold"))

# Create the window functions, one will sort ascending the other one descending.
Partition by the product_id
# and sort by the pieces sold
window_desc =
Window.partitionBy(col("product_id")).orderBy(col("num_pieces_sold").desc())
window_asc =
Window.partitionBy(col("product_id")).orderBy(col("num_pieces_sold").asc())

# Create a Dense Rank (to avoid holes)
sales_table = sales_table.withColumn("rank_asc", dense_rank().over(window_asc)). \
    withColumn("rank_desc", dense_rank().over(window_desc))

# Get products that only have one row OR the products in which multiple sellers sold
the same amount
# (i.e. all the employees that ever sold the product, sold the same exact amount)
single_seller = sales_table.where(col("rank_asc") == col("rank_desc")).select(
    col("product_id").alias("single_seller_product_id"),
col("seller_id").alias("single_seller_seller_id"),
    lit("Only seller or multiple sellers with the same results").alias("type")
)

# Get the second top sellers
second_seller = sales_table.where(col("rank_desc") == 2).select(
    col("product_id").alias("second_seller_product_id"),
col("seller_id").alias("second_seller_seller_id"),
    lit("Second top seller").alias("type")
)

# Get the least sellers and exclude those rows that are already included in the first
piece
# We also exclude the "second top sellers" that are also "least sellers"
least_seller = sales_table.where(col("rank_asc") == 1).select(
    col("product_id"), col("seller_id"),
    lit("Least Seller").alias("type")
```

```
).join(single_seller, (sales_table["seller_id"] ==
single_seller["single_seller_seller_id"]) & (
        sales_table["product_id"] == single_seller["single_seller_product_id"]),
"left_anti"). \
    join(second_seller, (sales_table["seller_id"] ==
second_seller["second_seller_seller_id"]) & (
        sales_table["product_id"] == second_seller["second_seller_product_id"]),
"left_anti")

# Union all the pieces
union_table = least_seller.select(
    col("product_id"),
    col("seller_id"),
    col("type")
).union(second_seller.select(
    col("second_seller_product_id").alias("product_id"),
    col("second_seller_seller_id").alias("seller_id"),
    col("type")
)).union(single_seller.select(
    col("single_seller_product_id").alias("product_id"),
    col("single_seller_seller_id").alias("seller_id"),
    col("type")
))
union_table.show()

# Which are the second top seller and least seller of product 0?
union_table.where(col("product_id") == 0).show()
```

The output for the second part of the question is the following:

```
+----------+---------+--------------------+
|product_id|seller_id|                type|
+----------+---------+--------------------+
|         0|        0|Only seller or mu...|
+----------+---------+--------------------+
```

Exercise #4

For this final exercise, we simply need to apply a fancy algorithm. We can do that through UDFs (User Defined Functions). A UDF is a custom function that can be invoked on datafarmes columns; as a rule of thumb, we should usually try to avoid UDFs, since Spark is not really capable to optimize them: UDF code usually runs slower than the non-UDF counterpart. Unfortunately, we cannot apply the algorithm described just using Spark SQL functions.

The solution is something like the following:

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import *
from pyspark.sql import Row, Window
from pyspark.sql.types import IntegerType
import hashlib

#   Init spark session
spark = SparkSession.builder \
    .master("local") \
```

```
    .config("spark.sql.autoBroadcastJoinThreshold", -1) \
    .config("spark.executor.memory", "1g") \
    .appName("Exercise1") \
    .getOrCreate()

#   Load source data
products_table = spark.read.parquet("./data/products_parquet")
sales_table = spark.read.parquet("./data/sales_parquet")
sellers_table = spark.read.parquet("./data/sellers_parquet")

#   Define the UDF function
def algo(order_id, bill_text):
    #   If number is even
    ret = bill_text.encode("utf-8")
    if int(order_id) % 2 == 0:
        #   Count number of 'A'
        cnt_A = bill_text.count("A")
        for _c in range(0, cnt_A):
            ret = hashlib.md5(ret).hexdigest().encode("utf-8")
        ret = ret.decode('utf-8')
    else:
        ret = hashlib.sha256(ret).hexdigest()
    return ret

#   Register the UDF function.
algo_udf = spark.udf.register("algo", algo)

#   Use the `algo_udf` to apply the aglorithm and then check if there is any
duplicate hash in the table
sales_table.withColumn("hashed_bill", algo_udf(col("order_id"),
col("bill_raw_text")))\

.groupby(col("hashed_bill")).agg(count("*").alias("cnt")).where(col("cnt") >
1).show()
```

First, we need to define the UDF function: def algo(order_id, bill_text). The algo function receives the order_id and the bill_text as input.

The UDF function implements the algorithm:

1. Check if the order_id is even or odd.
2. If order_id is even, count the number of capital ‘A’ in the bill text and iteratively apply MD5
3. If order_id is odd, apply SHA256
4. Return the hashed string

Afterward, this function needs to be registered in the Spark Session through the line algo_udf = spark.udf.register(“algo”, algo). The first parameter is the name of the function within the Spark context while the second parameter is the actual function that will be executed.

We apply the UDF at the following line:

```
sales_table.withColumn("hashed_bill", algo_udf(col("order_id"),
col("bill_raw_text")))
```

As you can see, the function takes two columns as input and it will be executed for each row (i.e. for each pair of order_id and bill_raw_text).

In the final dataset, all the hashes should be different, so the query should return an empty dataset

Take Away

If you completed all the exercises, congratulations! Those covered some very important topics about Spark SQL development:

- Joins Skewness: This is usually the main pain point in Spark pipelines; sometimes it is very difficult to solve, because it's not easy to find a balance among all the factors that are involved in these operations.

- Window functions: Super useful, the only thing to remember is to first define the windowing.

- UDFs: Although they are very helpful, we should think twice before jumping into the development of such functions, since their execution might slow down our code.

Of course, the exercises above could be solved in many different ways,