

**PRÁCTICA DE ETL**  
**(EXTRACCIÓN-TRANSFORMACIÓN-LOAD(CARGA)).**

## ÍNDICE.

INTRODUCCIÓN. ....	3
MOTIVACIÓN. ....	3
PROCESO ETL (EXTRACCIÓN → TRANSFORMACIÓN → LOAD (CARGA)). ....	5
EXTRACCIÓN & TRANSFORMACIÓN. ....	5
LOAD/CARGA.....	6
ANEXOS.....	8

## INTRODUCCIÓN.

Tienes que documentar como realizas un proceso ETL (Extracción, Transformación, Load-Carga) en tu ecosistema Hadoop (NameNode, ResourceManager, DataNode's).

Hay que tener en cuenta que tu ecosistema Hadoop está montado en un cluster de contenedores Docker Desktop.

Partirás de un archivo [2019-Nov.csv](#), y queremos cargarlo en nuestro sistema de archivos HDFS (Hadoop), pero lo más optimizado posible, es decir, que ocupe lo menos posible.

### EXTRACCIÓN.

La primera fase de tu proceso ETL. Partirás de nuestro archivo [2019-Nov.csv](#), alojado en la red o equipo repositorio. Tenemos que intentar automatizar el proceso de extracción. En nuestro caso realizaremos esa extracción mediante un script en Python.

### TRANSFORMACIÓN.

Consistirá en transformar tus datos que se encuentran en un formato \*.csv, a otro formato como: **parquet**, avro, orc.

### LOAD (CARGA).

Consistirá en cargar el archivo en formato (csv, **parquet**, avro, orc), en tu ecosistema HDFS de Hadoop.

A lo largo de este documento te iré guiando en el proceso. Junto con este documento he dejado los archivos que he necesitado para montar mi ecosistema Hadoop y los scripts para realizar este trabajo. Tú tienes que realizar un trabajo completo, donde vayas realizando tu ETL, explicado y razonado. Tendrás que entregar una memoria, junto con los archivos que hayas utilizado para el desarrollo de tu trabajo. Todo ello lo tendrás que entregar en la tarea de Moddle de la UT2 → Tarea5.

**Es evidente que no quiero que me copies directamente mis archivos y procesos. Este documento es una guía, un apoyo.**

## MOTIVACIÓN.

Tenemos archivos en formato \*.csv, \*.json, etc, de gran tamaño (TB), que necesitamos procesar, para realizar una consultas. El caso es que, es sabido que los archivos con formato Parquet son muy eficientes y efectivos para trabajar en un ecosistema Hadoop, por varias razones clave:

1. **Almacenamiento Eficiente:**

- Parquet es un formato de almacenamiento basado en columnas, lo que significa que los datos se almacenan y se leen por columnas en lugar de por filas. Esto es particularmente útil para operaciones analíticas donde típicamente se accede a un subconjunto de columnas de un gran conjunto de datos. Al almacenar datos por columnas, Parquet reduce la cantidad de datos que necesitan ser leídos para consultas específicas.

## **2. Compresión y Codificación Eficiente:**

- Parquet permite una compresión y codificación de columnas eficiente. Puede comprimir datos de manera más efectiva que los formatos basados en filas porque los datos de una misma columna son generalmente más homogéneos. Esto significa que Parquet puede utilizar técnicas de compresión que son más efectivas para tipos de datos específicos.

## **3. Integración con Herramientas de Big Data:**

- Parquet está integrado con herramientas y frameworks populares de big data como Apache Hadoop, Apache Spark y Apache Impala. Esto permite a los usuarios leer, escribir y procesar datos Parquet de manera eficiente dentro de estos ecosistemas.

## **4. Optimización de Consultas:**

- Algunos sistemas que trabajan con Parquet pueden aprovechar su formato de almacenamiento basado en columnas para optimizar las consultas. Por ejemplo, pueden omitir la lectura de datos innecesarios de columnas que no están involucradas en una consulta específica, lo que reduce el tiempo de procesamiento y la carga en el sistema de almacenamiento.

