

Centro de Investigación y de Estudios Avanzados

CIENCIAS DE DATOS

INTERACCIÓN SPARK Y PYTHON

Ejercicio 3

JOSÉ CARLOS MORÍN GARCÍA

Fecha de realización: 14 de febrero de 2021

Fecha de entrega: 14 de febrero de 2021

Firma del docente:



Problema o Resumen

En este reporte se expone la realización de calentamiento y ejercicios de un dataset para realizar preprocesamiento y análisis de estos datos. Todo con el fin de generar experiencia en el framework Spark y Python. Así mismo se realizaron lecturas acerca de estos dos componentes que se utilizaron.

Índice

1. Introducción Teórica	1
1.1. Framework Spark (1)	1
1.2. Python	2
1.3. Docker	3
2. Método	3
2.1. Características del servidor	3
2.2. Preparación del contenedor	3
2.3. Dataset	4
2.4. Tabla Sales	4
2.4.1. Tabla Products	5
2.4.2. Tabla Sellers	5
3. Practicas realizadas	5
3.1. Calentamiento 1	5
3.2. Calentamiento 2	6
3.3. Ejercicio 1	6
3.4. Ejercicio 2	6
3.5. Ejercicio 3	7
3.6. Ejercicio 4	7
4. Solución y Resultados	7
4.1. Calentamiento 1	8
4.2. Calentamiento 2	9
4.3. Ejercicio 1	9
4.4. Ejercicio 2	11
4.5. Ejercicio 3	12
4.6. Ejercicio 4	14

1. Introducción Teórica

Para una entrada al resto del documento se explicaran brevemente los temas del framework Spark y el lenguaje de Python así como Docker que se utilizo para la ejecución de los archivos de Python.

1.1. Framework Spark (1)

La API Spark Python (PySpark) expone el modelo de programación Spark a Python. Para aprender los conceptos básicos de Spark, recomendamos leer primero la guía de

programación de Scala; debería ser fácil de seguir incluso si no conoce Scala. Esta guía mostrará cómo usar las funciones de Spark que se describen allí en Python.

Hay algunas diferencias clave entre las API de Python y Scala:

- Python se escribe dinámicamente, por lo que los RDD pueden contener objetos de varios tipos.
- PySpark aún no admite algunas llamadas a la API, como archivos de entrada de búsqueda y sin texto, aunque se agregarán en versiones futuras.

En PySpark, los RDD admiten los mismos métodos que sus contrapartes de Scala, pero toman funciones de Python y devuelven tipos de colección de Python. Las funciones cortas se pueden pasar a los métodos RDD utilizando la sintaxis lambda de Python:

```
1 logData = sc.textFile(logFile).cache()
2 errors = logData.filter(lambda line: "ERROR" in line)
```

También puede pasar funciones que se definen con la palabra clave `def`; esto es útil para funciones más largas que no se pueden expresar usando lambda:

```
1 def is_error(line):
2     return "ERROR" in line
3 errors = logData.filter(is_error)
```

Las funciones pueden acceder a objetos en ámbitos adjuntos, aunque las modificaciones a esos objetos dentro de los métodos RDD no se propagarán:

```
1 error_keywords = ["Exception", "Error"]
2 def is_error(line):
3     return any(keyword in line for keyword in error_keywords)
4 errors = logData.filter(is_error)
```

PySpark enviará automáticamente estas funciones a los trabajadores, junto con cualquier objeto al que hagan referencia. PySpark serializará y enviará instancias de clases a los trabajadores, pero las clases en sí no se pueden distribuir automáticamente a los trabajadores. La sección Uso independiente describe cómo enviar dependencias de código a los trabajadores.

1.2. Python

Es un lenguaje de programación interpretado, multiparadigma y multiplataforma usado, principalmente, en Big Data, AI (Inteligencia Artificial), Data Science, frameworks de pruebas y desarrollo web. Esto lo convierte en un lenguaje de propósito general de gran nivel debido a su extensa biblioteca, cuya colección ofrece una amplia gama de instalaciones.(2)

Un lenguaje sencillo, legible y elegante que atiende a un conjunto de reglas que hacen muy corta su curva de aprendizaje. Si ya tienes unas nociones de programación o vienes de programar en otros lenguajes como Java no te será difícil comenzar a leer y entender el código desarrollado en Python.

