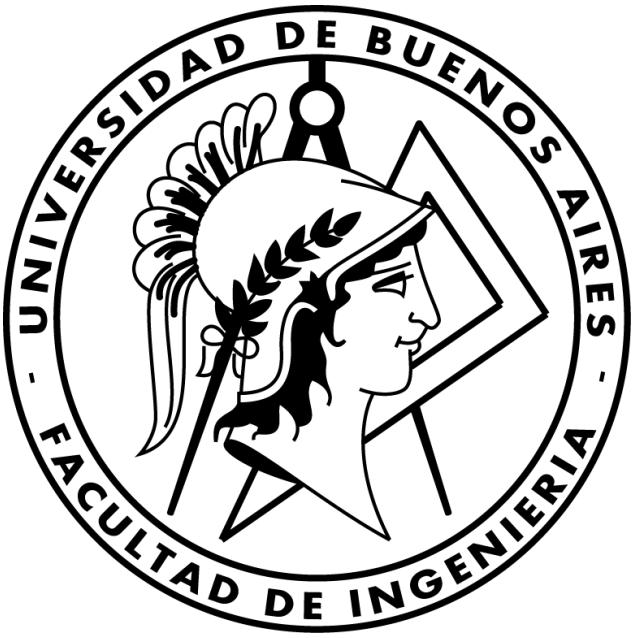

(TA047) - Ciencia de Datos

Trabajo Práctico 2 (2C2025)

Patricio Ibar - Padrón 109569

6 de octubre de 2025

Índice

1. Introducción

En el presente informe se documentan los pasos seguidos para la realización del Trabajo Práctico 2 de la materia Ciencia de Datos (TA047) de la Facultad de Ingeniería de la Universidad de Buenos Aires. El objetivo del trabajo práctico es utilizar Apache Spark para ejemplificar cómo realizar un procesamiento distribuido de datos, así como los desafíos que implica pensar en un análisis distribuido. La consigna del trabajo práctico y los datos a analizar pueden encontrarse en el siguiente enlace: `https://organizacion-de-datos-7506-argerich.github.io/consigna_tp2_2c2025.html`.

Se trabajó utilizando Python y la librería pyspark.

Tanto el código fuente, junto con las visualizaciones, como el informe se encuentran disponibles en el siguiente repositorio de GitHub: `https://github.com/patricioibar/datos-tp2`

2. Tratamiento Inicial de Datos

Antes de realizar las consultas, es fundamental llevar a cabo un tratamiento inicial de los datos para asegurar su calidad y consistencia. Este proceso incluye la carga de datos, el manejo de valores faltantes y la normalización de datos.

Los datos fueron cargados desde un archivo CSV utilizando SQLContext de Spark. Al cargar una tabla, lo primero que hice en todos los casos fue tomar el RDD subyacente y hacer un map para quedarme solo con las columnas que me interesan para las consultas que iba a realizar. Este map también se aprovecha para convertir los tipos de datos a tipos más adecuados, normalizar cadenas de texto, y extraer los datos necesarios para las consultas, construyendo nuevas columnas si es necesario.

En cuanto a la conversión a tipos de datos, se estableció el parámetro `inferSchema` en verdadero al leer el CSV, lo que le permite a Spark inferir automáticamente los tipos de datos de las columnas. Sin embargo, en muchos casos, fue necesario especificar explícitamente los tipos de datos al cargar los datos, especialmente para columnas de IDs, que se convirtieron en enteros. Esto no solo mejora la eficiencia del almacenamiento, sino que también facilita las operaciones de unión y filtrado.

Las columnas con valores categóricos como `status` y `payment_method` en la tabla de órdenes, se convirtieron también a números enteros identificadores. Estos números enteros se mapean a los valores reales a través de diccionarios en Python, lo que permite un acceso rápido y eficiente a los valores categóricos originales cuando es necesario. Se considera que estos diccionarios no ocupan un espacio significativo en memoria, por lo que fueron **broadcasteados** a todos los nodos del clúster para optimizar el rendimiento durante las consultas.

Esto se realizo siguiendo un patrón como el mostrado en el Listing 1. Se tiene un archivo de texto con todos los valores posibles para la columna, recolectados de lo aprendido en el TP1. Luego se crea un diccionario que mapea cada valor a un ID numérico, y se hace un broadcast de este diccionario para que esté disponible en todos los nodos del clúster. También se crea un diccionario inverso para mapear los IDs numéricos de vuelta a los valores originales cuando sea necesario, el cual en algunos casos se usa desde el nodo driver y en otros casos también se bradcastea a todos los nodos del cluster.

```
state_dict = {}
with open("states.txt", "r") as f:
    valid_states = [line.strip() for line in f.readlines()]
    id = 0
    for state in valid_states:
        state_dict[state] = id
        id += 1

states = sc.broadcast(state_dict)

state_id_to_name = {v: k for k, v in state_dict.items()}
```

Listing 1: Ejemplo de conversión de columna categórica a ID numérico

Este tratamiento se realizó para los campos de métodos de pago, razones de logs en el inventario, segmentos de cliente, estados de órdenes y estados de direcciones de la la orden. No se realizó para marcas de productos, aunque podría hacerse de ser necesario. Tampoco se realizó para categorías de productos.