## **5. Escalabilidad y Rendimiento:**

- Parquet está diseñado para ser eficiente tanto en términos de almacenamiento como de rendimiento de consulta en grandes conjuntos de datos. Su capacidad para manejar eficientemente grandes volúmenes de datos lo hace ideal para su uso en entornos de big data como Hadoop.

## **6. Soporte para Datos Estructurados y Semi-estructurados:**

- Parquet es adecuado para trabajar con datos estructurados y semi-estructurados, lo que lo hace versátil para diferentes tipos de aplicaciones analíticas en Hadoop.

En resumen, Parquet es ideal para Hadoop debido a su eficiencia en el almacenamiento y la lectura de datos, su capacidad para integrarse con diversas herramientas de big data, y su rendimiento en el procesamiento de grandes volúmenes de datos.

Por todo esto, he decidido realizar esta guía, para realizar un ETL, para un archivo CSV → Parquet → cargarlo en HDFS.

## PROCESO ETL (EXTRACCIÓN → TRANSFORMACIÓN → LOAD (CARGA)).

### EXTRACCIÓN & TRANSFORMACIÓN.

En este proceso vamos a realizar la Extracción y la Transformación en un único paso/script de python, el cual se ejecutará de forma local en nuestro W11

La extracción. - La realizaremos mediante un script de Python desde nuestra máquina física de W11. Este script de python lo ejecutaremos en un entorno embebido o virtual de Python.

#### Paso 1: Configurar Python

##### 1. Instalar Python:

- Si aún no lo tienes, instala Python en tu sistema. Puedes descargarlo desde el [sitio web oficial de Python](#).

##### 2. Configurar un Entorno Virtual (opcional, pero recomendado):

- En la línea de comandos, navega hasta tu directorio de proyecto.
- Instalar virtualenv → `pip3 install virtualenv`
- Ejecuta `python3 -m venv envIABDXX` para crear un entorno virtual llamado **envIABDXX**.
- Dar permisos varios: `Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope Process`
- Activa el entorno virtual con **envIABDXX\Scripts\activate**

Suponemos que los datos los tenemos en una red local, donde tendremos que acceder para capturar ese archivo \*.csv y poder transformarlo a formato parquet.

La transformación. - Para convertir un archivo CSV a un archivo Parquet en Python, puedes utilizar las bibliotecas **pandas** y **pyarrow**. Aquí te proporciono un ejemplo básico de cómo hacerlo:

#### Paso 1: Instalar las Bibliotecas Necesarias

Primero, necesitas instalar **pandas** y **pyarrow**. Puedes hacerlo mediante pip:

```
pip install pyarrow
pip install pandas
```

## Paso 2: Script para Convertir CSV a Parquet

Aquí tienes un script de ejemplo que lee un archivo CSV y lo convierte a un archivo Parquet:

```
import pandas as pd

# Cambia ' F:\\IABD\\BDA\\UT2\\TAREA2\\2019-Nov.csv' por la ruta de tu
# archivo CSV
ruta_csv = 'F:\\IABD\\BDA\\UT2\\TAREA2\\2019-Nov.csv'

# Leer el archivo CSV
df = pd.read_csv(ruta_csv)



# Cambia 'F:\\IABD\\BDA\\UT2\\TAREA2\\2019-Nov.parquet' por la ruta
# donde quieres guardar el archivo Parquet
ruta_parquet = 'F:\\IABD\\BDA\\UT2\\TAREA2\\2019-Nov.parquet'

# Convertir el DataFrame a Parquet
df.to_parquet(ruta_parquet, index=False)
```

Este script realiza las siguientes acciones:

1. Lee un archivo CSV en un DataFrame de pandas.
2. Convierte el DataFrame a un archivo Parquet y lo guarda en la ruta especificada.

Lo que se busca con este ejercicio, es que veas que diferencia de tamaño-peso hay entre el archivo \*.csv y su correspondiente \*.parquet.

