Caso Técnico – Ingeniería de Datos Comercial

Implementación de Flujo ETL para Procesamiento de Entregas.

José Carlos Porras.

Repositorio-caso-ingenieria-de-datos

Objetivo

Desarrollar un proceso robusto y parametrizable para la ingesta, transformación y carga de datos históricos de entregas.

Alcance

El proyecto abarca desde la extracción de archivos planos (CSV) hasta la generación de una capa "Silver" confiable en formatos Parquet y CSV. La solución implementa una **arquitectura de configuración centralizada basada en OmegaConf y YAML**, permitiendo la parametrización dinámica de reglas de negocio (como rangos de fechas, filtros de país y factores de conversión) sin necesidad de modificar el código fuente.

Objetivo del Caso

Diseñar e implementar un flujo ETL que permita:

- Procesar datos históricos no cargados previamente.
- Filtrar información por rango de fechas y país.
- Normalizar unidades de medida.
- Clasificar tipos de entrega.
- Garantizar calidad, consistencia y estandarización de los datos.
- Generar salidas particionadas por fecha de proceso.

Flujo de Trabajo

1. Inicialización y Parametrización

- El sistema arranca leyendo el archivo config.yaml.
- Se cargan las reglas críticas: rangos de fechas, país objetivo y factores de conversión. Si el archivo de configuración no existe, el proceso se detiene.

2. Ingesta Inteligente (Extract)

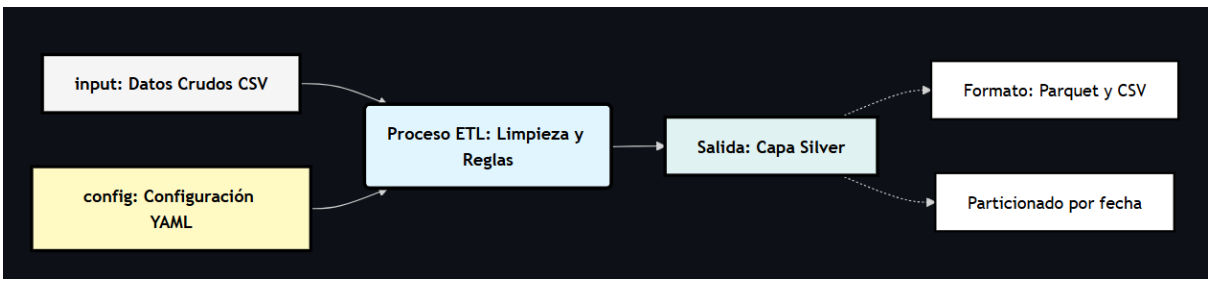
- Se conecta al archivo CSV definido.
- Se aplica un **filtrado temprano** (Push-down predicate) usando los parámetros de fecha y país. Esto asegura que solo entren al sistema los datos necesarios, optimizando el uso de memoria RAM.

3. Transformación y Calidad (Transform)

- **Normalización:** Se convierten todas las cantidades de "Cajas" a "Unidades" estándar.
- **Enriquecimiento:** Se generan columnas lógicas (entrega_rutina, entrega_bonificacion) basadas en los códigos de entrega.
- **Limpieza:** Se eliminan duplicados y se validan los tipos de datos (precios decimales, cantidades positivas).

4. Carga y Persistencia (Load)

- Se estandarizan los nombres de columnas al inglés.
- Se utiliza una estrategia híbrida con **Pandas** para escribir los resultados en disco local (Windows).
- **Resultado Final:** Se generan archivos particionados por fecha en formato **Parquet** (para consumo analítico eficiente) y se crean réplicas en **CSV** (para auditoría manual).



