Documentación del Proyecto ETL sobre la base de datos World

Fecha	@1 de abril de 2025
Estado	Completado

Introducción 🚀

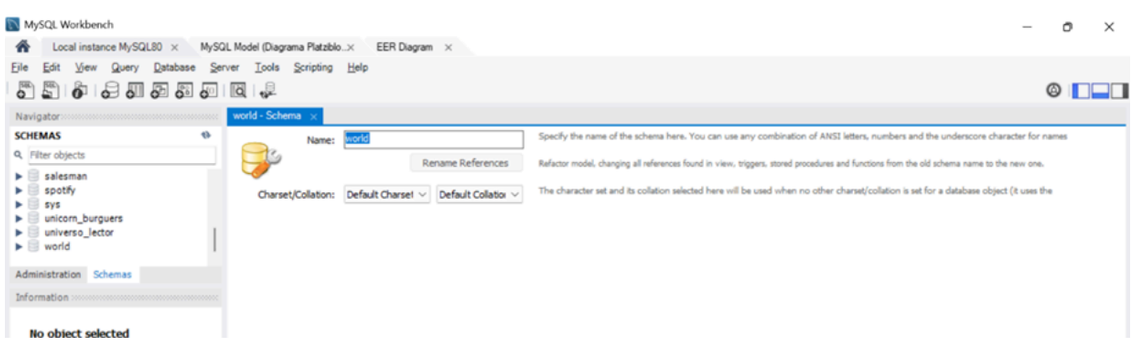
Este proyecto ETL (Extracción, Transformación y Carga) tiene como objetivo procesar los datos de la base de datos "world" de MySQL y analizarlos en Python. La implementación se realizó en Visual Studio Code (VS Code) para el desarrollo del código y Jupyter Notebook para la exploración visual de los datos.