Nombre	Fecha de modificación	Tipo	Tamaño
 2019-Nov	09/12/2019 20:07	Archivo de valores se...	8.795.667 KB
 2019-Nov.parquet	14/12/2023 11:33	Archivo PARQUET	2.479.694 KB

Al mismo tiempo tienes que decirme cual es el tiempo de ejecución de este script.

→ Measure-Command { python ruta/a/tu/script.py }.

## LOAD/CARGA.

La carga de datos se refiere a cargar nuestros datos, ya en formato “\*.parquet”, a nuestro sistema de archivos HDFS , implementado en nuestro ecosistema Hadoop.

Bien, lo primero a tener en cuenta es donde y como tenemos implementado nuestro ecosistema Hadoop:

- NameNode

- ResourceManager
- DataNode1
- DataNode2
- DataNode3

Todo ello implementado en un cluster de Docker Desktop.

Suponemos que los datos los tenemos en una red local, donde tendremos que acceder para capturar ese archivo “\*.parquet” y poder cargarlo/load a nuestro sistema de archivos HDFS, de nuestro ecosistema Hadoop.

La carga/load. - Para la carga de un archivo “\*.parquet” a nuestro HDFS, utilizaremos un script en Python. Este script utilizará la librería de python “**hdfs**”.

Aquí te proporciono un ejemplo básico de cómo hacerlo:

### **Paso 1: Instalar las Bibliotecas Necesarias**

Primero, necesitas instalar **hdfs**. Puedes hacerlo mediante pip:

```
pip install hdfs
```

### **Paso 2: Script para cargar/load nuestro archivo parquet a HDFS**

Aquí tienes un script de ejemplo que carga un archivo parquet en un path de nuestro HDFS:

```
from hdfs import InsecureClient

# Configura la conexión con tu HDFS
hdfs_url = 'http://localhost:9870'
hdfs_user = 'hdadmin'

client = InsecureClient(hdfs_url, user=hdfs_user)

# Ruta del archivo local y la ruta de destino en HDFS
# Cambia esto con la ruta de destino en HDFS
archivo_local = 'F:\\IABD\\BDA\\UT2\\TAREA2\\2019-Nov.parquet'
ruta_destino_hdfs = '/user/hdadmin/2019-Nov.parquet'

# Cargar el archivo desde el equipo local a HDFS
try:
    with open(archivo_local, 'rb') as file:
        client.write(ruta_destino_hdfs, file)
    print(f"Archivo {archivo_local} cargado a HDFS en {ruta_destino_hdfs}")
except Exception as e:
    print(f"Error al cargar el archivo a HDFS: {e}")
```

Este script realiza las siguientes acciones:

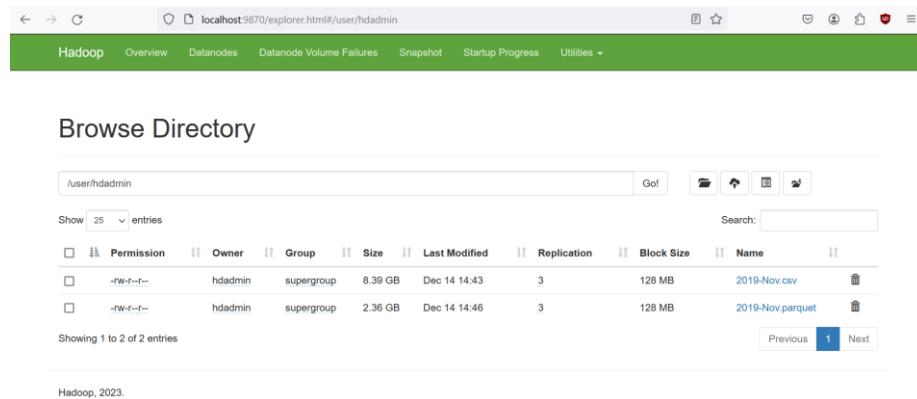
1. Lee un archivo parquet alojado en nuestro entorno local o de red y lo escribe en nuestro HDFS.

Lo que se busca con este ejercicio, es que conozcas como podemos cargar archivos en nuestro Sistemas de Archivos HDFS, mediante un script.

