

# Apache Spark Exercise



**Version:** 1.0.0

**Date:** June 2022

---

# Content

## Table of Contents

Objective.....	4
The Dataset .....	4
Sales Table.....	4
Products Table .....	5
Sellers Table.....	6
Exercises .....	6
Warm-up #1 .....	6
Warm-up #2 .....	7
Exercise #1 .....	7
Exercise #2 .....	7
Exercise #3 .....	8
Exercise #4 .....	8
Solutions .....	9
Warm-up #1 .....	9
Warm-up #2 .....	9
Exercise #1.....	11
Exercise #2 .....	15
Exercise #3 .....	16
Exercise #4 .....	17
Take Away.....	19

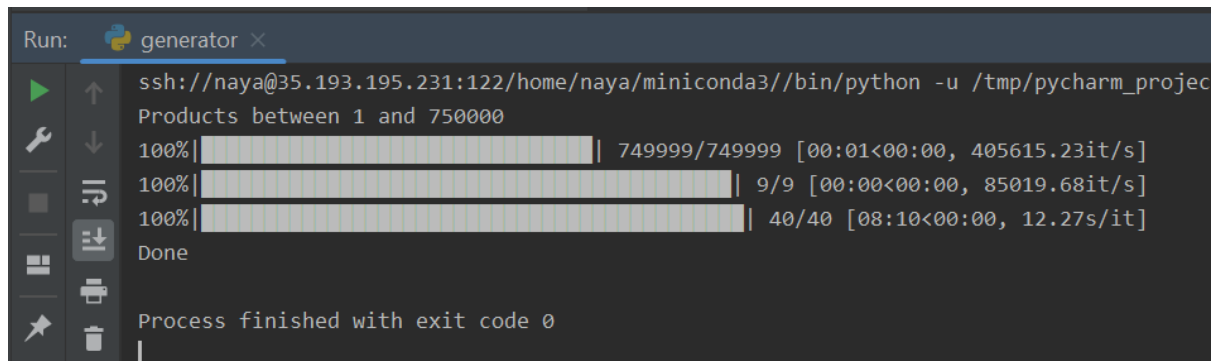
---

## Objective

This lab includes some challenging Spark questions, easy to lift-and-shift on many real-world problems and will resemble some typical situations that Spark developers face daily when building their pipelines. You can find the proposed solutions at the end of the lab!