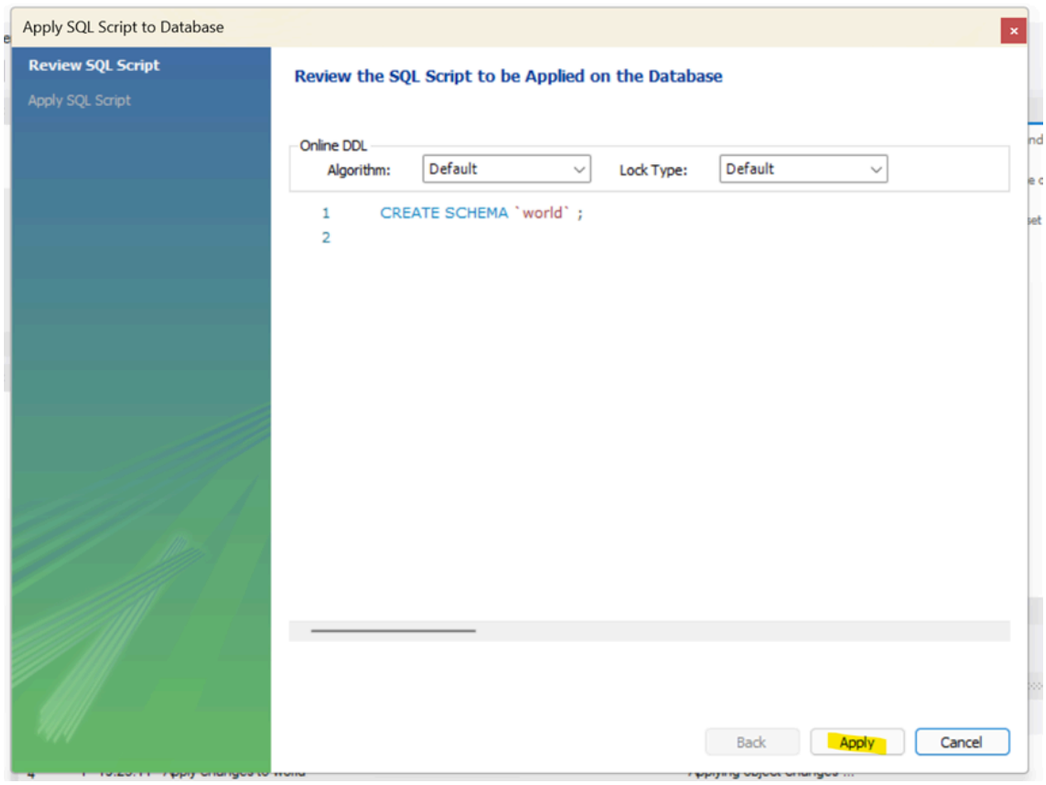
Tecnologías Utilizadas

- **Python**: Lenguaje principal para la automatización del proceso ETL.
- **MySQL**: Base de datos relacional utilizada como fuente de datos.
- **Numpy:** Para trabajar con grandes conjuntos de datos de manera eficiente. Operaciones matemáticas y estadísticas de alto rendimiento.
- **Pandas:** Para manipular y transformar los datos.
- **Seaborn (as sns)**: Es una librería basada en Matplotlib que facilita la creación de gráficos estadísticos atractivos y con estilo, proporcionando funciones más sencillas y poderosas para visualización de datos.
- **Matplotlib (as plt)**: Es una librería de gráficos en 2D para Python que permite crear una amplia variedad de gráficos estáticos, animados e interactivos, como líneas, barras, dispersión, etc.
- **Folium**: Es una librería para crear mapas interactivos utilizando `Leaflet.js` y permite visualizar datos geoespaciales de manera interactiva sobre mapas.
- **mysql-connector-python:** Para la conexión entre Python y MySQL.
- **SQLAlchemy y PyMySQL:** para Consultas SQL
- **Jupyter Notebook:** Para exploración de datos y análisis visual.
- **Git & GitHub:** Para control de versiones y colaboración.