Para la extracción de estado y código postal de la columna `billing_address` en la tabla de órdenes, se utilizó la misma estrategia que en el TP1: una expresión regular que busca un patrón específico en la cadena de texto. Este patrón consiste en dos letras mayúsculas seguidas de un espacio y cinco dígitos, que es el formato típico de los códigos postales en Estados Unidos. Si la dirección no contiene este patrón, se asigna un valor nulo a las nuevas columnas de estado y código postal.

```
def state_and_postal_code(address):
if address is None:
    return "UNDEFINED", None
pattern = r'([A-Z]{2})\s(\d{5})'
match = re.search(pattern, address)
if match:
    return match.group(1), int(match.group(2))
return "UNDEFINED", None
```

Listing 2: Extracción de estado y código postal de la dirección de facturación

Por último, para hacer más visual la lectura del código, se crearon diccionarios que mapean los nombres de las columnas a sus respectivos índices en el RDD. Esto facilita el acceso a las columnas por nombre en lugar de por índice numérico, mejorando la legibilidad del código. Preferí esta estrategia en lugar de usar objetos Row de Spark, ya que estos últimos tienen un overhead de memoria y procesamiento que hacía una diferencia notable en la rapidez en que se ejecutaban las consultas.

3. Consultas y Visualizaciones

Todas las consultas siguen un flujo de trabajo similar:

1. Limpieza y preprocesamiento de los datos. Esto es selección de columnas necesarias, manejo de valores nulos, creación de nuevas columnas si es necesario, etc.
2. Filtrado de los datos según las condiciones de la consulta.
3. Realización de otras transformaciones como joins, reduce by key y maps.
4. Caché de los resultados intermedios si es necesario.
5. Obtención del resultado final, a través de acciones como collect o show.

3.1. Consultas Propuestas por el Enunciado

A continuación se presentan las consultas propuestas por el enunciado, junto con las consideraciones tomadas para su resolución y los resultados obtenidos. El código fuente y los resultados completos pueden encontrarse en el notebook `consultas_enunciado.ipynb`.

3.1.1. ¿Cuál es el estado que más descuentos tiene en total? ¿y en promedio?

Para esta consulta se tomaron las siguientes suposiciones:

- Los estados con más descuentos son aquellos que tienen más órdenes registradas con un `discount_amount` no nulo y mayor a cero.
- Se tienen en cuenta los estados de la columna `billing_address` de la tabla `orders`.
- Se considera que el estado está definido por un conjunto de dos letras mayúsculas seguido de un espacio y cinco dígitos (código postal).
- No se consideran los estados que no están definidos por el anterior patrón, ni tampoco las filas con un valor nulo en `billing_address`.

```
discountAndTotalOrdersByState = ordersRDD \
.filter(lambda x: x[ordersIdx["state"]] != states.value["UNDEFINED"]) \
.map(
    lambda row: (
        row[ordersIdx["state"]],
        (
            1 if (row[ordersIdx["discount_amount"]] is not None
                and row[ordersIdx["discount_amount"]] > 0) else 0,
            1
        )
    )) \
.reduceByKey(lambda a, b: (a[0] + b[0], a[1] + b[1])) \
.cache()

highestCount = discountAndTotalOrdersByState\
    .reduce(lambda a, b: a if a[1][0] > b[1][0] else b)

highestAvg = discountAndTotalOrdersByState\
    .reduce(lambda a, b: a if a[1][0]/a[1][1] > b[1][0]/b[1][1] else b)
```

Listing 3: Resolución de la consulta 1 propuesta por el enunciado

Para resolver esta consulta se hace un filtro inicial para quedarnos solo con las filas que tienen un estado definido. Luego se hace un map para crear una tupla con el estado como clave, y como valor una tupla con la cantidad de órdenes con descuento y la cantidad total de órdenes para ese estado. Luego se hace un reduce by key para sumar las cantidades por estado, y se cachea el RDD intermedio ya que se utiliza para dos acciones distintas. Finalmente, se utilizan dos reducciones: una para encontrar el estado con la mayor cantidad de órdenes con descuento y otra para encontrar el estado con el mayor promedio de órdenes con descuento.

```
El Estado con mas ordenes con descuentos es AE con 30858 ordenes con descuentos.

El Estado con el mayor promedio de descuentos es KY con un promedio de 0.22.
```

Listing 4: Resultados de la consulta 1 propuesta por el enunciado

3.1.2. ¿Cuáles son los 5 códigos postales más comunes para las órdenes con estado 'Refunded'? ¿Y cuál es el nombre más frecuente entre los clientes de esas direcciones?