Parametrización del Flujo YAML

Todos los parámetros del flujo son controlados desde un archivo YAML:

- **Ruta de entrada**: Se estableció la dirección donde se debe buscar el archivo CSV para empezar a trabajar.
- **Rango de fechas:** Define el periodo de tiempo que queremos analizar.
- **País**: Se filtro para quedarnos solo con la información de una región específica (por ejemplo, "**GT**" para Guatemala), ignorando el resto.
- **Conversión de unidades** Se definió el valor **20** por el cual se deben multiplicar las cantidades marcadas como "**Cajas" (CS)** para estandarizarlas a **"Unidades" (ST).**
- **Tipos de entrega** Son las etiquetas que usa la empresa para diferenciar qué ventas son normales (Rutina) o (Bonificaciones).

```yaml
# config/config.yaml
input:
  file_path: "../data/input/data_entrega_productos.csv"

filters:
  start_date: 20250101
  end_date: 20250630
  country: "GT"
  date_format: "yyyyMMdd"

processing:
  box_size: 20
  delivery_types:
    routine: ["ZPRE", "ZVE1"]
    bonus: ["Z04", "Z05"]

output:
  base_path: "../data/processed"
  format: "parquet"
  mode: "overwrite"
  partition_col: "fecha_proceso"
```

Gestión de Configuración y Parametrización

Este bloque de código representa el **punto de entrada**. Utilizamos la librería **OmegaConf** para los parámetros de la lógica de procesamiento del archivo **config.yaml**.

1. **Definición de Ruta Relativa - config.yaml** usando una ruta relativa **(../)** para garantizar que el código funcione independientemente de la carpeta base del usuario.
2. **Carga de Objeto config.yaml**

```python
import os
from omegaconf import OmegaConf

#YAML
config_path = "../config/config.yaml"

if not os.path.exists(config_path):
    raise FileNotFoundError(f"no se encontro el file: {config_path}")

config = OmegaConf.load(config_path)
print(f"Archivo yaml leido correctamente")
```

Librerias utilizadas fueron:

Librería	Uso Principal
PySpark	Ingesta, Filtrado y Transformación Masiva.
OmegaConf	Lectura de reglas de negocio desde YAML.
Pandas	Escritura controlada de archivos locales (Parquet/CSV).

Tecnologías Utilizadas

- Python 3
- PySpark
- OmegaConf (YAML)
- Jupyter Notebook / Visual Studio Code
- GitHub (repositorio público)

Módulo de Extracción - get_data

- **Conexión:** Localiza el archivo CSV usando la ruta que definimos en el archivo de configuración **(config.yaml).**
- **Lectura Inteligente:** Utiliza **Spark** para detectar automáticamente qué tipo de dato hay en cada columna.
- **Filtrado Inmediato:** Antes de intentar hacer cálculos complejos, eliminamos todo lo que no nos sirve.

```python

def get_data(spark, config):
    #salida fecha proceso
    #(output_path: data/processed/${fecha_proceso})
    df = spark.read.option("header", True).option("inferSchema", True).csv(config.input.file_path)
    df = df.withColumn("fecha_proceso", col("fecha_proceso").cast("string"))

    return df.filter(
        (col("fecha_proceso") >= config.filters.start_date) &
        (col("fecha_proceso") <= config.filters.end_date) &
        (col("pais") == config.filters.country)
    )
```

Módulo de Transformación - transform_data

Este módulo implementa la lógica de normalización y limpieza mediante transformaciones sobre el DataFrame de Spark.

```python

def transform_data(df, config):

    df = df.dropDuplicates()

    df = df.withColumn(
        "cantidad_unidades",
        when(
            col("unidad") == "CS",
            col("cantidad") * config.processing.box_size #llamamos al parametro box_size del config.yaml que tiene como valor 20
        ).when(
            col("unidad") == "ST",
            col("cantidad")
        ).otherwise(None) # Default null si no coincide ninguna unidad
    )


    routine_types = list(config.processing.delivery_types.routine)
    bonus_types = list(config.processing.delivery_types.bonus)
    all_types = routine_types + bonus_types


    df = df.filter(
        col("tipo_entrega").isin(all_types)
    )


    df = (
        df
        .withColumn(
            "entrega_rutina",
            col("tipo_entrega").isin(routine_types)
        )
        .withColumn(
            "entrega_bonificacion",
            col("tipo_entrega").isin(bonus_types)
        )
    )

    df = df.withColumn(
        "precio",
        when(col("precio") > 0, col("precio"))
        .cast(DecimalType(10, 2))
    )

    df = (
        df
        .dropna(subset=["cantidad_unidades"])
        .filter(col("cantidad_unidades") > 0)
    )

    return df
```

1. Eliminacion de Registros Duplicados.

df = df.dropDuplicates()

- **Operación:** Eliminación de redundancia.
- **Descripción Técnica:** Se aplica una transformación para identificar y remover filas idénticas en todo el dataset, garantizando los registros antes de iniciar el procesamiento computacional.