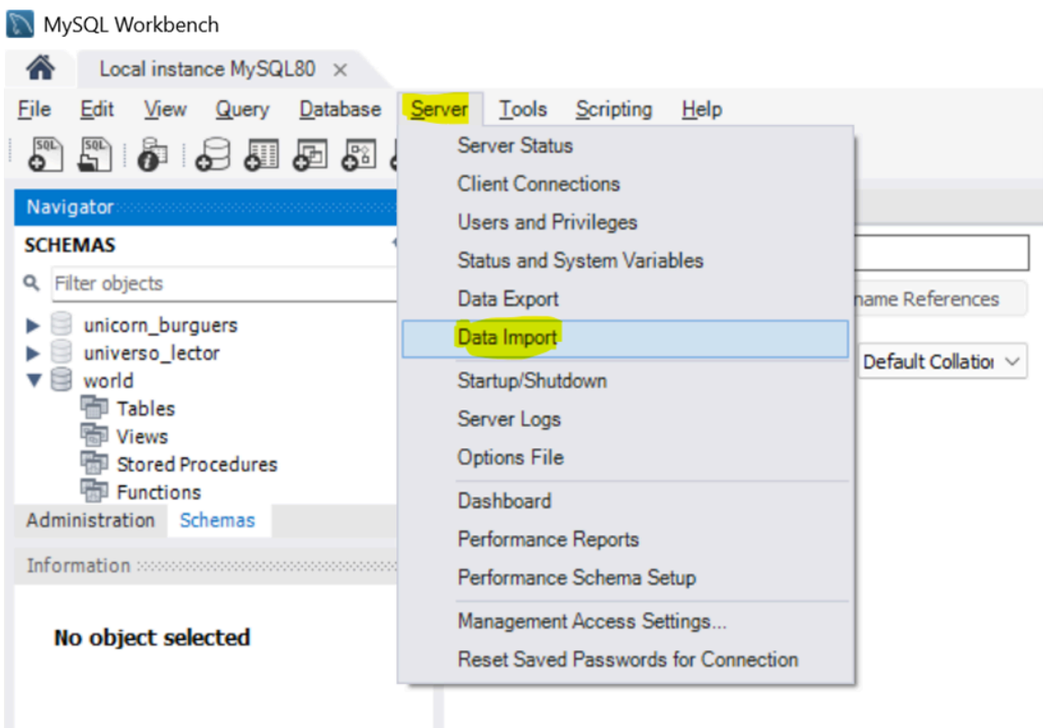
Importación de la base de datos 🗄️

1. Se crea un nuevo esquema que va a tomar por nombre `world`.

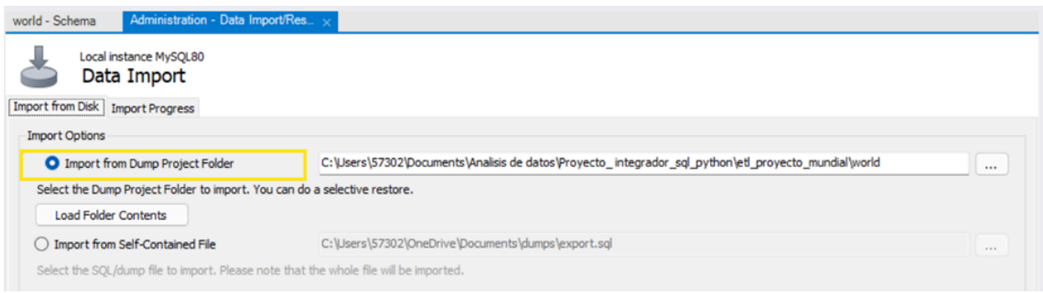




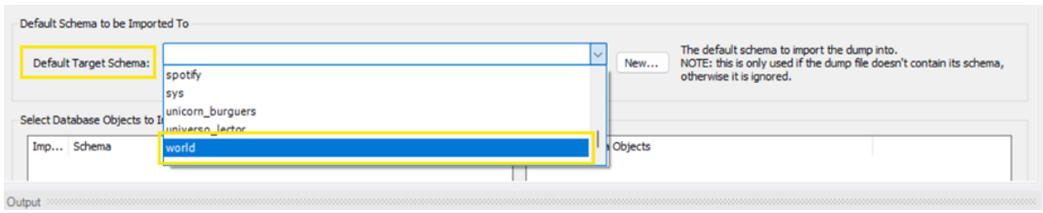
2. Luego de creado el esquema, se importa la base de datos `world` con todos los datos y estructura de las tablas:



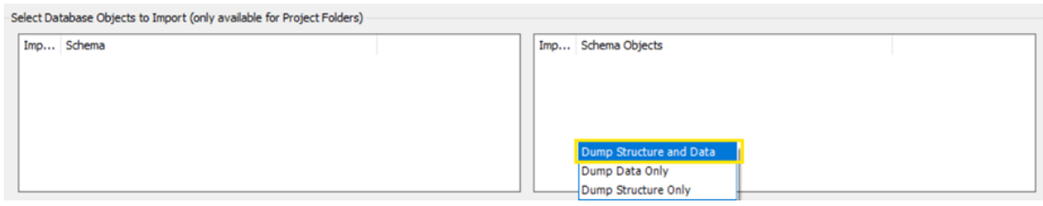
3. Se marca la opción de seleccionar carpeta y se busca la ubicación de estos archivos.



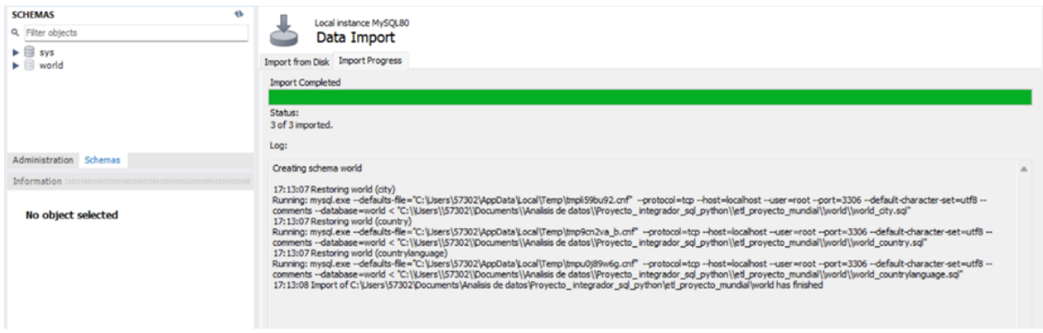
4. Posteriormente, se selecciona la base de datos `world` en donde se importarán los archivos.



