TAREA 2 - YAML Y CREACIÓN DE CSV - BIG DATA

Estudiante: Yoksan Varela Cambronero

27 de Junio, 2024

1. Contenido del archivo comprimido

Dentro del archivo comprimido se encuentran, además de este documento, los siguientes elementos:

1. **Folder datasets:** Contiene los cinco archivos YAML solicitados en la sección "Datos de Entrada"del enunciado de la tarea 2. Estos fueron creados usando ChatGPT y con algunos ajustes manuales, usando como referencia el ejemplo provisto en el instructivo de la tarea.

   Cada uno de estos archivos hace referencia a una caja registradora distinta, con al menos 10 compras y con una variedad de productos de supermercado. Para efectos de esta tarea, los archivos no cuenta información incompleta en las secciones requeridas. Por efecto de tiempo, no se pudo realizar la parte opcional de la tarea, por lo tanto, estos son todos los archivos YAML a ser usados en la ejecución de la tarea.

2. **Folder library:** Este folder contiene 4 archivos de Python que contienen todas funciones utilizadas a lo largos del código MAIN. En la sección *Funciones y sus respectivas Prueba Unitarias* de este documento se explica con más detalle las funciones y las pruebas unitarias asociadas a las mismas.

3. **Folder unitary_tests:** De firma análoga al folder anterior, en este se almacenan todas las pruebas unitarias para las funciones mencionadas anteriormente. Para más información detallada sobre estas funciones, referirse a la sección *Funciones y sus respectivas Prueba Unitarias.*

4. **Folder Results:** Este folder no existe dentro del archivo comprimido, pero será generado por el código MAIN a la hora de su ejecución. Este va a contener los archivos CSV solicitados en la tarea, pero podrían estar corruptos. Para mayor detalle, ver nota importante en el primer punto en la sección *Decisiones de diseño de la solución.*

5. **Archivo Big Data - Tarea 2.pdf:** Enunciado de la tarea provisto por el profesor.

6. **Archivo build_run_docker.bat:** Este documento fue creado para facilitar la creación y la ejecución del contenedor de Docker en mi computadora personal, la cual cuenta con el OS Windows 11. Internamente, este archivo tiene las siguentes líneas: *docker build –tag tarea2_yoksanvarela .* y *docker run -i -t tarea2_yoksanvarela /bin/bash.*

7. **Archivo command_list.txt:** Es un archivo plano de texto que contiene las líneas necesarias para poder correr el programa principal y las pruebas unitarias. Cabe la pena rescatar que estos comandos hay que correrlos de forma manual en la consola del contenedor de Docker, y son los mismos comandos que se citan en la sección *Ejecución del MAIN y las Pruebas Unitarias.*

8. **Archivo Dockerfile:** Es la imagen usada en Docker, la cual cuenta con cambios con respecto a la usada en la tarea 1. Los cambios son:

   a) La inclusión de *apk add build-base* y *python3 -m ensurepip.* Estas fueron necesarias para poder instalar Numpy. Inicialmente se tenía planeado usar funciones de Numpy para calcular los percentiles, pero al final se decidió usar un proceso más manual para esa sección.

   b) Además, se incluyó *pyyaml y numpy* en la línea de instalacion con pip3. Pyyaml es la libreria seleccionada para transformar los archivos YAML en objectos tipo diccionario que fueron procesados posteriormente para ser convertidos en Spark dataframes.

9. **Archivo tarea2_program_YoksanVarela.py:** Es el programa principal donde se resuelve el problema planteado en la tarea 1.

2. Decisiones de diseño de la solución

A continuación se listan las decisiones de diseño tomadas para el desarrollo de esta solución:

1. **La función para lectura de los archivos YAML (dentro de yaml_parser.py), cambio de nombre de los archivos CSV (dentro de tarea2_program_YoksanVarela.py) y manejo de los argumentos a la hora de llamar el código principal (args_parser.py) NO tienen pruebas unitarias:** Esto se debe a que la función de estas funciones es vital para que el código principal se pueda ejecutar o no, además de la dificultad de probarlas con elementos de memoria en lugar de archivos reales. Es por eso que el hecho que programa principal pueda ejecutarse es, en sí, la prueba a esta funciones.