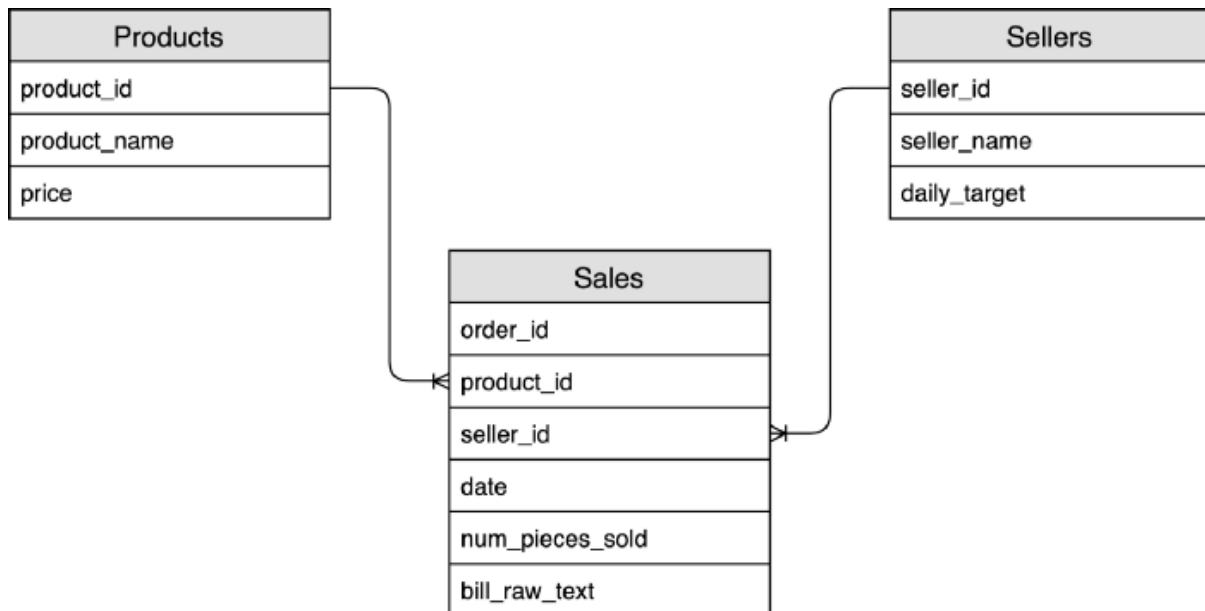
## The Dataset

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. In order to download the data, you can find the generator script clicking [here](#))



```
Run: generator x
ssh://naya@35.193.195.231:122/home/naya/miniconda3/bin/python -u /tmp/pycharm_projec
Products between 1 and 750000
100%|██████████████████████████████████████████████████████████████████████████████| 749999/749999 [00:01<00:00, 405615.23it/s]
100%|██████████████████████████████████████████████████████████████████████████████| 9/9 [00:00<00:00, 85019.68it/s]
100%|██████████████████████████████████████████████████████████████████████████████| 40/40 [08:10<00:00, 12.27s/it]
Done
Process finished with exit code 0
```

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:

rder_id	product_id	seller_id	date	num_pieces_sold	bill_raw_text
1	0	0	7/7/2022	12	bdgwqyboczbitzsxwmtjhehnxyvdfsddlftaiPmDbz
2	0	0	7/1/2022	71	wmewueqbyoqivcpjfsmadtgxsvxjaretdoitcymxqn
3	0	0	7/7/2022	7	qhweohgmqlxxlxzitjgbtnpqswakfihqywypyzuijd
4	0	0	7/7/2022	85	bxwiyotlyldofwovdmyrzuujhpgvjhwshlxpmhvjjyli
5	0	0	7/9/2022	53	qoSlyuqmnndmjleivcxijoqfcnftaxuqkabwdbgcgr
6	0	0	7/7/2022	17	yvjkecsvwxmImivopwbbiplmgkdmklzvmeibbrkvh
7	0	0	7/3/2022	16	iqkfxtfmznbtcrqbqqmiepvwfmqrkamlfceijbepfn
8	0	0	7/5/2022	18	euwaghlijdvkrwigxdwqvjegefiagyayggjemjpnxdsw

## 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

Here's a sample of the table:

product_id	product_name	price
0	product_0	22.00
1	product_1	85.00
2	product_2	109.00
3	product_3	100.00
4	product_4	49.00
5	product_5	102.00
6	product_6	101.00
7	product_7	147.00
8	product_8	85.00

---

## Sellers Table

This table contains the list of all the sellers:

- **seller\_id:** The seller ID
- **seller\_name:** The seller's 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

Here's a sample of the table:

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! We would advise doing so before reading the solutions that are available at the end of the page!

Tip: We 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).

### 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("Exercisel") \  
    .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("Exercisel") \  
    .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("Exercisel") \  
    .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:  $10/250 = 0.04$   
Order 2:  $8/250 = 0.032$   
Order 3:  $7/250 = 0.028$   
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("Exercisel") \  
    .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("Exercisel") \  
    .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 'nbAAAnllA', 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 textFinally, 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("Exercisel") \  
    .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:

We get the following output:

```
Number of Orders: 2,000,040
Number of sellers: 10
Number of products: 750,000
```

As you can see, we have 750,000 products in our dataset and 2,000,040 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:

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)|
+-----+
|                648865|
+-----+
```

```
Product present in more orders
```

```
+-----+-----+
|product_id|    cnt|
+-----+-----+
|         0|500,000|
+-----+-----+
```

Let's have a closer look at the second result: 500,000 orders out of 750,000 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":

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



```

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") \
    .config("spark.ui.port", "4040") \
    .appName("Exercisel") \
    .getOrCreate()