Para esta consulta se tomaron las siguientes suposiciones:

- Se tienen en cuenta los códigos postales de la columna `billing_address` de la tabla `orders`.
- Se considera que el código postal está definido por cinco dígitos precedidos de un espacio y un conjunto de dos letras mayúsculas (estado).
- Se buscan los códigos postales con más apariciones entre las órdenes con estado `Refunded`.
- Las órdenes con estado `Refunded` son todas aquellas que tienen como valor de la columna `status` el string `REFUNDED`, sin diferenciar mayúsculas de minúsculas.
- Para la búsqueda de nombres, se consideran a todos los usuarios de la tabla `customers` que tienen como valor de la columna `postal_code` alguno de los códigos postales encontrados.

```
zipCodesWithMostRefundedOrdersCount = ordersRDD \
.filter(
    lambda row: (
        row[ordersIdx["postal_code"]] is not None
        and row[ordersIdx["status"]] == statuses.value["REFUNDED"]
    )
) \
.map(lambda row: (row[ordersIdx["postal_code"]], 1)) \
.reduceByKey(lambda a, b: a + b) \
.takeOrdered(5, key=lambda x: -x[1])

topRefundedZipCodes = [postal_code for postal_code, _ in zipCodesWithMostRefundedOrdersCount]

mostFrecuentNameInTopRefundedZipCodes = customersRDD \
.filter(
    lambda row: (
        row[customersIdx["postal_code"]] in topRefundedZipCodes
        and row[customersIdx["name"]] != "UNDEFINED"
    )
) \
.map(lambda row: (row[customersIdx["name"]], 1)) \
.reduceByKey(lambda a, b: a + b) \
.reduce(lambda a, b: a if a[1] > b[1] else b)
```

Listing 5: Resolución de la consulta 2 propuesta por el enunciado

Para resolver esta consulta se hace un filtro inicial para quedarnos solo con las filas que tienen un código postal definido y un estado `Refunded`. Notar que la variabe `statuses` es un diccionario broadcasteado que

mapea los IDs de los status a sus nombres, como se mencionó en la sección 2. Luego se hace un map para crear una tupla con el código postal como clave y 1 como valor, y se hace un reduce by key para sumar las apariciones por código postal. Se utiliza la acción `takeOrdered` para obtener los 5 códigos postales con más apariciones. Luego, se hace otro filtro en la tabla de clientes para quedarnos solo con aquellos que tienen un código postal en la lista de los 5 más comunes y un nombre definido. Se hace un map para crear una tupla con el nombre como clave y 1 como valor, y se hace un reduce by key para sumar las apariciones por nombre. Finalmente, se utiliza una reducción para encontrar el nombre más frecuente entre los clientes de esos códigos postales.

Cabe destacar que, como se vió en el TP1, en esta consulta hay una gran cantidad de empates en los códigos postales más comunes, por lo que los resultados pueden variar entre ejecuciones. Sucede lo mismo con los nombres más frecuentes, ya que hay varios nombres que aparecen la misma cantidad de veces.

```
Los 5 codigos postales con mas ordenes reembolsadas son:
        Codigo  Ordenes reembolsadas
        31571   6
        14396   5
        9045    5
        38151   5
        91623   5

El nombre mas frecuente entre los clientes de esos codigos postales es: MICHAEL
```

Listing 6: Resultados de la consulta 2 propuesta por el enunciado

3.1.3. Para cada tipo de pago y segmento de cliente, devolver la suma y el promedio expresado como porcentaje, de clientes activos y de consentimiento de marketing.

Para esta consulta se tomó una única suposición: se consideran valores únicos por combinación (usuario, método de pago). De esta forma no contamos dos compras del mismo usuario con el mismo método de pago.

```python
clients_formated = customersRDD.map(
    lambda row: (
        row[customersIdx["id"]],
        (row[customersIdx["segment"]],
        row[customersIdx["is_active"]],
        row[customersIdx["consent"]])
    )
)
orders_formated = ordersRDD.map(
    lambda row: (
        row[ordersIdx["customer_id"]],
        row[ordersIdx["payment_method"]]
    )
)

clients_orders = clients_formated.join(orders_formated)\
    .map(
        lambda x: (
            (x[0], x[1][1]),
            (x[1][0][1], x[1][0][2], x[1][0][0])
        )
    )
# (customer_id, ((segment, is_active, consent), payment_method))
```

Listing 7: Resolución de la consulta 3 propuesta por el enunciado

```python
result = clients_orders_unique.map(
    lambda x: (
        (x[0][1], x[1][2]), # KEY = (payment_method, segment)
        (1 if x[1][0] else 0, 1 if x[1][1] else 0, 1)
    )  # VALUE = (is_active, consent, count)
).reduceByKey(
    lambda a, b: (
        a[0] + b[0],
        a[1] + b[1],
        a[2] + b[2]
    )
).map(
    lambda x: Row(
        payment_method=payments_id_to_name.value[x[0][0]],
        customer_segment=segment_id_to_name.value[x[0][1]],
        active_count=x[1][0],
        consent_count=x[1][1],
        active_percentage=float(f"{x[1][0]/x[1][2] * 100:.2f}"),
        consent_percentage=float(f"{x[1][1]/x[1][2] * 100:.2f}"),
    )
)
```

Listing 8: Resolución de la consulta 3 propuesta por el enunciado (continuación)