5. Se marca la opción "Importar la estructura y los datos" y luego se da clic en "Start Import".



6. Por último, se da clic en el ícono de "Actualizar todo" para visualizar las tablas que se importaron.



Estructura del Proyecto 📂

El proyecto está estructurado en los siguientes archivos y carpetas:

```
├── etl_proyecto_mundial/
│   ├── etl/
│   │   ├── extraccion.py   # 📥 Módulo de extracción de datos desde MySQL
│   │   ├── transformacion.py # 🔄Procesamiento y limpieza de datos
│   │   ├── carga.py        # 📤 Exportación a CSV
│   ├── config.py         # 📌Configuración de conexión a MySQL
│   ├── main.py           # ▶️ Script principal de ejecución
│   ├── datos_csv/        # 📁 Carpeta donde se guardan los archivos CSV
│   ├── world/            # Carpeta con las tablas de la base de datos world
```

│   ├── Exploración.ipynb # 📊 Notebook para análisis exploratorio
│   ├──
README.md   # Documentación general del proyecto

Conexión a MySQL 🔗

Para centralizar la configuración de MySQL y evitar exponer credenciales en cada script, se creó el archivo `config.py`:

```
# Configuración de conexión a MySQL
MYSQL_CONFIG = {
    "host": "localhost",
    "user": "root",
    "password": "Angie830*",
    "database": "world",
    "port": 3306
}
```

✅ Separar la configuración en un archivo dedicado mejora la seguridad y facilita la reutilización del código.

Este archivo es importado en los demás scripts para establecer la conexión de manera segura:

```
from etl_proyecto_mundial.config import MYSQL_CONFIG
conexion = mysql.connector.connect(**MYSQL_CONFIG)
```

Uso de SQLAlchemy y PyMySQL para Consultas SQL 🛠️

Para ejecutar consultas SQL de manera flexible y eficiente, se utilizaron `sqlalchemy`, `mysql-connector-python` y `pymysql`. Estas librerías permiten establecer una conexión segura y realizar consultas SQL dentro de Jupyter Notebook.

Instalación de dependencias ⬇️

Antes de ejecutar las consultas, se instalaron las librerías necesarias:

```
!pip install sqlalchemy mysql-connector-python pymysql
```

✅ Uso de `sqlalchemy` para manejar conexiones de base de datos de forma más eficiente.

Conexión a MySQL con SQLAlchemy 🛜

```
from sqlalchemy import create_engine, text
from config import MYSQL_CONFIG  # Importa la configuración desde config.py

# Crear la URL de conexión
engine = create_engine(f"mysql+pymysql://{MYSQL_CONFIG['user']}:{MYSQL_CONFIG['passwor
d']}@{MYSQL_CONFIG['host']}:{MYSQL_CONFIG['port']}/{MYSQL_CONFIG['database']}")
```

✅ `create_engine` permite gestionar conexiones y ejecutar consultas sin necesidad de mantener conexiones abiertas manualmente.

Ejecución de consultas SQL desde Python 📜

```
# Función para ejecutar consultas SELECT
def ejecutar_query(query, title="Resultados"):

    pd.set_option('display.float_format', '{:,.0f}'.format)  # Configura pandas para que muestre los núm

    try:
```

```
        with engine.connect() as connection:  # Se conecta a la base de datos utilizando el engine (motc
            df = pd.read_sql(query, connection)  # Ejecuta la consulta SQL y guarda los resultados en un D

        if not df.empty:  # Verifica si el DataFrame tiene resultados (no está vacío).
            print(f"\n📊 {title}:\n(df, headers='keys', tablefmt='fancy_grid')")
            print(f"\n🔍 {title}: No se encontraron datos.")

        return df  # Devuelve el DataFrame con los resultados de la consulta.
    except Exception as err:  # Si ocurre un error durante la consulta, entra en el bloque except.
        print(f'❌ Error en la consulta: {err}')  # Imprime un mensaje de error con la descripción del prob
        return None

# Función para ejecutar consultas UPDATE, INSERT o DELETE
def ejecutar_update(query, description=""):
    try:
        # Usar text() para envolver la consulta SQL
        query = text(query)  # Se utiliza `text()` de SQLAlchemy para envolver la consulta, permitiendo su

        with engine.connect() as connection:
            connection.execute(query)  # Ejecuta la consulta SQL proporcionada (UPDATE, INSERT o DELE
            print(f"✅ {description} realizada con éxito.")  # Imprime un mensaje de éxito si la consulta se

    except Exception as err:  # Si ocurre un error durante la ejecución de la consulta, entra en el bloque
        print(f'❌ Error al actualizar: {err}')
        print(f'Consulta SQL que falló: {query}')
```