Al mismo tiempo tienes que decirme cual es el tiempo de ejecución de este script.

→ `Measure-Command { python ruta/a/tu/script.py }`

Cuando se haya ejecutado nuestro script, comprobamos por la UI de nuestro NameNode que tenemos el archivo parquet, alojado en nuestro HDFS



## ANEXOS.

Esta es una nota aclaratoria, referente a como hay que configurar nuestro espacio de nombres, para que se puedan ejecutar estos scripts y puedan acceder a nuestro HDFS implementado en un cluster de Docker Desktop

Agregar las direcciones de los nodos de tu cluster Hadoop a tu archivo **/etc/hosts** en Windows 11 es una estrategia común para resolver problemas de conexión con servicios que se ejecutan en Docker, especialmente cuando los servicios dentro de Docker están configurados para utilizar nombres de host específicos.

En un sistema Windows, el archivo equivalente a **/etc/hosts** en sistemas basados en UNIX se encuentra generalmente en **C:\Windows\System32\drivers\etc\hosts**. La idea detrás de modificar este archivo es proporcionar un mapeo manual entre los nombres de host y las direcciones IP, que en este caso es **127.0.0.1** (la dirección de loopback, que se refiere a tu propia máquina).

### Pasos para Editar el Archivo Hosts en Windows 11:

1. **Abrir el Bloc de Notas como Administrador:** Esto es necesario porque el archivo hosts es un archivo del sistema protegido.
2. **Abrir el Archivo Hosts:** En el Bloc de Notas, ve a Archivo -> Abrir, y navega a **C:\Windows\System32\drivers\etc**. Es posible que necesites cambiar el filtro de tipo de archivo a "Todos los archivos" para ver el archivo hosts.



3. **Agregar las Entradas:** Agrega las líneas que mencionaste al final del archivo. Por ejemplo:

```
127.0.0.1 localhost namenode datanode1 datanode2 datanode3  
::1 localhost namenode datanode1 datanode2 datanode3
```

Esto asocia los nombres de host **namenode**, **datanode1**, **datanode2**, y **datanode3** con la dirección IP **127.0.0.1** y **::1** (para IPv6).

4. **Guardar y Cerrar:** Guarda el archivo y cierra el Bloc de Notas.

#### Consideraciones:

- **Efecto en la Red:** Estas entradas harán que cualquier intento de tu máquina de Windows 11 de conectarse a **namenode**, **datanode1**, **datanode2**, o **datanode3** sea redirigido a **127.0.0.1** o **::1**. Esto es útil cuando tienes servicios en Docker que esperan comunicarse con esos nombres de host.
- **Docker y Redes:** Asegúrate de que la configuración de red de Docker esté alineada con estos cambios. Por ejemplo, si los contenedores Docker están utilizando una red personalizada, necesitas asegurarte de que la configuración de red permita esta forma de comunicación.
- **Permisos y Seguridad:** Ten en cuenta que editar el archivo hosts requiere privilegios de administrador y puede tener implicaciones de seguridad, así que hazlo con precaución.

Esta solución es específica para entornos donde los servicios Docker se comunican utilizando nombres de host que necesitan ser resueltos localmente a tu máquina. Es una práctica común en el desarrollo local y pruebas de aplicaciones que se ejecutan en contenedores Docker.