El siguiente paso es comenzar a programar, verás que con muy pocas líneas de código es posible programar algoritmos complejos. Esto hace de Python un lenguaje práctico que permite ahorrar mucho tiempo.(3)

1.3. Docker

Docker es una plataforma de software que le permite crear, probar e implementar aplicaciones rápidamente. Docker empaqueta software en unidades estandarizadas llamadas contenedores que incluyen todo lo necesario para que el software se ejecute, incluidas bibliotecas, herramientas de sistema, código y tiempo de ejecución. Con Docker, puede implementar y ajustar la escala de aplicaciones rápidamente en cualquier entorno con la certeza de saber que su código se ejecutará.

Puede utilizar los contenedores de Docker como bloque de construcción principal a la hora de crear aplicaciones y plataformas modernas. Docker facilita la creación y la ejecución de arquitecturas de microservicios distribuidos, la implementación de código con canalizaciones de integración y entrega continuas estandarizadas, la creación de sistemas de procesamiento de datos altamente escalables y la creación de plataformas completamente administradas para sus desarrolladores.(4)

2. Método

Para comenzar con estas practicas de spark con python es un servidor se coloco mediante la plataforma de docker, un contenedor que contenia este framework y python instalado, el nombre del contenedor utilizado fue submit-st03_0, el cual se le genero un volumen en donde se ubicaban los archivos correspondientes al dataset y los ejecutables de python de los diferentes ejercicios a aplicar.

2.1. Características del servidor

El servidor utilizado fue prestado por la misma institución, el cual se accedió mediante una VPN y ssh al mismo con las claves correspondientes. Las características de hardware del computador son:

- Procesador: Intel(R) Xeon(R) CPU E5645 @ 2.40GHz.
- Memoria RAM: 12 GB
- Cores: 6
- SO: CentOS Linux release 7.4.1708 (Core)

2.2. Preparación del contenedor

Primeramente se creo un contenedor con el volumen de la ruta especifica de los archivos del dataset y los py, como podemos observar en el siguiente comando:

```
1 docker run -ti -v /data/st03/:/data/st03/py --name submit-st03_0 --user st03 --network spark-net -p 4049:4040 submituser:st03
```

Como se puede observar la ruta se encuantra en una carpeta especifica en la dirección */data/* y es ahí donde se colocan los archivos, como se puede observar en la Figura 1.

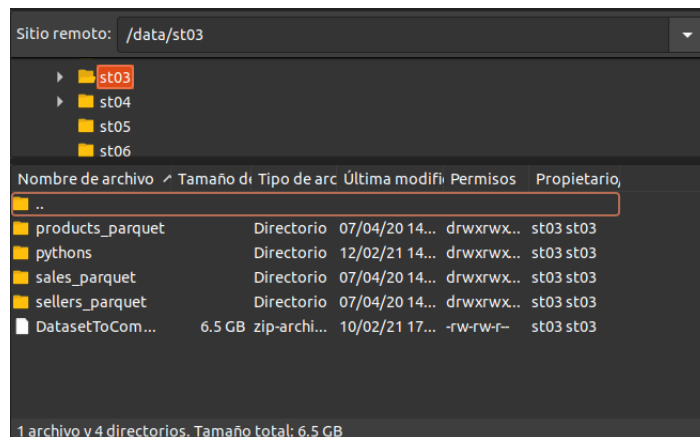


Figura 1: Ubicación de los archivos.

Para ingresar al contenedor y visualizar los datos se realizaron en el volumen los siguientes comandos:

```
1 docker exec -ti submit-st03_0 bash
2 cd /data/st03/
```

2.3. Dataset

El conjunto de datos que vamos a utilizar: consta de tres tablas provenientes de la base de datos de una tienda, con productos, ventas y vendedores. En la Figura 2 podemos observar la relaciones entre las tablas del dataset.