Para resolver esta consulta se hace un join entre las tablas de clientes y órdenes, utilizando como clave el ID del cliente. Luego se hace un map para crear una tupla con la combinación (cliente, método de pago) como clave, y como valor una tupla con los valores de `is_active`, `consent` y `segment` del cliente. De esta forma, si un cliente tiene varias órdenes con el mismo método de pago, solo se contará una vez. Luego se hace un reduce by key para sumar los valores de `is_active`, `consent` y la cantidad total de combinaciones por cada (método de pago, segmento). Finalmente, se hace un map para calcular los porcentajes de clientes activos y de consentimiento de marketing por cada combinación (método de pago, segmento), y se formatea el resultado en un objeto Row para facilitar su visualización.

```
+----------------+----------------+------------+-------------+------------+-------------+
|  payment_method|customer_segment|active_count|consent_count|active_prcnt|consent_prcnt|
+----------------+----------------+------------+-------------+------------+-------------+
|   BANK TRANSFER|          BUDGET|       16041|        12451|       89.78|        69.69|
|     CREDIT CARD|         PREMIUM|       16434|        12807|       89.87|        70.04|
|      DEBIT CARD|         REGULAR|       49130|        38295|       89.97|        70.13|
|      DEBIT CARD|          BUDGET|       16034|        12441|       89.78|        69.66|
|CASH ON DELIVERY|         REGULAR|       49140|        38307|       89.97|        70.13|
|       UNDEFINED|          BUDGET|       15806|        12263|       89.82|        69.68|
|  DIGITAL WALLET|       UNDEFINED|        8231|         6365|       89.97|        69.57|
|      DEBIT CARD|       UNDEFINED|        8234|         6369|       89.99|        69.61|
|          PAYPAL|       UNDEFINED|        8233|         6367|       89.98|        69.58|
|          PAYPAL|         PREMIUM|       16435|        12808|       89.88|        70.04|
|   BANK TRANSFER|         REGULAR|       49126|        38297|       89.96|        70.13|
|     CREDIT CARD|       UNDEFINED|        8232|         6367|       89.98|        69.59|
|          PAYPAL|         REGULAR|       49138|        38306|       89.96|        70.13|
|CASH ON DELIVERY|         PREMIUM|       16437|        12807|       89.88|        70.03|
|       UNDEFINED|       UNDEFINED|        8123|         6287|       89.98|        69.64|
|  DIGITAL WALLET|          BUDGET|       16039|        12445|       89.79|        69.67|
|CASH ON DELIVERY|       UNDEFINED|        8233|         6368|       89.97|        69.59|
|       UNDEFINED|         PREMIUM|       16172|        12606|       89.82|        70.02|
|  DIGITAL WALLET|         REGULAR|       49127|        38299|       89.97|        70.14|
|      DEBIT CARD|         PREMIUM|       16442|        12810|       89.89|        70.03|
+----------------+----------------+------------+-------------+------------+-------------+
only showing top 20 rows
```

Listing 9: Resultados de la consulta 3 propuesta por el enunciado

3.1.4. Para los productos que contienen en su descripción la palabra 'stuff', calcular el peso total de su inventario agrupado por marca

Para esta consulta se tomó una dos consideracioes:

- Se consideran solo los productos que tienen un valor definido en la columna `brand`.
- Se consideran solo los productos que tienen un valor mayor a cero y no nulo en la columna `weight_kg`.

Los productos que contienen la palabra 'stuff' en su descripción son aquellos que tienen la palabra 'stuff' en cualquier parte de la cadena de texto, sin diferenciar mayúsculas de minúsculas. Esto se identifica al momento de cargar los datos, creando una nueva columna booleana `contains_stuff` en la tabla de productos en el mapeo inicial de las columnas a través de la función mostrada en el listado 10.

```python
def retain_products_columns(row: Row):
    brand = "UNDEFINED" if row.brand is None else row.brand.strip().upper()
    weight = 0.0 if row.weight_kg is None else row.weight_kg
    stock = 0 if row.stock_quantity is None else row.stock_quantity
    is_stuff = False if row.description is None else ("STUFF" in row.description.upper())
    return (
        brand,
        weight,
        stock,
        is_stuff,
    )
```

Listing 10: Mapeo de columnas inicial para la consulta 4

Luego, para resolver la consulta simplemente se hace un filtro usando la columna recién generada `is_stuff`, y se hace un map para crear una tupla con la marca como clave y el peso como valor. Finalmente, se hace un reduce by key para sumar los pesos por marca, y se utiliza la acción `takeOrdered` para obtener las 5 marcas con mayor peso total.

```python
stuff_products_weight_by_brand = productsRDD \
    .filter(lambda x: (
        x[productsIdx["is_stuff"]] and x[productsIdx["brand"]] != "UNDEFINED"
    )) \
    .map(lambda x: (x[productsIdx["brand"]], x[productsIdx["weight"]])) \
    .reduceByKey(lambda a, b: a + b)

heaviest_brands = stuff_products_weight_by_brand.takeOrdered(5, key=lambda x: -x[1])
```

Listing 11: Resolución de la consulta 4 propuesta por el enunciado

```
Las 5 marcas con mayor peso en productos cuya descripcion contiene 'Stuff' son:
Marca: 3M, Peso total: 4250.86
Marca: WAYFAIR, Peso total: 4080.17
Marca: ADIDAS, Peso total: 4057.34
Marca: NIKE, Peso total: 3614.96
Marca: HASBRO, Peso total: 3338.58
```

Listing 12: Resultados de la consulta 4 propuesta por el enunciado

3.1.5. Calcular el porcentaje de productos cuyo stock es al menos 20 % más alto que el stock promedio de su marca

Para esta consulta se tomó una única consideración: se consideran solo los productos que tienen un valor definido en la columna `brand`.

Primero se calcula el stock promedio por marca, haciendo un map para crear una tupla con la marca como clave y una tupla con el stock y 1 como valor. Luego se hace un reduce by key para sumar los stocks y las cantidades por marca, y finalmente se hace un mapValues para calcular el promedio de stock por marca.