El ecosistema Hadoop, lo hemos montado con este Docker-compose.yml

```
version: '3.1'  
  
networks:  
  hadoop-net:  
    driver: bridge  
  
services:  
  
  namenode:  
    build: ./NameNode  
    image: namenode-image:latest  
    container_name: namenode  
    hostname: namenode  
    restart: always  
    command: /opt/bd/start-daemons.sh  
    ports:  
      - "9870:9870"
```

```

    - "8020:8020"
    - "9000:9000"
  networks:
    - hadoop-net
  volumes:
    - /f/mi_datos:/home
  env_file:
    - ./hadoop.env

resourcemanager:
  build: ./ResourceManager
  image: resourcemanager-image:latest
  container_name: resourcemanager
  hostname: resourcemanager
  restart: always
  command: /opt/bd/start-daemons.sh
  ports:
    - "8088:8088"
  networks:
    - hadoop-net
  volumes:
    - /f/mi_datos:/home
  environment:
    SERVICE_PRECONDITION: "namenode:9000 namenode:9870 datanode1:9864
datanode2:9864 datanode3:9864 datanode1:9866 datanode2:9866
datanode3:9866"
  env_file:
    - ./hadoop.env

dnnm1:
  build: ./DataNode-NodeManager
  image: dnnm-image:latest
  container_name: datanode1
  hostname: datanode1
  restart: always
  ports:
    - 9866:9866
    - 9864:9864
  depends_on:
    - namenode
    - resourcemanager
  command: /opt/bd/start-daemons.sh
  networks:
    - hadoop-net
  environment:
    SERVICE_PRECONDITION: "namenode:9870"
  env_file:
    - ./hadoop.env

```

```

dnnm2:
  build: ./DataNode-NodeManager
  image: dnnm-image:latest
  container_name: datanode2
  hostname: datanode2
  restart: always
  ports:
    - 9867:9866
    - 9863:9864
  depends_on:
    - namenode
    - resourcemanager
  command: /opt/bd/start-daemons.sh
  networks:
    - hadoop-net
  environment:
    SERVICE_PRECONDITION: "namenode:9870"
  env_file:
    - ./hadoop.env

dnnm3:
  build: ./DataNode-NodeManager
  image: dnnm-image:latest
  container_name: datanode3
  hostname: datanode3
  restart: always
  ports:
    - 9868:9866
    - 9865:9864
  depends_on:
    - namenode
    - resourcemanager
  command: /opt/bd/start-daemons.sh
  networks:
    - hadoop-net
  environment:
    SERVICE_PRECONDITION: "namenode:9870"
  env_file:
    - ./hadoop.env

```

Donde se menciona el archivo de configuración hadoop.env

```

CORE_CONF_fs_defaultFS=hdfs://namenode:9000
CORE_CONF_hadoop_http_staticuser_user=root
CORE_CONF_io_compression_codecs=org.apache.hadoop.io.compress.SnappyCodec
CORE_CONF_hadoop_tmp_dir=/hadoop-data
CORE_CONF_dfs_client_use_datanode_hostname=true
CORE_CONF_dfs_datanode_use_datanode_hostname=true

```

```
HDFS_CONF_dfs_webhdfs_enabled=true
HDFS_CONF_dfs_permissions_enabled=false
HDFS_CONF_dfs_namenode_datanode_registration_ip__hostname__check=false
HDFS_CONF_dfs_client_use_datanode_hostname=true
HDFS_CONF_dfs_datanode_use_datanode_hostname=true
```

Este docker-compose.yml, define un entorno Hadoop utilizando Docker Compose, configurando varios servicios relacionados con Hadoop, como Namenode, ResourceManager y DataNodes. Aquí te explico en detalle lo que hace cada parte:

### 1. Definición de la Red:

- **networks: hadoop-net:** Define una red llamada "hadoop-net" utilizando el driver **bridge**. Esto permite que los contenedores se comuniquen entre sí en una red aislada.

### 2. Servicio Namenode:

- **build: ./NameNode:** Construye la imagen del contenedor a partir del Dockerfile en el directorio **./NameNode**.
- **image: namenode-image:latest:** Asigna la imagen construida con el tag **namenode-image:latest**.
- **container\_name, hostname:** Define el nombre del contenedor y el hostname como **namenode**.
- **ports:** Expone los puertos 9870, 8020 y 9000 del contenedor a los mismos puertos en la máquina host.
- **volumes: - /f/mi\_datos:/home:** Monta el directorio **/f/mi\_datos** del host en **/home** dentro del contenedor.
- **env\_file: - ./hadoop.env:** Utiliza un archivo de variables de entorno llamado **hadoop.env**.