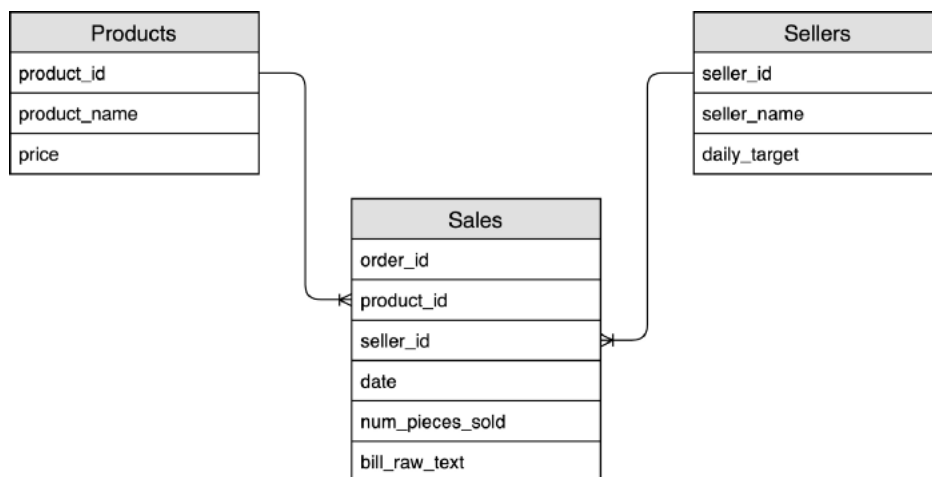


Figura 2: Tablas y sus conexiones

2.4. Tabla Sales

Cada fila de esta tabla es un pedido y cada pedido puede contener solo un producto. Cada fila almacena los siguientes campos:

- **order_id:** Registra el numero de orden.
- **product_id:** El único producto vendido en el pedido, por lo menos los pedidos cuentan con un pedido.
- **seller_id:** Identificación del empleado que hizo la venta.
- **date:** fecha en que se realizó la venta.
- **num_pieces_sold:** Número de piezas vendidas.
- **bill_raw_text:** Una cadena que representa el texto sin formato de la factura asociada con el pedido.

2.4.1. Tabla Products

Representa los distintos productos de la tienda, en cada registro se toman en cuenta los siguientes atributos:

- **product_id:** Identificador del producto.
- **product_name:** Nombre del producto.
- **price:** Precio del producto.

2.4.2. Tabla Sellers

Contiene la lista de todos los vendedores de la tienda. Y se registran mediante los siguientes atributos.

- **seller_id:** Identificador de los vendedores.
- **seller_name:** Nombre del vendedor.
- **daily_target:** La cantidad de artículos (independientemente del tipo de producto) que el vendedor necesita para alcanzar su cuota.

3. Practicas realizadas

En este apartado se explicara cada uno de los ejercicio y su objetivo a realizar.

3.1. Calentamiento 1

En primera instancia para conocer los datos se realizo un proceso pequeño en donde se supo cuántos pedidos, cuántos productos y cuántos vendedores hay en los datos.

Para este primer calentamiento de los datos se resolvieron las siguientes preguntas:

- ¿Cuántos productos se han vendido al menos una vez?

- ¿Cuál es el producto contenido en más pedidos?

Dentro de la configuración de sesión en Spark se realizó bajo los siguientes términos:

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

3.2. Calentamiento 2

Para este calentamiento y poder conocer un poco más los datos, se respondió la pregunta ¿Cuántos productos distintos se han vendido cada día?.

Para la sesión de Spark se configuro de la siguiente manera:

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

3.3. Ejercicio 1

Dentro de este ejercicio, ya más completos en comparación con los calentamientos, se realizó un código el cual nos ayuda a responder la siguiente pregunta:

- ¿Cuál es el ingreso promedio de los pedidos?

De acuerdo con la configuración de la sesión con Spark fue la siguiente:

```
1 spark = SparkSession.builder \
2     .master("local") \
3     .config("spark.sql.autoBroadcastJoinThreshold", -1) \
4     .config("spark.executor.memory", "3g") \
5     .appName("Exercise1_st03") \
6     .getOrCreate()
```

3.4. Ejercicio 2

Dentro de este ejercicio se buscó obtener para cada vendedor, ¿cuál es el porcentaje medio de contribución de un pedido a la cuota diaria del vendedor?

