

Resultados de prueba técnica

Desarrollador: Tomás Ignacio Garcés Chaparro.

Cargo al que postula: Data Engineer en Yapo.cl

Objetivo de la prueba: Generar un archivo Json con las series de tiempo por producto, tomando en cuenta la fecha y la cantidad de productos vendidos ese día.

Objetivo del documento: Explicar el desarrollo de la solución y responder la pregunta planteada.

Link del enunciado:

https://docs.google.com/document/d/1GP9Flu_G3bU12TlnEuJ6tyLU-aCgdfBRqPqMjlFh-EI/edit

Diseño de la solución

Dado que se cuenta con una gran cantidad de datos de ventas (superior a 6GB), distribuida en cientos de archivos, el enfoque que se escogió para analizar toda la información y registrarlo en un archivo Json comprende los siguientes pasos:

1. Unir todos los CSV en una sola tabla.
2. Reducir la información para mantener sólo lo estrictamente necesario y obtener el cálculo solicitado.
3. Escribir el resultado en un archivo de formato Json.

Para implementar la solución propuesta se tomaron las siguientes consideraciones técnicas:

- Uso de Google Colab para el desarrollo, dado que entrega un ambiente listo para el desarrollo, fácil de usar y gratuito, que además ofrece conexión a Drive.
- Uso de PySpark, dado que no sólo se cuenta con una gran cantidad de datos a procesar, sino que se plantea la posibilidad de que el tamaño de cada archivo aumente a 10GB, por lo que la arquitectura de cómputo distribuido que ofrece Spark entrega mayor escalabilidad al desarrollo (siempre y cuando se tenga acceso a un *cluster* acorde).

Código Paso a Paso

El código ha sido segmentado en cuatro macro-celdas, las cuales representan un paso del proceso, los cuales son detallados a continuación:

Paso 1: Descargas e Importaciones iniciales

```
▶ #Establece conexión a cuenta de google drive que contiene la data input
from google.colab import drive
drive.mount('/content/drive')

#Descarga de java jdk
!apt-get install openjdk-8-jdk-headless -qq > /dev/null
#Descarga Apache Spark
!wget -q https://downloads.apache.org/spark/spark-3.1.2/spark-3.1.2-bin-hadoop3.2.tgz
#Descomprime Tar
!tar xf spark-3.1.2-bin-hadoop3.2.tgz
#Instala Findspark
!pip install -q findspark

from datetime import datetime
import json
import os
#Definine Variables de Entorno
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
os.environ["SPARK_HOME"] = "/content/spark-3.1.2-bin-hadoop3.2"

import findspark
findspark.init()

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

Mounted at /content/drive

En esta primera parte del código, se prepara el ambiente para poder ejecutar el proceso sin problemas. Aquí se descargan elementos necesarios para el funcionamiento de Spark, como Java JDK , Apache Spark y FindSpark.

Se importan además las bibliotecas a utilizar:

- google.colab: acceso a carpetas de Google Drive
- datetime: para poder realizar *casteo* de variable Date a String
- json: para escritura de archivo Json
- os: para acceder a carpetas y archivos input, además de definir variables de entorno
- findspark: para que Colab encuentre las dependencias de Spark necesarias
- pyspark.sql: para manejo de Dataframes

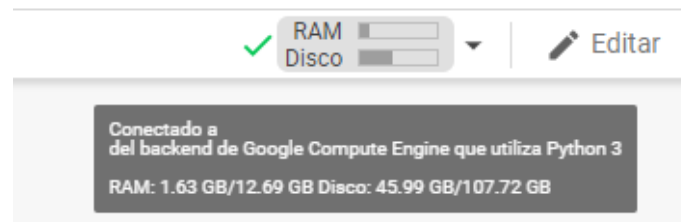
Paso 2: Definición de Funciones

En la segunda parte del código, se definen las funciones que componen el proceso. Estas son descritas a continuación:

init_spark:

```
#----- Función de inicio de sesión Spark -----  
def init_spark():  
    spark = SparkSession.builder\  
        .master("local")\  
        .appName("YapoCL")\  
        .getOrCreate()  
    return spark
```

Esta función inicia una sesión Spark para el proceso. Para esta sesión Spark, Google Colab ofrece los siguientes recursos:



data_merging:

```
#----- Funcion que unifica todos los csv en un solo Dataframe -----  
def data_merging(main_path, folders, egCSV_name):  
    for folder in folders:  
        month_files = os.listdir(main_path+folder)  
        #Lectura del CSV que contiene los headers  
        df_eg = spark.read.options(header='True', inferSchema='True', delimiter=',')\  
            .csv(main_path+folder+"/"+egCSV_name)  
  
        mySchema = df_eg.schema  
        #Elimina CSV que ya fue leído de la lista de CSV por leer  
        month_files.remove(egCSV_name)  
  
        #Construye rutas completas (path) de los csv para su lectura  
        for position in range(0, len(month_files)):  
            month_files[position] = main_path+folder+"/"+month_files[position]  
  
        #Lee la lista de CSV pendientes  
        month_df = spark.read.options(delimiter=',')\  
            .schema(mySchema)\  
            .csv(month_files)  
  
        if folder == folders[0]:  
            month_df_all = month_df.unionByName(df_eg)  
        else:  
            month_df_aux = month_df.unionByName(df_eg)  
            month_df_all = month_df_all.unionByName(month_df_aux)  
    return month_df_all
```

Función que une todos los archivos CSV en un Dataframe de Spark. Esta función accede a las carpetas definidas en la Definición de Variables y lee los archivos CSV alojados allí, los cuales son unificados en un gran Dataframe.

sales_per_product_df:

```
#----- Función que procesa el Dataframe de ventas por productos y fechas -----
def sales_per_product_df(month_df_all):
    month_df_all = month_df_all.withColumn("creation_date", to_date(col("creation_date"), "yyyy-MM-dd"))\
        .groupBy("product_name", "creation_date")\
        .agg(count(col("product_name")).alias("num_of_sales"))\
        .sort("product_name", "creation_date")

    return month_df_all
```

Función que obtiene la cantidad de ventas por día, realiza un agrupamiento de la data en base a las columnas “product_name” y “creation_date”, en base a la cual calcula la cantidad de productos vendidos en una fecha en particular.

Además, para poder ordenar las fechas de manera correcta, se realiza un *casteo* de string (formato de origen de la fecha) a date.

sales_per_product_dict

```
#----- Función que genera el diccionario de ventas por producto a partir del Dataframe previamente generado -----
def sales_per_product_dict(month_df_all, data_dict):
    product_list = month_df_all.select("product_name").distinct().collect()

    for product in product_list:

        date_list = month_df_all.filter(col("product_name")==product.product_name).select("creation_date").collect()
        sales_list = month_df_all.filter(col("product_name")==product.product_name).select("num_of_sales").collect()
        for elem in range(0, len(date_list) ):
            if elem == 0:
                data_dict[product.product_name] = {}
            data_dict[product.product_name][date_list[elem].creation_date.strftime("%Y-%m-%d")] = sales_list[elem].num_of_sales

    return data_dict
```

Para poder guardar la información procesada en un archivo Json, se almacena con la estructura requerida en el diccionario definido en el paso de “Definición de Variables”.

Esto se realiza por cada producto vendido, se obtienen las respectivas fechas de venta y cantidad de ventas de ese producto en particular, y esto es almacenado en el diccionario.

dict_to_jsonFile:

```
#----- Función que escribe archivo Json a partir del diccionario creado -----  
def dict_to_jsonFile(data_dict, file_name, main_path):  
    with open(main_path+file_name, "w") as output:  
        json.dump(data_dict, output, indent=6)
```

Finalmente, la información almacenada en el diccionario es escrita en un archivo Json, utilizando la función dump de la biblioteca importada para este objetivo.

Paso 3: Definición de Variables

