

Revisión de una Metodología para la Extracción, Limpieza y Organización de Datos SQL con Programación Declarativa en Python para el Desarrollo de Análisis de Datos Descriptivos.

April 22, 2024

1 Procedimientos ETL

Recordemos que los procedimientos ETL constan de Extract, Transform y Load, los cuales nos dan la pauta a seguir para seguir el ciclo de la Ingeniería de Datos. De esta forma seguimos una metodología establecida y evitamos estar trabajando sin una dirección específica.

En este ejemplo, se utilizará la base de datos de Sakila, la cual es una base de datos ejemplo proporcionada por MySQL que es utilizada comúnmente para propósitos educativos, demostraciones y pruebas. Sakila simula una base de datos de una tienda de alquiler de películas, similar a la cadena de alquiler de videos Blockbuster que solía existir.

Sakila contiene tablas que representan películas, actores, clientes, tiendas, alquileres, etc.

Para hacer uso de ella, hemos descargado el archivo script SQL a través de la página oficial de MySQL: <https://dev.mysql.com/doc/index-other.html>.

Ahora para instalar la base de datos podremos utilizar los siguientes comandos de MySQL.

```
$ mysql -u root -p sakila < sakila-schema.sql
$ mysql -u root -p sakila < sakila-data.sql
```

Después de haber ejecutado estos comandos, deberíamos ser capaces de ver la base de datos

instalada en nuestro sistema, para comprobarlo podemos ejecutar:

```
mysql> USE sakila;
Database changed
mysql> SHOW TABLES;
+-----+
| Tables_in_sakila |
+-----+
| actor              |
| actor_info         |
| address            |
| category           |
| city               |
| country            |
| customer           |
| customer_list      |
| film               |
| film_actor         |
| film_category      |
| film_list          |
| film_text          |
| inventory          |
| language           |
| nicer_but_slower_film_list |
| payment            |
| rental             |
| sales_by_film_category |
| sales_by_store     |
| staff              |
| staff_list         |
| store              |
+-----+
```

Ahora, con ello podremos comenzar a trabajar nuestro procedimiento ETL.

Primeramente, es necesario determinar que incógnita queremos responder sobre los datos disponibles. Esto variará dependiendo de la base de datos y los tipos de datos que ésta almacene. Para este caso, intentaremos responder las preguntas:

- **Patrones de alquiler:** ¿Cuáles son las películas más alquiladas?

Una vez planteadas las incógnitas, podemos determinar que tablas serán relevantes para nuestro análisis, las cuales son:

- **Patrones de alquiler:**

1. rental
2. inventory
3. film

1.1 Extract.

Para nuestro ejemplo, utilizaremos Python con SQL Alchemy.

```
from sqlalchemy import create_engine, MetaData, Table, select, func

# Configurar la conexión a la base de datos
engine = create_engine('mysql://starboy:starboyc00l@localhost/sakila')
metadata = MetaData()

# Asociar el motor a la Metadata después de su creación
metadata.bind = engine

# Reflejar las tablas
metadata.reflect(bind=engine)

# Obtener las tablas reflejadas
rental_table = metadata.tables['rental']
inventory_table = metadata.tables['inventory']
film_table = metadata.tables['film']
```

En este código hemos creado la conexión respectiva a nuestra base de datos de manera local, donde generamos un “motor” de base de datos que nos permitirá enlazarnos al motor de base de datos de MySQL, podemos pensar en él cómo una “reflejo” de la base de datos original.

1.2 Transform.

Ahora bien, después de haber extraído los datos, es necesario organizarlos de manera que sea más sencillo utilizarlos para el análisis de datos.

```
# Construir la consulta para obtener las películas más alquiladas
# Obtener las 10 películas más alquiladas
query = (
```

```

select(film_table.c.title, film_table.c.film_id,
      func.count(rental_table.c.inventory_id).label('rental_count'))
.select_from(
    rental_table.join(inventory_table, rental_table.c.inventory_id ==
                      inventory_table.c.inventory_id)
    .join(film_table, inventory_table.c.film_id == film_table.c.film_id)
)
.group_by(film_table.c.film_id)
.order_by(func.count(rental_table.c.inventory_id).desc())
.limit(10)
)

```

De esta forma, en query se almacenarán los datos extraídos de la consulta en la base de datos, de manera que con SQL Alchemy podemos escribir consultas de SQL a través del lenguaje de programación Python, agregando una capa de programación declarativa.