   Nota importante con respecto a la función de cambio de nombre de los CSV generados por Spark: Esta es una función definida dentro del código MAIN. Como la función de creación de archivos CSV de Spark (.coalesce(1).write.csv() para generar solo un archivo CSV por dataframe) crea los archivos con nombres especiales que permiten un mejor manejo a la hora de almacenaje o lectura distribuida, fue necesario crear esta función para poder cambiarle el nombre a esos archivos y dejarlos como se solicita en el enunciado. Cuando esta función ya mencionada se utiliza es probable que el archivo copia generado de los originales quede corrupto y su contenido sea solo basura. Es por este problema que se decidió hacer un copia con el nombre correcto dentro del folder Results, pero se mantienen los folders temporales con los archivos originales. Estos son:

   a) Si el archivo **total_productos.csv** está corrupto, revisar el archivo que empieza con la palabra *part-* dentro de **./Results/tmp_prod_total**

   b) Si el archivo **total_cajas.csv** está corrupto, revisar el archivo que empieza con la palabra *part-* dentro de **./Results/tmp_cashier_total**

   c) Si el archivo **metricas.csv** está corrupto, revisar el archivo que empieza con la palabra *part-* dentro de **./Results/tmp_metrics**

2. **Líneas con NaN or NULL en el Spark dataframe principal (creado con la contatenación de cada dataframe por cada archivo YAML) de entrada son removidos antes de la ejecución del resto de funciones:** Se consideró que el objectivo de esta tarea no era lidiar con esos elementos, lo cual implica que todas las funciones de apoyo usadas en esta tarea no cuentan con lógica para lidiar con NaN NULL (esto ya fue probado en la tarea 1). No obstante, las funciones sí cuentan con manenjo de errores en el caso de faltar información importante, lógica que sí es probada en las pruebas unitarias.

3. **El uso del inglés en todos los códigos:** Aunque parezca trivial, considero importante mencionar esta decisión. De forma personal, me siento más cómodo programando en inglés (incluyendo comentarios, markdowns, etc.) dada mi experiencia laboral, además de poder hacer códigos legibles de manera universal.

   Esta práctica podría dar pie a pensar que esta solución podría ser un plagio o que fue desarrollada por otra persona, pero hago constar que el total de esta entrega fue desarrollado por mi persona.

3. Ejecución del MAIN y las Pruebas Unitarias

Estos son los pasos para poder ejecutar tanto el código principal como las pruebas unitarias:

1. Construir el contenedor:

   ```
   docker build --tag tarea2_yoksanvarela .
   ```

2. Correr el contenedor:

   ```
   docker run -i -t tarea2_yoksanvarela /bin/bash
   ```

3. Correr las pruebas unitarias: En la consola del contenedor, ejecutar los siguientes comandos:

```
pytest unitary_tests/test_yaml_parser.py
pytest unitary_tests/test_data_transformation.py
pytest unitary_tests/test_data_metrics.py
```

4. Correr el programa principal: En la consola del contenedor, ejecutar el siguiente comando (es una sola líne):

```
spark-submit tarea2_program_YoksanVarela.py ./datasets
```

5. Revisar los archivos CSV generador por el código: Revisar los archivos en el folder *Results* creado luego de la ejecución del programa principal.

4. Funciones y sus respectivas Prueba Unitarias

Las funciones usadas a lo largo del programa principal están contenidas en 5 archivos dentro de la carpeta *library*. Las pruebas unitarias están contenidas en 3 archivos dentro de la carpeta *unitary_tests*. A continuación se explican todas las funciones:

- **Función en args_parser.py:**
  - **Función:** yaml_files_loader()
  - **Descripción:** Se encarga de procesar el argumento que se pasa a la hora de ejecutar el código principal, el directorio donde están almacenados los archivos YAML. A la salida de esta función se retornan el directorio en sí y un arreglo de Python que contiene todos los nombres de los archivos YAML encontrados en el directorio.
  - **Pruebas Unitarias:** Ninguna.
- **Función en yaml_parser.py:**
  - **Función:** yaml_file_parser(yaml_file)
  - **Descripción:** Recibe un archivo YAML y retorna un objeto YAML que es, escencialmente, un diccionario de Python.
  - **Pruebas Unitarias:** Ninguna.
  - **Función:** yaml_to_spark_df(yaml_data)
  - **Descripción:** Esta funció va a tomar un objecto YAML y convertirlo en un Spark dataframe. La forma en la que lo hace es desempacando el diccionario de forma manual, y usando unos contadores para llevar la cantidad de compras y productos por caja registradora. Al final se retorna un dataframe con las siguientes columnas: *Numero_Caja*, *Numero_Compra*, *Numero_Producto*, *Nombre*, *Cantidad* y *Precio*, o bien, se retorna un FAIL de no poder realizarse este proceso.
  - **Pruebas Unitarias:** En el archivo test_yaml_parser.py:
    - **test_create_spark_df(spark_session):** Usando un YAML string se ejercita la creación de un Spark dataframe.
    - **test_no_numero_caja(spark_session):** Prueba el modo de fallo al faltar el número de caja.
    - **test_no_all_sections(spark_session):** Prueba el modo de fallo al faltar alguna de las secciones requeridas durante el desempacado.
- **Funciones en data_metrics.py:**
  - **Función:** max_min_sell(dataframe)
  - **Descripción:** Retorna los números de las cajas registradoras donde se vendió más y menos, respectivamente. Se retorna un FAIL en caso que este proceso no se pueda realizar.