```
[3] # Ruta Drive donde se encuentran los CSV  
    ruta_principal = "drive/MyDrive/Prueba Yapo/"  
  
    # Carpetas de los meses de información disponibles  
    carpetas = ["may", "june"]  
  
    # Nombre del único archivo CSV por carpeta que contiene headers  
    #(debe ser el mismo para todas las carpetas definidas)  
    headers_csvNombre = "products00.csv"  
  
    # Nombre de archivo Json de salida  
    nombre_output = "sales.json"  
  
    #Se crea diccionario con key "", el cual sera eliminado previo a exportar el archivo json  
    dataDict = {"": {}}
```

Para la ejecución del proceso se definieron 5 variables globales:

- **ruta_principal:** variable string que contiene la ruta principal donde se alojan los CSV a procesar
- **carpetas:** lista que contiene las carpetas a procesar dentro del main_path
- **dataDict:** diccionario donde se almacenará la información a escribir en Json, este tiene como Key un string y como Value otro diccionario, el cual tiene como Key un String y como valor un número
- **headers_csvNombre:** Nombre de único archivo CSV que contiene nombre de columnas
- **nombre_output:** Nombre que tendrá el archivo de salida del proceso en formato Json

Paso 4: Ejecución del algoritmo

```
[4] # Inicio de Spark
    spark = init_spark()

    # Union de CSVs
    consolidado_df = data_merging(ruta_principal, carpetas, headers_csvNombre)

    # Calculo de ventas de productos por fecha
    ventas_df = sales_per_product_df(consolidado_df)

    ventas_df = ventas_df.persist()

    # Generación del Diccionario a partir del Dataframe procesado
    dataDict = sales_per_product_dict(ventas_df, dataDict)
    #Elimina key ""
    del dataDict[""]

    # Escritura de Archivo Json
    dict_to_jsonFile(dataDict, nombre_output, ruta_principal)
```

En el paso 4 y final, se realizan los llamados a las funciones previamente definidas, entregando como parámetros las variables que correspondan.

Resultado

```
{
  "Daily Bump": {
    "2021-05-01": 34694,
    "2021-05-02": 34366,
    "2021-05-03": 34390,
    "2021-05-04": 34439,
    "2021-05-05": 34726,
    "2021-05-06": 34318,
    "2021-05-07": 34572,
    "2021-05-08": 34228,
    "2021-05-09": 34595,
    "2021-05-10": 34663,
    "2021-05-11": 34427,
    "2021-05-12": 34510,
    "2021-05-13": 34873,
```

El proceso finaliza escribiendo en un archivo Json la información procesada, con la estructura mostrada en la imagen (la imagen es sólo una pequeña parte del output).

Pregunta Teórica

¿Cómo cambiarías tu solución si cada archivo pesara 10GB?

Pensando en la escalabilidad de una solución Big Data, cambiaría el formato de los archivos a procesar dado que CSV no es considerado un formato eficiente para las grandes cantidades de datos, por lo que se propondría utilizar en su lugar formatos como Parquet o HDFS, los cuales son conocidos por optimizar el manejo de archivos en soluciones Big Data.

Para fundamentar esta respuesta, se hizo una prueba de convertir los CSV facilitados para la prueba a parquet y comparar el tamaño final de la misma información en 2 formatos diferentes.

Esto se realizó mediante el siguiente código:

▼ Bibliotecas

```
[ ] import pandas as pd
    from google.colab import drive
    import os
    drive.mount('/content/drive')
```

Mounted at /content/drive

▼ Variables

```
[ ] #Ruta de Drive donde están los csv
    main_path = "drive/MyDrive/Prueba Yapo/"