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

# 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()

spark.stop()

```

how many distinct products have been sold in each date

```

+-----+-----+
|    date|distinct_products_sold|
+-----+-----+
|2022-07-04|          136390|
|2022-07-03|          136308|
|2022-07-01|          136182|
|2022-07-08|          136133|
|2022-07-06|          136068|
|2022-07-02|          136003|
|2022-07-05|          135974|
|2022-07-10|          135877|
|2022-07-07|          135504|
|2022-07-09|          135204|
+-----+-----+

```

---

## 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  $\text{revenue} = \text{price} * \text{quantity}$ . Petty 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

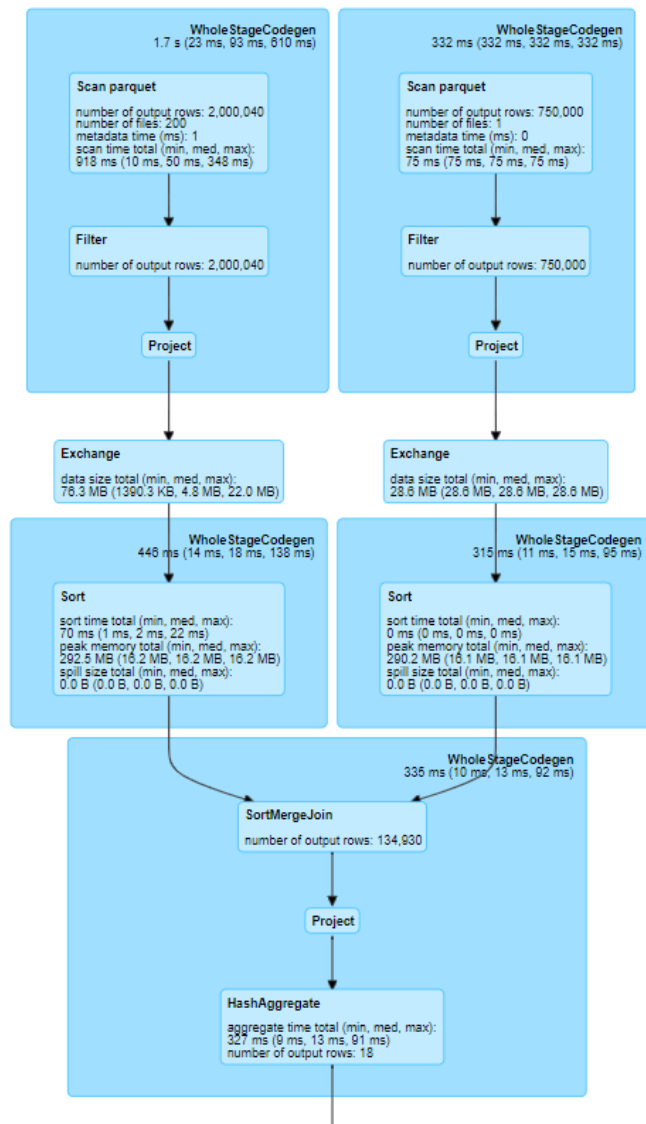
# Create the Spark session
spark = SparkSession.builder \
    .master("local") \
    .config("spark.sql.autoBroadcastJoinThreshold", -1) \
    .config("spark.executor.memory", "500mb") \
    .config("spark.ui.port", "4040") \
    .appName("Exercisel") \
    .getOrCreate()

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

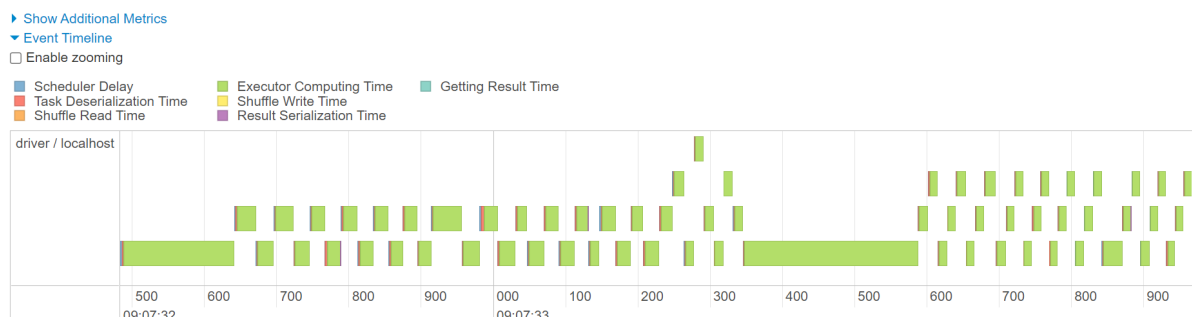
# Do the join and print the results
print(sales_table.join(products_table, sales_table["product_id"] ==
    products_table["product_id"], "inner").
    agg(avg(products_table["price"] * sales_table["num_pieces_sold"])).show())
```

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:

Submitted Time: 2022/06/14 11:46:51  
Duration: 4 s  
Running Jobs: 3



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



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`).