2. Normalización Condicional de Métricas (cantidad_unidades)

```
df = df.withColumn("cantidad_unidades",
  when(col("unidad") == "CS", col("cantidad") * config.processing.box_size)
  .when(col("unidad") == "ST", col("cantidad"))
  .otherwise(None)
```

- **Condición:** (Case When).
- **Descripción Técnica:** Se estandariza la unidad de medida aplicando una lógica de ramificación:
    - **Caso CS:** Se realiza una operación aritmética de multiplicación utilizando el factor de conversión **(box_size = 20)** desde la configuración de **YAML**.
    - **Caso ST:** Se deja el valor original.
    - **(Otherwise):** Se asigna None (Null) a cualquier unidad no reconocida para facilitar su depuración o eliminación en etapas posteriores.

3. Filtrado por Pertenencia de Conjuntos (tipo_entrega)

```
all_types = routine_types + bonus_types
df = df.filter(col("tipo_entrega").isin(all_types))
```

- **Operación:** Filtrado de tipo de entrega.
- **Descripción Técnica:** Se restringe el dataset únicamente a los registros cuyo tipo_entrega pertenezca a la lista de códigos válidos concatenados desde la configuración de **YAML**.

4. Atributos booleanos

```
.withColumn("entrega_rutina", col("tipo_entrega").isin(routine_types))
.withColumn("entrega_bonificacion", col("tipo_entrega").isin(bonus_types))
```

- **Operación:** Generación de atributos booleanos.
- **Descripción Técnica:** Se crean dos nuevas columnas de tipo BooleanType evaluando la pertenencia del código de entrega a las sub-listas de "Rutina" o "Bonificación". Esto optimiza consultas.

5. Validación de Tipos y Redondeo de Decimales

when(col("precio") > 0, col("precio")).cast(DecimalType(10, 2))

- **Operación: Casteo de Tipo (*Type Casting*) con Redondeo.**
- **Descripción Técnica:**
    - **Validación:** Se aplica una condición para preservar solo valores estrictamente positivos (> 0). Los valores incorrectos se convierten en Null.
    - **Redondeo y Casteo:** Se fuerza la conversión a DecimalType(10, 2). Esto no solo cambia el tipo de dato, sino que redondea aritméticamente los valores a 2 decimales, asegurando la precisión necesaria para cálculos monetarios y evitando errores de "punto flotante".