La sesión de Spark se se configuró como se muestra a continuación:

```
1 spark = SparkSession.builder \
2     .master("local") \
3     .config("spark.sql.autoBroadcastJoinThreshold", -1) \
4     .config("spark.executor.memory", "3g") \
5     .appName("Exercise2_st03") \
6     .getOrCreate()
```

3.5. Ejercicio 3

Lo resuelto en el ejercicio 3 se realizó lo siguiente: ¿Quiénes son las segundas personas más vendidas y las que menos venden (vendedores) para cada producto? ¿Quiénes son los del producto con 'product_id = 0'.

La sesión de Spark fue la siguiente:

```
1 spark = SparkSession.builder \
2     .master("local") \
3     .config("spark.sql.autoBroadcastJoinThreshold", -1) \
4     .config("spark.executor.memory", "3g") \
5     .appName("Exercise3_st03") \
6     .getOrCreate()
```

3.6. Ejercicio 4

En este ultimo ejercicio y mas largo de todos se realizó una nueva columna llamada "hashed.bill" definida de la siguiente manera:

- Si el order_id es par: aplique el hash MD5 iterativamente al campo bill_raw.text, una vez por cada 'A' (mayúscula 'A') presente en el texto.
Por ejemplo, si el texto de la factura es 'nbAAnIIA', aplicaría hash tres veces de forma iterativa (solo si el número de pedido es par).
- Si el order_id es impar: aplique el hash SHA256 al texto de la factura.

Y finalmente, se verificó si hay algún duplicado en el nuevo columna.

De acuerdo a la sesión de Spark fue la siguiente:

```
1 spark = SparkSession.builder \
2     .master("local") \
3     .config("spark.sql.autoBroadcastJoinThreshold", -1) \
4     .config("spark.executor.memory", "3g") \
5     .appName("Exercise4_st03") \
6     .getOrCreate()
```

4. Solución y Resultados

En este apartado se mostraran los códigos y resultados arrojados para cada calentamiento y ejercicio.

Para correr el archivo de python se ejecutó el siguiente comando en la terminal del contenedor. Simplemente se cambio el nombre del archivo.

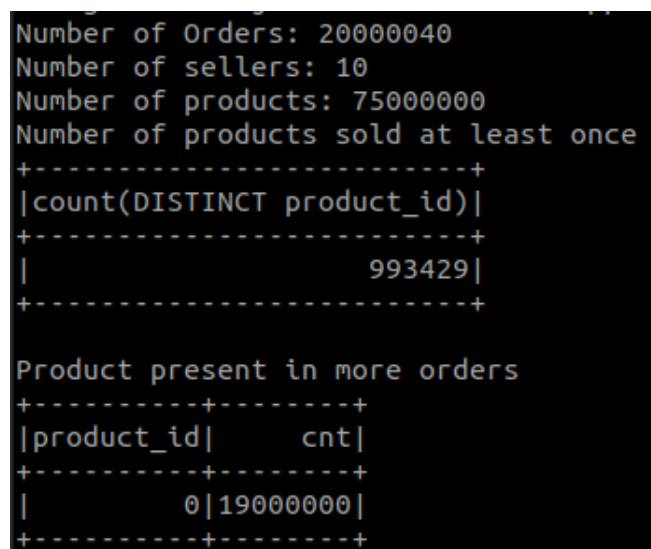
```
1 $SPARK_HOME/bin/spark-submit --conf spark.executor.cores=3 --conf spark.executor.memory=3G --master spark://spark-master:7077 /data/st03/py/python/name.py
```


4.1. Calentamiento 1

Para el calentamiento 1 solo se necesita contar las filas en cada tabla del dataset.

```
1 # Load source data
2 products_table = spark.read.parquet("./products_parquet")
3 sales_table = spark.read.parquet("./sales_parquet")
4 sellers_table = spark.read.parquet("./sellers_parquet")
5
6 # Define the UDF function
7 def algo(order_id, bill_text):
8     # If number is even
9     ret = bill_text.encode("utf-8")
10    if int(order_id) % 2 == 0:
11        # Count number of 'A'
12        cnt_A = bill_text.count("A")
13        for _c in range(0, cnt_A):
14            ret = hashlib.md5(ret).hexdigest().encode("utf-8")
15        ret = ret.decode('utf-8')
16    else:
17        ret = hashlib.sha256(ret).hexdigest()
18    return ret
19
20 # Register the UDF function.
21 algo_udf = spark.udf.register("algo", algo)
22
23 # Use the 'algo_udf' to apply the algorithm and then check if there is any
24 # duplicate hash in the table
25 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()
```