Luego, se hace un join entre el RDD original de productos y el RDD con el stock promedio por marca, utilizando como clave la marca. Luego se hace un map para crear una tupla con dos valores: un indicador que vale 1 si el stock del producto es al menos 20 % mayor que el stock promedio de su marca, y 0 en caso contrario; y un contador que vale 1 para cada producto. Finalmente, se hace una reducción para sumar ambos valores, y se calcula el porcentaje dividiendo la cantidad de productos con stock alto por la cantidad total de productos.

```
avg_stock_by_brand = productsRDD \
    .filter(lambda x: x[productsIdx["brand"]] != "UNDEFINED") \
    .map(lambda x: (x[productsIdx["brand"]], (x[productsIdx["stock"]], 1))) \
    .reduceByKey(lambda a, b: (a[0] + b[0], a[1] + b[1])) \
    .mapValues(lambda x: x[0]/x[1])

high_stock_prods_and_total = productsRDD\
    .map(lambda x: (x[productsIdx["brand"]], x[productsIdx["stock"]])) \
    .join(avg_stock_by_brand) \
    .map(lambda x: ( # (brand, (stock, avg_stock))
        1 if x[1][0] > x[1][1]*1.2 else 0,  # productos con stock > 20% del promedio
        1
    ))\
    .reduce(lambda a, b: (a[0] + b[0], a[1] + b[1]))

result = high_stock_prods_and_total[0] / high_stock_prods_and_total[1] * 100
```

Listing 13: Resolución de la consulta 5 propuesta por el enunciado

s

```
El porcentaje de productos con stock mayor al 20% del promedio de su marca es: 41.63%
```

Listing 14: Resultados de la consulta 5 propuesta por el enunciado

3.1.6. Obtener la cantidad de órdenes que no hayan comprado ninguno de los 10 productos más vendidos

Para esta consulta se considera que los productos más vendidos son aquellos que tienen mayor cantidad vendida (`quantity`) entre todas las órdenes que aparecen.

Primero se buscan los 10 productos más vendidos, haciendo un map para crear una tupla con el ID del producto como clave y la cantidad vendida como valor. Luego se hace un reduce by key para sumar las cantidades vendidas por producto, y se utiliza la acción `takeOrdered` para obtener los 10 productos con mayor cantidad vendida.

Luego se hace un map en el RDD de items para crear una tupla con el ID de la orden como clave y un booleano que indica si el producto de esa fila está entre los 10 más vendidos. Luego se hace un reduce by key para combinar los booleanos por orden, utilizando una función OR para determinar si la orden contiene

al menos uno de los productos más vendidos. Finalmente, se filtra para quedarnos con las órdenes que no contienen ninguno de los productos más vendidos y se cuenta la cantidad de estas órdenes.

```
top_products_counts = itemsRDD \
    .map(lambda x: (x[itemsIdx["product_id"]], x[itemsIdx["quantity"]])) \
    .reduceByKey(lambda a, b: a + b) \
    .takeOrdered(10, key=lambda x: -x[1])

top_products_ids = [product_id for product_id, _ in top_products_counts]

not_top_products_orders = itemsRDD \
    .map(lambda x: (
        x[itemsIdx["order_id"]],
        True if x[itemsIdx["product_id"]] in top_products_ids else False
        )) \
    .reduceByKey(lambda a, b: a or b) \
    .filter(lambda x: not x[1]) \
    .count()
```

Listing 15: Resolución de la consulta 6 propuesta por el enunciado

```
La cantidad de ordenes que no contienen ninguno de los 10 productos mas vendidos es: 99507
```

Listing 16: Resultados de la consulta 6 propuesta por el enunciado

3.2. Consultas Propias

A continuación se presentan consultas adicionales para cumplir con los requerimientos del trabajo práctico, junto con las consideraciones tomadas para su resolución y los resultados obtenidos. El código fuente y los resultados completos pueden encontrarse en el notebook `consultas_propias.ipynb`.

3.2.1. Monto total recaudado por ventas de los 5 productos con más reseñas positivas.

Para esta consulta se tomaron las siguientes hipótesis:

- Se considera que una reseña es positiva cuando el rating de la misma es mayor que 3.
- Si una reseña tiene un valor nulo en el *rating*, no se considera.
- Si un *order_item* tiene valor nulo en *line_total*, el valor puede ser inferido a través del precio unitario y la cantidad comprada.

Al igual que en el TP1, si un productor tiene un valor nulo en *line_total*, el mismo se infiere a través de la siguiente fórmula:

$$line\_total = unit\_price \times quantity$$

Primero se buscan los 5 productos con más reseñas positivas. Esto se hace filtrando la tabla de reseñas para quedarse solo con las reseñas positivas, luego se mapea cada reseña positiva a un par (product_id, 1), y finalmente se reduce por clave sumando los valores y se toman los 5 productos con mayor cantidad de reseñas positivas.

Luego, con los IDs de los 5 productos obtenidos, se filtra la tabla de *order_items* para quedarse solo con los items que pertenecen a esos productos, y se mapea cada item a un par (product_id, line_total). Finalmente, se reduce por clave sumando los valores para obtener el monto total recaudado por cada uno de esos productos.