6. Integridad de Datos (Null)

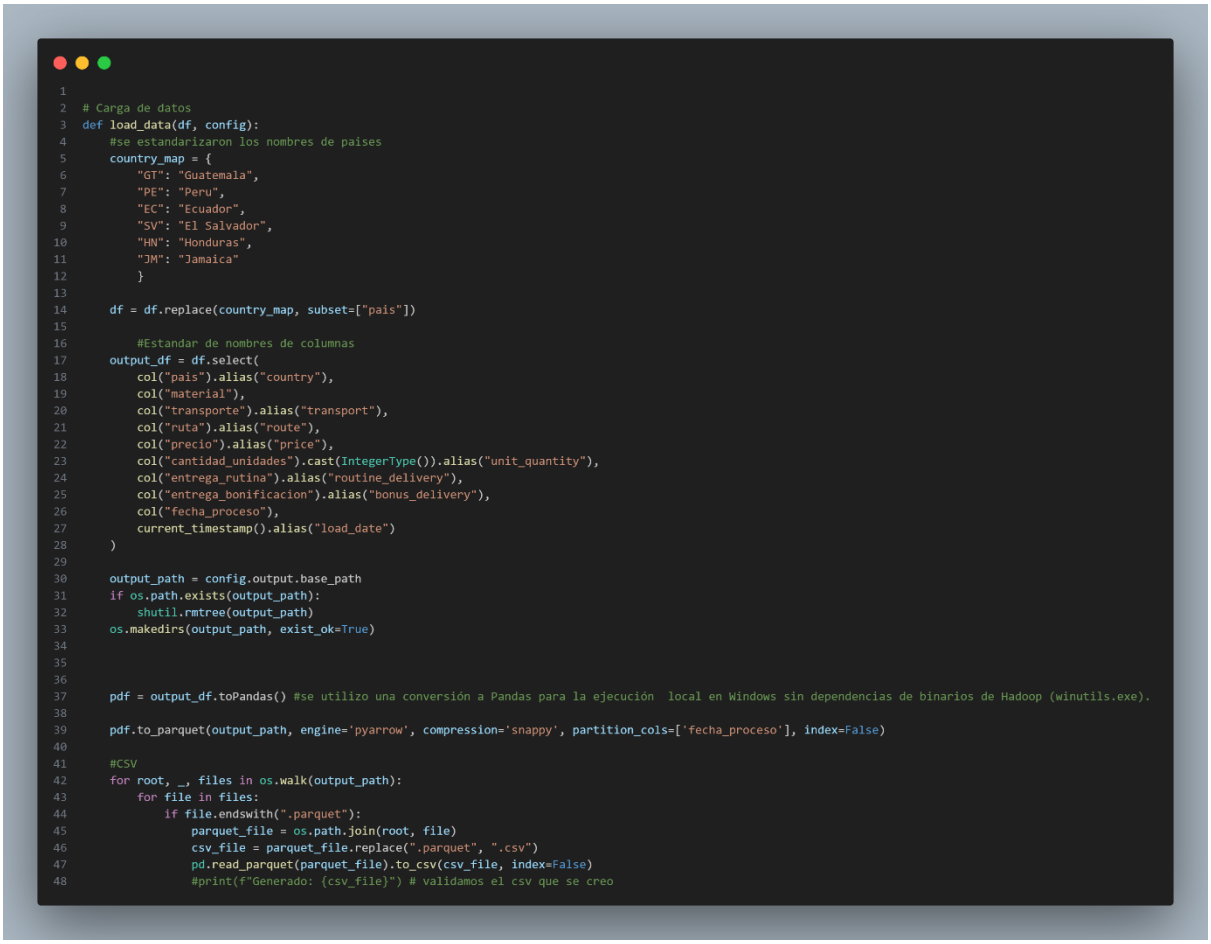
.dropna(subset=["cantidad_unidades"])

.filter(col("cantidad_unidades") > 0)

- **Operación:** Manejo de Nulos y Restricciones Lógicas.
- **Descripción Técnica:**
    - **dropna:** Elimina los registros que resultaron en **Null** durante la normalización de unidades, limpiando datos con unidades de medida inválidas.
    - **filter:** Aplica una restricción de negocio para asegurar que no existan transacciones con cantidad cero o negativa en el dataset final.

Módulo de Carga y Estandarización - (load_data)

Esta etapa final se encarga de darle formato a los datos para su consumo **("Capa Silver")**. No solo guardamos los datos, sino que los preparamos.



```python

# Carga de datos
def load_data(df, config):
    #se estandarizaron los nombres de paises
    country_map = {
        "GT": "Guatemala",
        "PE": "Peru",
        "EC": "Ecuador",
        "SV": "El Salvador",
        "HN": "Honduras",
        "JM": "Jamaica"
        }

    df = df.replace(country_map, subset=["pais"])

        #Estandar de nombres de columnas
    output_df = df.select(
        col("pais").alias("country"),
        col("material"),
        col("transporte").alias("transport"),
        col("ruta").alias("route"),
        col("precio").alias("price"),
        col("cantidad_unidades").cast(IntegerType()).alias("unit_quantity"),
        col("entrega_rutina").alias("routine_delivery"),
        col("entrega_bonificacion").alias("bonus_delivery"),
        col("fecha_proceso"),
        current_timestamp().alias("load_date")
    )

    output_path = config.output.base_path
    if os.path.exists(output_path):
        shutil.rmtree(output_path)
    os.makedirs(output_path, exist_ok=True)



    pdf = output_df.toPandas() #se utilizo una conversión a Pandas para la ejecución  local en Windows sin dependencias de binarios de Hadoop (winutils.exe).

    pdf.to_parquet(output_path, engine='pyarrow', compression='snappy', partition_cols=['fecha_proceso'], index=False)

    #CSV
    for root, _, files in os.walk(output_path):
        for file in files:
            if file.endswith(".parquet"):
                parquet_file = os.path.join(root, file)
                csv_file = parquet_file.replace(".parquet", ".csv")
                pd.read_parquet(parquet_file).to_csv(csv_file, index=False)
                #print(f"Generado: {csv_file}") # validamos el csv que se creo
```