Y como resultados arrojados se construyó una tabla en donde se muestran los conteos de cada tabla. (La podemos observar en la Figura 3).



```
Number of Orders: 20000040
Number of sellers: 10
Number of products: 75000000
Number of products sold at least once
+-----+
|count(DISTINCT product_id)|
+-----+
|                          993429|
+-----+

Product present in more orders
+-----+-----+
|product_id|    cnt|
+-----+-----+
|          0|19000000|
+-----+-----+
```

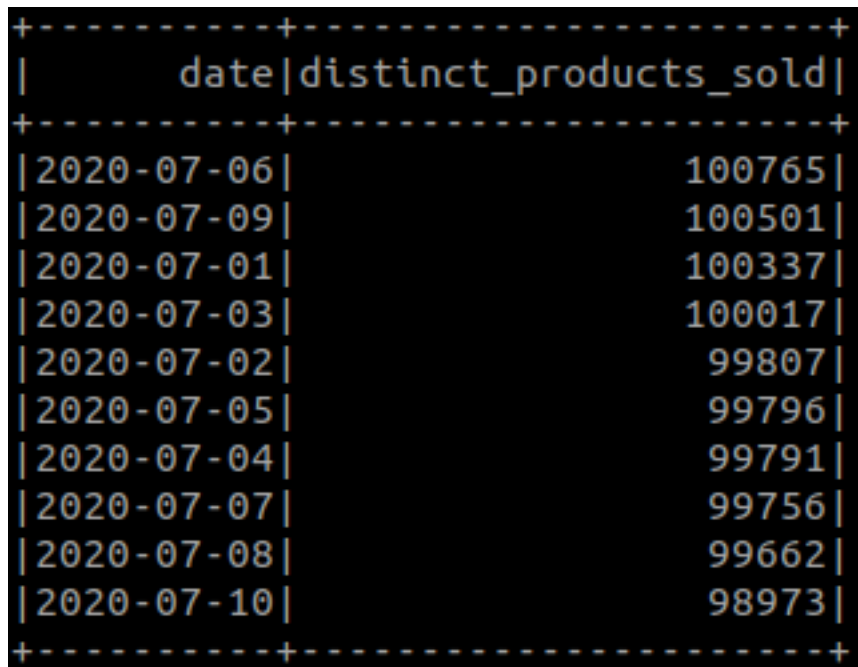
Figura 3: Resultados del Calentamiento 1.

4.2. Calentamiento 2

Dentro del calentamiento 2, simplemente se contó cuantos productos distintos se vendieron dentro de una fecha en específico, todo esto se realizó con el siguiente fragmento de código.

```
1 # Read Source tables
2 products_table = spark.read.parquet("./products_parquet")
3 sales_table = spark.read.parquet("./sales_parquet")
4 sellers_table = spark.read.parquet("./sellers_parquet")
5
6 sales_table.groupby(col("date")).agg(countDistinct(col("product_id")).alias(
7     "distinct_products_sold")).orderBy(
8     col("distinct_products_sold").desc()).show()
```

El resultado del proceso lo podemos ver en la Figura 4.