También se obtiene el nombre de cada producto para mostrarlo en los resultados. Esto se hace filtrando la tabla de productos para quedarse solo con los productos cuyos IDs están en la lista de los 5 productos con más reseñas positivas, y luego mapeando cada producto a un par (id, name). Finalmente, se convierte el resultado en un diccionario para facilitar la muestra de resultados.

```
top_5_products = reviewsRDD \
    .filter(lambda row: row[reviewsIdx["rating"]] > 3) \
    .map(lambda row: (row[reviewsIdx["product_id"]], 1)) \
    .reduceByKey(lambda a, b: a + b) \
    .takeOrdered(5, key=lambda x: -x[1])

top_5_products_ids = [prod[0] for prod in top_5_products]

top_5_products_sells = itemsRDD \
    .filter(lambda row: row[itemsIdx["product_id"]] in top_5_products_ids) \
    .map(lambda row: (row[itemsIdx["product_id"]], row[itemsIdx["line_total"]])) \
    .reduceByKey(lambda a, b: a + b) \
    .collect()

top_5_products_names = productsRDD \
    .filter(lambda row: row[productsIdx["id"]] in top_5_products_ids) \
    .map(lambda row: (row[productsIdx["id"]], row[productsIdx["name"]])) \
    .collectAsMap()
```

Listing 17: Resolución de la consulta 1 propuesta propia.

```
Top 5 productos con mas resenas positivas y monto de sus ventas totales:
Producto Fully-configurable high-level circuit: $8320.50
Producto Persevering logistical help-desk: $12220.00
Producto Innovative solution-oriented installation: $291.20
Producto Seamless radical architecture: $13547.28
Producto Robust cohesive utilization: $271.51
```

Listing 18: Resultados de la consulta 1 propuesta propia

3.2.2. Durante 2024 ¿Qué porcentaje de las órdenes 'REFUNDED' fueron órdenes con descuento? ¿La mayoría eran de usuarios activos? ¿Qué segmento de usuario realizó la mayor cantidad de reembolsos?

Para esta consulta se tomaron las siguientes hipótesis:

- Si el valor del campo *discount_amount* en *orders* es nulo, se asume que la orden no tuvo descuento.
- Si el usuario de una orden no está en la tabla de *customers*, se asume que no es usuario activo.

Para resolver la consulta primero se filtran las órdenes para quedarse solo con las que tienen estado 'REFUNDED' y fueron realizadas en 2024. Luego, se hace un *left outer join* con la tabla de clientes para obtener la información del cliente asociado a cada orden. Después, se mapea cada orden a una tupla que contiene el monto del descuento, el segmento del cliente y si el cliente es activo o no. En este map se tiene en cuenta de considerar los datos de clientes que podrían ser nulos debido al *left outer join*. Finalmente, se cachea el resultado para optimizar las siguientes operaciones.

Luego, se mapea cada fila del RDD cacheado a una tupla que contiene tres valores: si la orden tuvo descuento (1 si tuvo descuento, 0 si no), si el cliente es activo (1 si es activo, 0 si no), y un contador de 1 para contar el total de órdenes. Se reduce por clave sumando los valores para obtener el total de órdenes con descuento, el total de órdenes de usuarios activos y el total de órdenes. Con estos totales, se calcula el porcentaje de órdenes con descuento y el porcentaje de órdenes de usuarios activos.

Finalmente, para obtener el segmento de usuario que realizó la mayor cantidad de reembolsos, se mapea cada fila del RDD cacheado a un par (segmento, 1) y se reduce por clave sumando los valores. Luego, se reduce el resultado para quedarse con el segmento que tiene la mayor cantidad de reembolsos.

```
orders_user_active_and_segment = ordersRDD \
    .filter(lambda row: (
        row[ordersIdx["status"]] == statuses.value["REFUNDED"]
        and row[ordersIdx["year"]] == 2024
        )) \
    .map(lambda row: (row[ordersIdx["customer_id"]], row)) \
    .leftOuterJoin(customersRDD.map(lambda row: (row[customersIdx["id"]], row))) \
    .map(lambda row: (
        row[1][0][ordersIdx["discount_amount"]],
        row[1][1][customersIdx["segment"]] if row[1][1] is not None \
            else bc_segment_id_to_name.value["UNDEFINED"],
        row[1][1][customersIdx["is_active"]] if row[1][1] is not None else False,
    )).cache()

discount_total_and_active_users = orders_user_active_and_segment \
    .map(
        lambda row: (
            1 if row[0] > 0 else 0, # tiene descuento
            1 if row[2] else 0, # es de usuario activo
            1,  # ordenes totales
        )
    ).reduce(lambda a, b: (a[0] + b[0], a[1] + b[1], a[2] + b[2]))

discount_refunded_orders_percentaje = \
    (discount_total_and_active_users[0] / discount_total_and_active_users[2]) * 100

active_user_percentaje = \
    (discount_total_and_active_users[1] / discount_total_and_active_users[2]) * 100

most_refunded_segment = orders_user_active_and_segment \
    .map(lambda row: (row[1], 1)) \
    .reduceByKey(lambda a, b: a + b) \
    .reduce(lambda a, b: a if a[1] > b[1] else b)
```

Listing 19: Resolución de la consulta 2 propuesta propia.

```
El 21.29% de las ordenes REFUNDED durante 2024 fueron ordenes con descuento.
La mayoria eran de usuarios activos.
El segmento que mas ordenes REFUNDED tuvo fue REGULAR con 7284 ordenes.
```

Listing 20: Resultados de la consulta 2 propuesta propia

3.2.3. ¿Cuáles son las 3 marcas que vendieron menos unidades de productos durante 2025? Mostrar los nombres de los productos que más ingresos generaron de esas marcas.