1. **(Mapping) – Estandarizacion de Nombre de Paises:**
    - Los códigos de país son útiles para sistemas, pero confusos para el analisis. Aplicamos un mapeo (replace) para expandirlos a sus nombres completos **("Guatemala", "GT").**
2. **Estándar de Nomenclatura:**
    - Renombramos todas las columnas del español al **Inglés** usando el estándar *snake_case* (minúsculas con guiones bajos).
    - *Ejemplo:* cantidad_unidades → unit_quantity.
    - *Por qué:* Esto facilita la integración con herramientas globales de BI y equipos de desarrollo internacionales.

3. **Trazabilidad (Linaje del Dato):**
    - Agregamos la columna **load_date** con la fecha y hora exacta de ejecución **(current_timestamp)**. Esto permite saber cuándo se generó cada registro.
4. **(Spark + Pandas):**
    - Se convertimos el **DataFrame** procesado a **Pandas** para realizar la escritura física en el disco local.
    - Generamos la salida en formato **Parquet** (comprimido con Snappy) particionado por fecha, y creamos copias espejo en **CSV**.

Orquestación y Ejecución Principal - Main

Aquí se inicializa el motor de Spark, se gestiona el flujo de ejecución y, lo más importante, se controlan los errores y el ciclo de vida de la aplicación.

```python
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .appName("ETL_CASO_INGENIERIA_DATOS") \
    .master("local[*]") \
    .config("spark.sql.execution.arrow.pyspark.enabled", "false") \
    .getOrCreate()

spark.sparkContext.setLogLevel("ERROR")

try:
    # obtener datos
    df_raw = get_data(spark, config)

    if df_raw.count() == 0:
        print("No se encontraron registros que cumplan con los filtros")
    else:
        print(f"Registros encontrados: {df_raw.count()}")

    #transformacion de la data y normalizacion
        df_processed = transform_data(df_raw, config)
        #df_processed.show(5)

    #Carga de los datos
        load_data(df_processed, config)

    print("ETL REALIZADO")

except Exception as e:
    print(f"Error durante la ejecucion: {e}")

finally:
    spark.stop()
```

1. **Inicialización del Motor (Spark Session):**
    - Levantamos una sesión de Spark configurada para ejecutarse en modo local **(local[*]).**

    Decisión Técnica:

    Se utilizó **local[*]** para simular un clúster utilizando todos los hilos del CPU.

2. **Manejo de Errores - (Try-Except-Finally):**
    - Envolvemos todo el proceso en un bloque de seguridad. Si algo ocurre el se detecta el error, nos dice qué pasó y detiene el proceso de la ejecución.

3. **Validación de Flujo (Early Stop):**
    - Después de leer los datos **(get_data),** hacemos un conteo rápido. Si no hay registros que cumplan con los filtros, el proceso se detiene.

Conclusión

La solución presentada cumple integralmente con los requerimientos del caso técnico mediante un flujo ETL desarrollado en **PySpark**. La arquitectura aplica principios sólidos de ingeniería de datos, garantizando flexibilidad a través de la parametrización con **OmegaConf**.

La estrategia técnica adoptada para la ejecución local en entornos Windows: se automatizó el paso de carga utilizando **Pandas** como puente de escritura. Esta decisión permitió prescindir de la configuración compleja de binarios de Hadoop (Winutils), logrando exportar los datos particionados por fecha tanto en formato **Parquet** como en **CSV** de manera eficiente. El resultado es un sistema escalable, portable y listo para ser reutilizado en distintos contextos de cliente.