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

Figura 4: Resultados del Calentamiento 2

4.3. Ejercicio 1

Para el ejercicio 1 se resolvió el promedio de ingresos de los pedidos; primero se calcularon los ingresos de cada pedido y luego obtener el promedio. Todo esto se ejecutó con el siguiente código:

```
1 # Read the source tables
2 products_table = spark.read.parquet("./products_parquet")
3 sales_table = spark.read.parquet("./sales_parquet")
4 sellers_table = spark.read.parquet("./sellers_parquet")
5
6 # Step 1 - Check and select the skewed keys
7 # In this case we are retrieving the top 100 keys: these will be the only
8   salted keys.
```

```

8 results = sales_table.groupby(sales_table["product_id"]).count().sort(col("
    count").desc()).limit(100).collect()
9
10 # Step 2 - What we want to do is:
11 # a. Duplicate the entries that we have in the dimension table for the most
    common products, e.g.
12 #     product_0 will become: product_0-1, product_0-2, product_0-3 and so
    on
13 # b. On the sales table, we are going to replace "product_0" with a random
    duplicate (e.g. some of them
14 #     will be replaced with product_0-1, others with product_0-2, etc.)
15 # Using the new "salted" key will unskew the join
16
17 # Let's create a dataset to do the trick
18 REPLICATION_FACTOR = 101
19 l = []
20 replicated_products = []
21 for _r in results:
22     replicated_products.append(_r["product_id"])
23     for _rep in range(0, REPLICATION_FACTOR):
24         l.append((_r["product_id"], _rep))
25 rdd = spark.sparkContext.parallelize(l)
26 replicated_df = rdd.map(lambda x: Row(product_id=x[0], replication=int(x[1])
    ))
27 replicated_df = spark.createDataFrame(replicated_df)
28
29 # Step 3: Generate the salted key
30 products_table = products_table.join(broadcast(replicated_df),
31                                     products_table["product_id"] ==
32                                     replicated_df["product_id"], "left"). \
33     withColumn("salted_join_key", when(replicated_df["replication"].isNull()
34     , products_table["product_id"]).otherwise(
35     concat(replicated_df["product_id"], lit("-"), replicated_df["replication
36     "]]))
37
38 sales_table = sales_table.withColumn("salted_join_key", when(sales_table["
39     product_id"].isin(replicated_products),
40     concat(
41     sales_table["product_id"], lit("-"),
42     round(
43     rand() * (REPLICATION_FACTOR - 1), 0).cast(
44     IntegerType()))).otherwise(
45     sales_table["product_id"]))
46
47 # Step 4: Finally let's do the join
48 print(sales_table.join(products_table, sales_table["salted_join_key"] ==
49     products_table["salted_join_key"],
50     "inner").
51     agg(avg(products_table["price"] * sales_table["num_pieces_sold"])).
52     show())
53
54 print("Ok")

```

Los resultados obtenidos se muestran en la Figura 5.

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

None
Ok
```

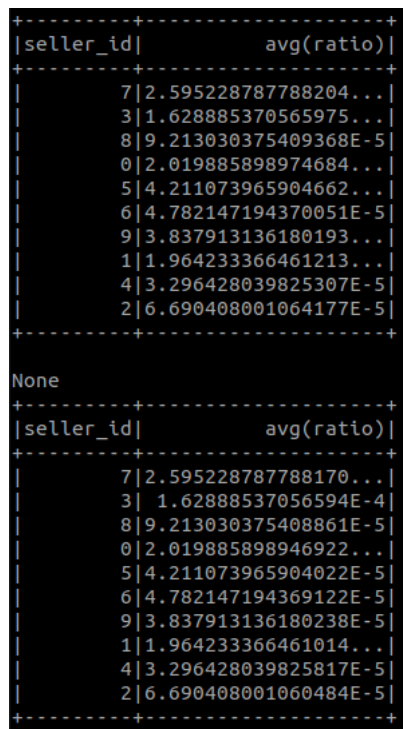
Figura 5: Resultados del Ejercicio 1

4.4. Ejercicio 2

Este ejercicio es similar al primer ejercicio: se hizo un proceso de unión con la tabla de vendedores, y calculamos el porcentaje de acierto de cuota gracias a un pedido específico y hacemos la media, agrupando por el seller_id. El código se muestra a continuación:

```
1  # Read the source tables
2  products_table = spark.read.parquet("./products_parquet")
3  sales_table = spark.read.parquet("./sales_parquet")
4  sellers_table = spark.read.parquet("./sellers_parquet")
5
6  # Wrong way to do this - Skewed
7  # (Note that Spark will probably broadcast the table anyway, unless we
8  # forbid it through the configuration parameters)
9  print(sales_table.join(sellers_table, sales_table["seller_id"] ==
10                        sellers_table["seller_id"], "inner").withColumn(
11                        "ratio", sales_table["num_pieces_sold"]/sellers_table["daily_target"]
12                        ).groupBy(sales_table["seller_id"]).agg(avg("ratio")).show())
13
14 # Correct way through broadcasting
15 print(sales_table.join(broadcast(sellers_table), sales_table["seller_id"] ==
16                        sellers_table["seller_id"], "inner").withColumn(
17                        "ratio", sales_table["num_pieces_sold"]/sellers_table["daily_target"]
18                        ).groupBy(sales_table["seller_id"]).agg(avg("ratio")).show())
```

El resultado obtenido es el siguiente Figura 6:



```
+-----+-----+
|seller_id|      avg(ratio)|
+-----+-----+
7|2.595228787788204...|
3|1.628885370565975...|
8|9.213030375409368E-5|
0|2.019885898974684...|
5|4.211073965904662...|
6|4.782147194370051E-5|
9|3.837913136180193...|
1|1.964233366461213...|
4|3.296428039825307E-5|
2|6.690408001064177E-5|
+-----+-----+

None
+-----+-----+
|seller_id|      avg(ratio)|
+-----+-----+
7|2.595228787788170...|
3| 1.62888537056594E-4|
8|9.213030375408861E-5|
0|2.019885898946922...|
5|4.211073965904022E-5|
6|4.782147194369122E-5|
9|3.837913136180238E-5|
1|1.964233366461014...|
4|3.296428039825817E-5|
2|6.690408001060484E-5|
+-----+-----+
```

Figura 6: Resultados del Ejercicio 2

4.5. Ejercicio 3

Aquí en este ejercicio se tuvo que condicionar el programa debido a que se necesitó encontrar el segundo vendedor con más ventas y el último vendedor. Para esto se establecieron las siguientes condiciones:

- Un solo vendedor ha vendido un producto.
- Si un producto ha sido vendido por más de un vendedor, pero todos vendieron la misma cantidad.
- Si el "menos vendido" es también el "segundo vendedor".

Para generar esto, se creó el siguiente código:

```
1 # Read the source tables
2 products_table = spark.read.parquet("./products_parquet")
3 sales_table = spark.read.parquet("./sales_parquet")
4 sellers_table = spark.read.parquet("./sellers_parquet")
5
6 # Calculate the number of pieces sold by each seller for each product
7 sales_table = sales_table.groupby(col("product_id"), col("seller_id")). \
8     agg(sum("num_pieces_sold").alias("num_pieces_sold"))
9
10 # Create the window functions, one will sort ascending the other one
11 # and sort by the pieces sold
```

```

12 window_desc = Window.partitionBy(col("product_id")).orderBy(col("
    num_pieces_sold").desc())
13 window_asc = Window.partitionBy(col("product_id")).orderBy(col("
    num_pieces_sold").asc())
14
15 # Create a Dense Rank (to avoid holes)
16 sales_table = sales_table.withColumn("rank_asc", dense_rank().over(
    window_asc)). \
17     withColumn("rank_desc", dense_rank().over(window_desc))
18
19 # Get products that only have one row OR the products in which multiple
    sellers sold the same amount
20 # (i.e. all the employees that ever sold the product, sold the same exact
    amount)
21 single_seller = sales_table.where(col("rank_asc") == col("rank_desc")).
    select(
22     col("product_id").alias("single_seller_product_id"), col("seller_id").
    alias("single_seller_seller_id"),
23     lit("Only seller or multiple sellers with the same results").alias("type
    ")
24 )
25
26 # Get the second top sellers
27 second_seller = sales_table.where(col("rank_desc") == 2).select(
28     col("product_id").alias("second_seller_product_id"), col("seller_id").
    alias("second_seller_seller_id"),
29     lit("Second top seller").alias("type")
30 )
31
32 # Get the least sellers and exclude those rows that are already included in
    the first piece
33 # We also exclude the "second top sellers" that are also "least sellers"
34 least_seller = sales_table.where(col("rank_asc") == 1).select(
35     col("product_id"), col("seller_id"),
36     lit("Least Seller").alias("type")
37 ).join(single_seller, (sales_table["seller_id"] == single_seller["
    single_seller_seller_id"]) & (
38     sales_table["product_id"] == single_seller["single_seller_product_id
    "]), "left_anti"). \
39     join(second_seller, (sales_table["seller_id"] == second_seller["
    second_seller_seller_id"]) & (
40     sales_table["product_id"] == second_seller["second_seller_product_id
    "]), "left_anti")
41
42 # Union all the pieces
43 union_table = least_seller.select(
44     col("product_id"),
45     col("seller_id"),
46     col("type")
47 ).union(second_seller.select(
48     col("second_seller_product_id").alias("product_id"),
49     col("second_seller_seller_id").alias("seller_id"),
50     col("type")
51 ).union(single_seller.select(
52     col("single_seller_product_id").alias("product_id"),
53     col("single_seller_seller_id").alias("seller_id"),
54     col("type")
55 ))
56 union_table.show()

```