- **Pruebas Unitarias:** En el archivo test_data_metrics.py:
  - **test_max_values(spark_session):** Verifica el retorno de las caja con mayor ventas y con menor ventas.
  - **test_max_values_missing_column(spark_session):** Verifica el modo de fallo cuando alguna de las columns importantes para el cálculo no está presente.

- **Función:** most_sold_product(dataframe)
- **Descripción:** Retorna cual el producto que más se vendió entre todas las cajas. Se retorna un FAIL en caso que este proceso no se pueda realizar.
- **Pruebas Unitarias:** En el archivo test_data_metrics.py:
  - **test_most_sold_product(spark_session):** Verifica el retorno del producto que más se vendió.
  - **test_most_sold_product_missing_column(spark_session):** Verifica el modo de fallo cuando alguna de las columns importantes para el cálculo no está presente.

- **Función:** most_profit_by_product(dataframe)
- **Descripción:** Retorna cual el producto que generó la mayor cantidad de ganancias o dinero por ventas. Se retorna un FAIL en caso que este proceso no se pueda realizar.
- **Pruebas Unitarias:** En el archivo test_data_metrics.py:
  - **test_most_profit(spark_session):** Verifica el retorno del producto que más ganancias generó.
  - **test_most_profit_missing_column(spark_session):** Verifica el modo de fallo cuando alguna de las columns importantes para el cálculo no está presente.

- **Función:** percentiles(dataframe)
- **Descripción:** Retorna las ventas puntuales (en dinero) que representan los percentiles 25, 50 y 75. Para hacer este cálculo, se eligió un proceso en donde se lee la cantidad total de líneas del dataframe, se divide es numero en 4 y se convierte al entero más cercano, y luego se retornan los valores de los percentiles de acuerdo a ese índice, de la siguiente forma: percentil 25 es el valor del índice como tal, el 50 es multiplicando ese índice por 2 y el 75 es multiplicando ese índice por 3. Se retornan esos tres valores indexados o bien un FAIL si el proceso no se pudo completar.
- **Pruebas Unitarias:** En el archivo test_data_metrics.py:
  - **test_percentile_calculation(spark_session):** Verifica correcto retorno de los valores que representan los percentiles deseados.
  - **test_percentile_calculation_missing_column(spark_session):** Verifica el modo de fallo cuando alguna de las columns importantes para el cálculo no está presente.

- **Funciones en data_transformation.py:**
  - **Función:** dataframe_union(dataframe1, dataframe2)
  - **Descripción:** Aplica un Union entre los dos dataframes en los parámetros, siempre y cuando ambos dataframes tengan las mismas columnas. El dataframe resultante es retornado, o bien, un estado de FAIL si no es capaz de realizar este proceso debido a que alguna de las columnas importantes no está presente.
  - **Pruebas Unitarias:** En el archivo test_data_transformation.py:
    - **test_concatenate_same_columns(spark_session):** Verifica una unión exitosa de dos dataframes con las mismas columnas.
    - **test_concatenate_different_columns(spark_session):** Prueba la prevención del caso donde las columnas no son las mismas.

  - **Función:** product_count(dataframe)

- **Descripción:** Esta función hace una agrupación usando el nombre del producto y agrega la cantidad del producto, generando una columna con la cantidad total vendida del producto entre todas las cajas registradoras. Luego el dataframe se ordena de forma descendente con respecto a la cantidad total calculada y se retorna dicho dataframe. Se retorna FAIL si este proceso no es completado con éxito.
- **Pruebas Unitarias:** En el archivo test_data_transformation.py:
  - **test_product_count_correct_sum(spark_session):** Verifica que el proceso de agrupación se complete de forma correcta y el cálculo la cantidad este bien.
  - **test_product_count_missing_column(spark_session):** Verifica el modo de fallo cuando alguna de las columns importantes para el cálculo no está presente.

- **Función:** cashier_total_sell(dataframe)
- **Descripción:** Esta función agrega una columna con el cálculo del total vendido (cantidad multiplicado por precio) y luego hace una agrupación usando el número de cajas, lo cual provoca una agregaciíon de la suma de todos los totales vendidos. Al final se ordena el dataframe de form descendiente con respecto al total vendido y se retorna dicho dataframe. Se retorna FAIL si este proceso no es completado con éxito.
- **Pruebas Unitarias:** En el archivo test_data_transformation.py:
  - **test_cashier_total_sell_correct_sum(spark_session):** Verifica que el proceso de agrupación se complete de forma correcta y el cálculo la suma este bien.
  - **test_cashier_total_sell_missing_column(spark_session):** Verifica el modo de fallo cuando alguna de las columns importantes para el cálculo no está presente.