Ejemplo de consulta para obtener los primeros registros de la tabla `city`:

```
import pandas as pd
query = "SELECT * FROM city LIMIT 5;"
df_city = pd.read_sql(query, engine)
df_city
```

✅ Se pueden ejecutar consultas SQL directamente en Pandas para facilitar el análisis de datos.

Proceso ETL 🔄

1. Extracción de Datos 📥

El archivo `extraccion.py` se encarga de conectarse a la base de datos y extraer las tablas de interés, exportándolas a CSV para su posterior procesamiento:

```
import mysql.connector

def extraer_datos():
    """ Extrae los datos de todas las tablas de la base de datos 'world'y los guarda en archivos CSV
en la carpeta 'datos_csv'."""
    try:
        # Conectar a MySQL
        conexion = mysql.connector.connect(**MYSQL_CONFIG)

        if conexion.is_connected():
            print("✅ Conexión exitosa a MySQL")

            # Crear un cursor
            cursor = conexion.cursor()

cursor.execute("SHOW TABLES;")
tablas = [tabla[0] for tabla in cursor.fetchall()]
for tabla in tablas:
    query = f"SELECT * FROM {tabla};"
```

```
    df = pd.read_sql(query, conexion)
    df.to_csv(f"../datos_csv/{tabla}.csv", index=False, encoding='utf-8')
```

Nota: Un **cursor** en el contexto de bases de datos es un objeto que permite interactuar con la base de datos de manera eficiente. Es utilizado para ejecutar comandos SQL y recuperar los resultados de las consultas.

✅ Se automatiza la extracción para todas las tablas disponibles en la base de datos, lo que hace el proceso escalable.

2. Transformación de Datos 🔄

El archivo `transformacion.py` realiza diversas tareas de limpieza y enriquecimiento, como completar valores nulos y cambiar tipos de datos:

```
if 'IndepYear' in df_country.columns:
    df_country['IndepYear'] = df_country['IndepYear'].astype('Int64')
df_country.insert(df_country.columns.get_loc("IndepYear") + 1, "EsIndependizado",
                  np.where(df_country["IndepYear"].isna(), "No aplica", "Sí"))
```

✅ Se mejora la calidad de los datos al agregar una nueva columna derivada de valores nulos en `IndepYear`.

3. Carga de Datos 📤

El archivo `carga.py` toma los datos transformados y los guarda en CSV:

```
df_city.to_csv("../datos_csv/city.csv", index=False)
df_country.to_csv("../datos_csv/country.csv", index=False)
```

✅ Se asegura que los datos procesados estén almacenados de manera estructurada para su posterior análisis.

Exploración en Jupyter Notebook 📊

En `Exploración.ipynb` se cargan los CSV generados y se analizan con Pandas y Seaborn. Ejemplo de carga de datos:

```
df_city = pd.read_csv("../datos_csv/city.csv")
df_city.head()
```

✅ Se facilita la visualización y validación de los datos mediante la exploración interactiva.

Se generan visualizaciones como gráficos de distribución y mapas interactivos para entender mejor los datos.

Conclusión 🎯

Este proyecto demuestra cómo estructurar un proceso ETL de manera modular y profesional, asegurando la reutilización del código y facilitando su mantenimiento.

✅ La separación de la conexión en `config.py` permite automatizar la conexión y mejorar la seguridad del proyecto.

✅ La organización en módulos mejora la escalabilidad y facilidad de mantenimiento del código.