    #Carpetas donde están los csv
    folders = ["may", "june"]
```

▼ CSV a Parquet

```
[ ] for folder in folders:
    all_files = os.listdir(main_path+folder)
    for file_name in all_files:
        df = pd.read_csv(main_path+folder+"/"+file_name)
        df.to_parquet('drive/MyDrive/Parquets/'+folder+"/"+file_name[:len(file_name)-4]+".parquet")
```

Tiempo probado: 7:45 min

✓ 7 min 45 s completado a las 12:15

Este código, transformó los aproximadamente 200 archivos CSV en archivos Parquet en un tiempo de 7 minutos y 45 segundos, almacenando la información en Drive.

El resultado, una vez descargados los archivos de Drive al PC local, evidenció que en formato parquet el tamaño total es aproximadamente $\frac{1}{3}$ del tamaño de la misma información en CSV.

A modo de ejemplo la carpeta de la información de “may” en formato parquet pesa 1.07GB:

Nombre	Fecha de modificación	Tipo	Tamaño
products00.parquet	17-07-2021 9:07	Archivo PARQUET	11.318 KB
products01.parquet	17-07-2021 9:08	Archivo PARQUET	11.314 KB
products02.parquet	17-07-2021 9:08	Archivo PARQUET	11.316 KB
products03.parquet	17-07-2021 9:08	Archivo PARQUET	11.319 KB
products04.parquet	17-07-2021 9:08	Archivo PARQUET	11.313 KB
products05.parquet	17-07-2021 9:08	Archivo PARQUET	11.316 KB
products06.parquet	17-07-2021 9:08	Archivo PARQUET	11.312 KB
products07.parquet	17-07-2021 9:08	Archivo PARQUET	11.313 KB
products08.parquet	17-07-2021 9:08	Archivo PARQUET	11.312 KB
products09.parquet	17-07-2021 9:08	Archivo PARQUET	11.309 KB
products10.parquet	17-07-2021 9:08	Archivo PARQUET	11.306 KB
products11.parquet	17-07-2021 9:08	Archivo PARQUET	11.312 KB

Propiedades: may

General Compartir Seguridad Versiones anteriores Personalizar

may

Tipo: Carpeta de archivos

Ubicación: D:\Hawk 2\Prueba Yapo\Parquets\Parquets

Tamaño: 1,07 GB (1.158.446.519 bytes)

Tamaño en disco: 1,07 GB (1.158.643.712 bytes)

Contiene: 101 archivos, 0 carpetas

Por otro lado, la misma información en el formato CSV pesa 3.35GB:

Nombre	Fecha de modificación	Tipo	Tamaño
products00.csv	06-07-2021 12:39	Archivo CSV	35.229 KB
products01.csv	06-07-2021 12:39	Archivo CSV	35.222 KB
products02.csv	06-07-2021 12:39	Archivo CSV	35.219 KB
products03.csv	06-07-2021 12:39	Archivo CSV	35.209 KB
products04.csv	06-07-2021 12:39	Archivo CSV	35.220 KB
products05.csv	06-07-2021 12:39	Archivo CSV	35.228 KB
products06.csv	06-07-2021 12:39	Archivo CSV	35.196 KB
products07.csv	06-07-2021 12:39	Archivo CSV	35.220 KB
products08.csv	06-07-2021 12:39	Archivo CSV	35.205 KB
products09.csv	06-07-2021 12:39	Archivo CSV	35.208 KB
products10.csv	06-07-2021 12:39	Archivo CSV	35.204 KB
products11.csv	06-07-2021 12:39	Archivo CSV	35.230 KB

Propiedades: may

General Compartir Seguridad Versiones anteriores Personalizar

may

Tipo: Carpeta de archivos

Ubicación: D:\Hawk 2\Prueba Yapo\CSVs

Tamaño: 3,35 GB (3.605.822.102 bytes)

Tamaño en disco: 3,35 GB (3.606.016.000 bytes)

Contiene: 101 archivos, 0 carpetas

Conclusiones

El formato CSV no es recomendado para soluciones Big Data, ya que genera un gasto de espacio de almacenamiento innecesario, por lo que se recomienda que, siempre que sea posible, utilizar otro formato para el manejo de grandes cantidades de información.

Información de contacto:

En caso de preguntas o comentarios sobre la solución del ejercicio:

Nombre: Tomás Ignacio Garcés Chaparro

Correo: tomas.igarces@gmail.com