### 3. Servicio ResourceManager:

- Similar al Namenode, pero construye y configura el servicio ResourceManager de Hadoop.
- **ports: - "8088:8088":** Expone el puerto 8088 para la interfaz web del ResourceManager.
- **environment: SERVICE\_PRECONDITION:** Define las dependencias de servicio, es decir, espera que los servicios Namenode y Datanodes estén disponibles antes de iniciar.

### 4. Servicios DataNode-NodeManager (dnnm1, dnnm2, dnnm3):

- Configura tres contenedores para actuar como nodos de datos y administradores de nodos en el cluster Hadoop.

- Cada uno tiene su propio **container\_name** y **hostname** (datanode1, datanode2, datanode3).
- **ports**: Expone y mapea puertos para comunicaciones de DataNode.
- **depends\_on**: Establece una dependencia en los servicios namenode y resourcemanager, asegurando que estos contenedores se inicien primero.
- Comparten la misma imagen y configuración básica, pero tienen diferentes nombres y mapeos de puerto para evitar conflictos.

En resumen, este **docker-compose.yml** establece un entorno Hadoop básico con un Namenode, un ResourceManager y tres DataNodes, cada uno en su propio contenedor y comunicándose a través de una red Docker definida. La configuración permite ejecutar un cluster Hadoop distribuido en tu máquina local para fines de desarrollo y pruebas.

El archivo **hadoop.env**, en tu configuración de Docker Compose define varias variables de entorno utilizadas para configurar tu cluster Hadoop dentro de los contenedores Docker. Aquí te explico lo que hace cada línea:

1. **CORE\_CONF\_fs\_defaultFS=hdfs://namenode:9000**
  - Establece el sistema de archivos por defecto para Hadoop a HDFS y especifica que el Namenode es accesible en **namenode:9000**.
2. **CORE\_CONF\_hadoop\_http\_staticuser\_user=root**
  - Define al usuario **root** como el usuario estático para las interfaces web de Hadoop que no están autenticadas.
3. **CORE\_CONF\_io\_compression\_codecs=org.apache.hadoop.io.compress.SnappyCodec**
  - Habilita el codec de compresión Snappy para la compresión de datos dentro de Hadoop.
4. **CORE\_CONF\_hadoop\_tmp\_dir=/hadoop-data**
  - Especifica el directorio temporal de Hadoop, donde se almacenarán datos temporales y otros archivos.
5. **CORE\_CONF\_dfs\_client\_use\_datanode\_hostname=true**
  - Indica que los clientes de HDFS deben usar nombres de host para conectarse a DataNodes, en lugar de direcciones IP.
6. **CORE\_CONF\_dfs\_datanode\_use\_datanode\_hostname=true**
  - Configura los DataNodes para usar nombres de host en lugar de direcciones IP al comunicarse entre ellos.

**7. HDFS\_CONF\_dfs\_webhdfs\_enabled=true**

- Habilita WebHDFS, una interfaz REST para acceder a HDFS a través de HTTP.

**8. HDFS\_CONF\_dfs\_permissions\_enabled=false**

- Deshabilita el sistema de control de acceso basado en permisos en HDFS, permitiendo un acceso más abierto al sistema de archivos.

**9. HDFS\_CONF\_dfs\_namenode\_datanode\_registration\_ip\_\_hostname\_\_check=false**

- Desactiva la verificación de correspondencia entre la dirección IP y el nombre de host durante el registro del DataNode en el NameNode, lo que es útil en entornos donde las IP pueden cambiar o no ser consistentes.

**10. HDFS\_CONF\_dfs\_client\_use\_datanode\_hostname=true**

- Repetición de la configuración para asegurar que los clientes de HDFS usen los nombres de host de los DataNodes.

**11. HDFS\_CONF\_dfs\_datanode\_use\_datanode\_hostname=true**

- Repetición de la configuración para asegurar que los DataNodes usen sus nombres de host en la comunicación interna.

Estas configuraciones son fundamentales para el funcionamiento correcto de tu cluster Hadoop, especialmente en un entorno de contenedores donde la comunicación entre servicios y la gestión de datos pueden diferir de una instalación tradicional de Hadoop.