Para esta consulta se toma una única consideración: No se tienen en cuenta para este análisis las ventas cuyo producto no está registrado en la tabla de *products*.

Primero se hace un *inner join* entre las tablas de *order_items* y *products* para obtener la información de la marca de cada producto vendido. Luego, se mapea cada fila del resultado a un par (marca, cantidad vendida) y se reduce por clave sumando las cantidades vendidas para obtener el total de unidades vendidas por cada marca. Finalmente, se toman las 3 marcas con menos unidades vendidas.

Luego, se filtra la tabla de *order_items* para quedarse solo con los items que pertenecen a las marcas obtenidas en el paso anterior. Se mapea cada item a un par ((marca, product_id), line_total) y se reduce por clave sumando los valores para obtener el total de ingresos generados por cada producto de esas marcas.

Luego, se mapea el resultado a un par (marca, (product_id, total_ingresos)) y se reduce por clave para quedarse con el producto que generó más ingresos para cada marca.

Finalmente, se obtiene el nombre de cada producto para mostrarlo en los resultados. Esto se hace filtrando la tabla de productos para quedarse solo con los productos cuyos IDs están en la lista de productos que generaron más ingresos para las marcas con menos ventas, y luego mapeando cada producto a un par (id, name). Finalmente, se convierte el resultado en un diccionario para facilitar la muestra de resultados.

```
items_products_joined = itemsRDD \
.map(
    lambda row: (
        row[itemsIdx["product_id"]],
        (row[itemsIdx["quantity"]], row[itemsIdx["line_total"]])
    )
).join(productsRDD.map(
    lambda row: (row[productsIdx["id"]], (row[productsIdx["brand"]]))
))

less_sells_brands = items_products_joined.map(
    lambda row: (
        row[1][1], # brand
        row[1][0][0], # quantity
    )
).reduceByKey(lambda a, b: a + b) \
.takeOrdered(3, key=lambda x: x[1])

less_sells_brands_names = [brand[0] for brand in less_sells_brands]

brand_sells = itemsRDD.map(
    lambda row: (row[itemsIdx["product_id"]], row[itemsIdx["line_total"]])
).join(productsRDD.map(
    lambda row: (row[productsIdx["id"]], row[productsIdx["brand"]]))
)

top_products_per_brand = brand_sells.filter(
    lambda row: row[1][1] in less_sells_brands_names
).map(
    lambda row: ((row[1][1], row[0]), row[1][0])  # ((brand, product_id), line_total)
).reduceByKey(lambda a, b: a + b) \
.map(
    lambda row: (row[0][0], (row[0][1], row[1]))  # (brand, (product_id, total_line))
).reduceByKey(lambda a, b: a if a[1] > b[1] else b) \
.collect()

top_products_per_brand_ids = [prod[1][0] for prod in top_products_per_brand]
top_products_per_brand_names = productsRDD.filter(
    lambda row: row[productsIdx["id"]] in top_products_per_brand_ids
).map(
    lambda row: (row[productsIdx["id"]], row[productsIdx["name"]])
).collectAsMap()
```

Listing 21: Resolución de la consulta 3 propuesta propia.

```
Las marcas con menos unidades vendidas en 2025 son:
 - APPLE: 4942 unidades vendidas
 - ASHLEY FURNITURE: 5132 unidades vendidas
 - CASTROL: 5135 unidades vendidas

El producto mas vendido de cada una de esas marcas es:
- APPLE: FACE-TO-FACE TANGIBLE STRATEGY con $133905.24 en ventas
- ASHLEY FURNITURE: Distributed interactive neural-net con $30814.72 en ventas
- CASTROL: Grass-roots directional success  con $83496.97 en ventas
```

Listing 22: Resultados de la consulta 3 propuesta propia

3.2.4. Rating promedio de los productos pertenecientes a la categoría más vendida.

Primero se hace un *inner join* entre las tablas de *order_items* y *products* para obtener la información de la categoría de cada producto vendido. Luego, se mapea cada fila del resultado a un par (categoría, monto de la venta) y se reduce por clave sumando los montos para obtener el total de ventas por cada categoría. Finalmente, se reduce el resultado para quedarse con la categoría que tiene el mayor monto de ventas y se obtiene su nombre de la tabla de categorías.