```

57
58 # Which are the second top seller and least seller of product 0?
59 union_table.where(col("product_id") == 0).show()

```

A mi experiencia fue el ejercicio que mas se complico por la estrategia utilizada de obtener el segundo y último vendedor. Los resultados obtenidos los podemos observar en la Figura

```

+-----+-----+-----+
|product_id|seller_id|      type|
+-----+-----+-----+
|  19986717|         1|Least Seller|
|  40496308|         5|Least Seller|
|  52606213|         7|Least Seller|
|  14542470|         5|Least Seller|
|  28592106|         5|Least Seller|
|  17944574|         8|Least Seller|
|  61475460|         7|Least Seller|
|   3534470|         3|Least Seller|
|  35669461|         4|Least Seller|
|  32602520|         9|Least Seller|
|  72017876|         1|Least Seller|
|  67723231|         5|Least Seller|
|  56011040|         5|Least Seller|
|  34681047|         5|Least Seller|
|  57735075|         9|Least Seller|
|  18182299|         7|Least Seller|
|  69790381|         5|Least Seller|
|  31136332|         9|Least Seller|
|  10978356|         7|Least Seller|
|  20774718|         9|Least Seller|
+-----+-----+-----+
only showing top 20 rows
+-----+-----+-----+
|product_id|seller_id|      type|
+-----+-----+-----+
|         0|         0|Only seller or mu...|
+-----+-----+-----+

```

Figura 7: Resultados del Ejercicio 3

4.6. Ejercicio 4

El cuarto y ejercicio final de esta practica se creó una funcion creada por el susuario, en donde se pasan como paramtros los atributos que se desea calcular, en este caso se pasaron los atributos order_id y bill_text, que ste ultimo es el que se iba a cifrar dependiendo del valor de order_id. El código implementado fue el siguiente:

```

1  # Load source data
2  products_table = spark.read.parquet("./products_parquet")
3  sales_table = spark.read.parquet("./sales_parquet")
4  sellers_table = spark.read.parquet("./sellers_parquet")
5
6  # Define the UDF function
7  def algo(order_id, bill_text):
8      # If number is even
9      ret = bill_text.encode("utf-8")
10     if int(order_id) % 2 == 0:
11         # Count number of 'A'
12         cnt_A = bill_text.count("A")
13         for _c in range(0, cnt_A):
14             ret = hashlib.md5(ret).hexdigest().encode("utf-8")

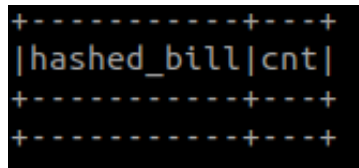
```

```

15         ret = ret.decode('utf-8')
16     else:
17         ret = hashlib.sha256(ret).hexdigest()
18     return ret
19
20 # Register the UDF function.
21 algo_udf = spark.udf.register("algo", algo)
22
23 # Use the 'algo_udf' to apply the algorithm and then check if there is any
24 # duplicate hash in the table
25 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()

```

En la Figura 8 podemos observar los resultados obtenidos de acuerdo a lo implementado en el código. En el conjunto de datos final, todos los hash deberían ser diferentes, por lo que la consulta debería devolver un conjunto de datos vacío



```

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

```

Figura 8: Resultados del Ejercicio 4

Referencias

- [1] "Python programming guide."
- [2] Crehana, "Python: El lenguaje de programación más popular para aprender en 2021," Jan 2021.
- [3] Robledano, "Qué es python: Características, evolución y futuro," Jul 2020.
- [4] A. Ciro, "El mundo es ancho y ajeno," 2010.