**Skewed join:** A skew join is used when there is a table with skew data in the joining column. A skew table is a table that is having values that are present in large numbers in the table compared to other data. Skew data is stored in a separate file while the rest of the data is stored in a separate file.

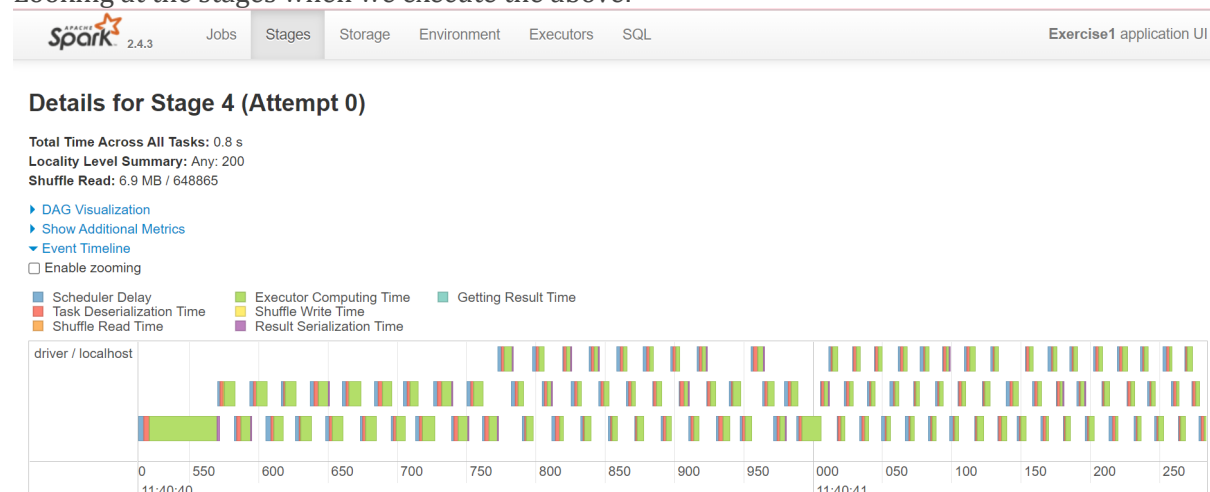
**Note that this is not a huge problem in case you are running Spark on a local system.** On a distributed environment (and with more data), though, this join could take **an incredible amount of time to complete** (maybe never complete at all!).

Let's fix this issue using a technique known as "key salting". We won't describe in detail but 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_01`, `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":**

Looking at the stages when we execute the above:



```

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("Exercisel") \
    .getOrCreate()

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

```

The result of the query should be the following

```

+-----+
|avg((price * num_pieces_sold))|
+-----+
|          2998.7587468250636|
+-----+

```

---

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("Exercisel") \
    .getOrCreate()

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

# Wrong way to do this - Skewed
# (Note that Spark will probably broadcast the table anyway, unless we forbid it
# through the configuration parameters)
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 broadcasting
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())

spark.stop()

```

### 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:

1. We get the sum of sales for each product and seller pairs.
2. We add two new ranking columns: one that ranks the products' sales in descending order and another one that ranks in ascending order.
3. We split the dataset obtained in three pieces: one for each case that we want to handle (second top selling, least selling, single selling).
4. 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
5. We merge the pieces back together.

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 dataframes 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:

**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



```

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("Exercise4") \
    .getOrCreate()

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

spark.stop()

```

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

---

```
+-----+---+
|hashed_bill|cnt|
+-----+---+
+-----+---+
```

```
Process finished with exit code 0
```

## Take Away

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

SQL development:

1. **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.
2. **Window functions:** Super useful, the only thing to remember is to first define the windowing.
3. **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, I'm open to suggestions! I hope you enjoyed!