Luego, se filtra la tabla de productos para quedarse solo con los productos que pertenecen a la categoría obtenida en el paso anterior. Se mapea cada producto a su ID y se colecta el resultado en una lista. Luego, se filtra la tabla de reseñas para quedarse solo con las reseñas cuyos product_id están en la lista de productos de la categoría más vendida. Finalmente, se mapea cada reseña para que quede unicamente su rating y se calcula el promedios.

```
category_sells = itemsRDD.map(
    lambda row: (row[itemsIdx["product_id"]], row[itemsIdx["line_total"]])
).join(
    productsRDD.filter(
        lambda row: row[productsIdx["category_id"]] is not None
    ).map(
        lambda row: (row[productsIdx["id"]], row[productsIdx["category_id"]])
    )
)

most_selled_category = category_sells.map(
    lambda row: (row[1][1], row[1][0])  # (category_id, line_total)
).reduceByKey(lambda a, b: a + b) \
.reduce(lambda a, b: a if a[1] > b[1] else b)

most_selled_category_name = categoriesRDD.filter(
    lambda row: row[categoriesIdx["id"]] == most_selled_category[0]
).first()[1]

category_products = productsRDD.filter(
    lambda row: (
        row[productsIdx["category_id"]] is not None
        and row[productsIdx["category_id"]] == most_selled_category[0]
    )
).map(
    lambda row: row[productsIdx["id"]]
).collect()

category_mean_rating = reviewsRDD.filter(
    lambda row: row[reviewsIdx["product_id"]] in category_products
).map(
    lambda row: row[reviewsIdx["rating"]]
).mean()
```

Listing 23: Resolución de la consulta 4 propuesta propia.

```
El rating promedio de los productos pertenecientes a la categoria mas vendida es de: 3.37
```

Listing 24: Resultados de la consulta 4 propuesta propia

3.2.5. Obtener los 3 productos ‘ELECTRONICS‘ con más movimientos por daños. Mostrar el cambio en la cantidad total para esos productos, y el promedio de cambios en la cantidad para los movimientos dañados.

Se toma una sola consideración: un producto es ‘ELECTRONICS‘ si su categoría padre es ‘ELECTRONICS‘.

Primero se filtra la tabla de categorías para quedarse solo con las categorías que tienen como padre a ‘ELECTRONICS‘, y se mapea cada categoría a su ID. Luego, se filtra la tabla de productos para quedarse solo con los productos que pertenecen a las categorías obtenidas en el paso anterior, y se mapea cada producto a su ID. Ambos resultados se colectan en listas.

Luego, se filtra la tabla de inventario para quedarse solo con los movimientos que tienen como razón ‘DAMAGE‘, y se cachea el resultado para optimizar las siguientes operaciones. Se filtra el RDD cacheado para quedarse solo con los movimientos cuyos product_id están en la lista de productos ‘ELECTRONICS‘. Se mapea cada movimiento a un par (product_id, (1, quantity)) y se reduce por clave sumando los valores para obtener el total de movimientos por daño y el total de cambios en la cantidad para cada producto. Se toman los 3 productos con más movimientos por daños y se obtienen sus nombres de la tabla de productos.

Finalmente, se calcula el promedio de cambios en la cantidad para los movimientos dañados mapeando cada movimiento del RDD cacheado a su cantidad y calculando el promedio.

```
electronics_categories_id = categoriesRDD.filter(
    lambda row: row[categoriesIdx["parent"]] == "ELECTRONICS"
).map(
    lambda row: row[categoriesIdx["id"]]
).collect()

electronics_products_ids = productsRDD.filter(
    lambda row: (
        row[productsIdx["category_id"]] is not None
        and row[productsIdx["category_id"]] in electronics_categories_id
    )
).map(
    lambda row: row[productsIdx["id"]]
).collect()

damaged_movements = inventoryRDD.filter(
    lambda row: row[inventoryIdx["reason"]] == reasons.value["DAMAGE"]
).cache()

damaged_electronics_movements_by_product = damaged_movements \
.filter(lambda row: row[inventoryIdx["product_id"]] in electronics_products_ids) \
.map(
    lambda row:
        (row[inventoryIdx["product_id"]], (1, row[inventoryIdx["quantity"]]))
).reduceByKey(lambda a, b: (a[0] + b[0], a[1] + b[1]))

most_damaged_products = damaged_electronics_movements_by_product \
    .takeOrdered(3, key=lambda x: -x[1][0])
most_damaged_products_ids = [prod[0] for prod in most_damaged_products]

most_damaged_products_names = productsRDD.filter(
    lambda row: row[productsIdx["id"]] in most_damaged_products_ids
).map(
    lambda row: (row[productsIdx["id"]], row[productsIdx["name"]])
).collectAsMap()

damaged_movements_quantity_mean = damaged_movements.map(
    lambda row: row[inventoryIdx["quantity"]]
).mean()
```

Listing 25: Resolución de la consulta 5 propuesta propia.

```
Los 3 productos ELECTRONICS con mas movimientos por danos son:
ID      Total Movs.     Cambio tot. en Cant.    Nombre
909144  4               349                     Self-enabling discrete open system
986689  4               327                     UNDEFINED
969306  4               -647                    Pre-emptive zero tolerance encryption

El promedio de cambios en la cantidad para los movimientos danados es de -0.00 unidades.
```

Listing 26: Resultados de la consulta 5 propuesta propia

Notar que en el resultado de la consulta 5, el promedio de cambios en la cantidad para los movimientos dañados es -0.00 unidades. Esto se debe a que, como se identificó en el TP1, hay movimientos con cantidades positivas y negativas que se cancelan entre sí, resultando en un promedio cercano a cero.

También puede verse que los tres productos con más movimientos por daños empatan en 4 movimientos cada uno. Si se realiza el chequeo, se puede observar que en realidad hay varios productos que empatan en 4 movimientos.

4. Anexo