Para ver los resultados de la consulta anterior podemos realizar lo siguiente:

```

# Ejecutar la consulta y obtener los resultados
with engine.connect() as connection:
    result = connection.execute(query)
    for row in result:
        print(row)

```

Como respuesta tendríamos que ver algo como lo siguiente:

```

$ python3 main.py

metadata.reflect(bind=engine)
('BUCKET BROTHERHOOD', 103, 34)
('ROCKETEER MOTHER', 738, 33)
('RIDGEMONT SUBMARINE', 730, 32)
('GRIT CLOCKWORK', 382, 32)
('SCALAWAG DUCK', 767, 32)
('JUGGLER HARDLY', 489, 32)
('FORWARD TEMPLE', 331, 32)
('HOBBIT ALIEN', 418, 31)
('ROBBERS JOON', 735, 31)
('ZORRO ARK', 1000, 31)

```

1.3 Load.

Una vez que se haya realizado la Organización de los datos, es una buena idea es crear un acceso a esta información de manera que no sea necesario acceder a la información y organizarla nuevamente, para esto son muy útiles las vistas (views) de SQL.

```
# Crear una vista en la base de datos a partir de la consulta
view_name = "top_rented_films"
create_view_query = text(f"CREATE OR REPLACE VIEW {view_name} AS
    {str(query.compile(engine, compile_kwargs={'literal_binds': True}))}")

with engine.begin() as conn:
    conn.execute(create_view_query)
    print(f"Vista '{view_name}' creada exitosamente.")

# Consultar la información de la vista
view = Table(view_name, metadata, autoload_with=engine)

# Consultar la información de la vista
with engine.connect() as connection:
    result = connection.execute(select(view))
    for row in result:
        print(row)
```

2 Python y sus librerías

Python cuenta con muchas librerías para manejo de información y para diferentes actividades, por ahora nos centraremos en una específicamente, la cual es SQLAlchemy.

2.1 SQLAlchemy

SQLAlchemy es un conjunto de herramientas de software y una biblioteca de Python que facilita la comunicación entre la aplicación de Python y las bases de datos relacionales.

En otras palabras, es un **Mapeador de Objetos Relaciones** o ORM, el cual traduce las clases de Python a tablas de bases de datos relacionales y viceversa. Proporcionando una capa de abstracción sobre la base de datos que permite escribir consultas utilizando lenguaje Python en lugar de SQL.

Para instalar esta librería, requiere de ciertas dependencias, gracias a la versatilidad de Linux (Fedora), podemos hacerlo de la siguiente manera:

```
$ sudo yum install python3-devel mysql-devel gcc
$ pip install mysqlclient
```

Ahora, para manejar correctamente un proyecto de python ejecutaremos los siguientes comandos en una carpeta donde almacenaremos nuestro proyecto.

```
$ mkdir proyecto
$ cd proyecto
/proyecto$ python3 -m venv venv
/proyecto$ touch main.py
```

Con esto, ya tendremos la estructura básica de nuestro proyecto, pero será necesario utilizar en entorno virtual de Python recién creado para instalar la librería de SQLAlchemy, lo podemos realizar de la siguiente forma.

```
/proyecto$ source venv/bin/activate
(venv) /proyecto$ pip install sqlalchemy
```

De esta forma, ya tenemos la librería instalada y haremos uso de ella de la siguiente forma.

```
from sqlalchemy import create_engine

# Configurar la conexión a la base de datos

engine = create_engine('mysql://user:password@localhost/sakila')
```

3 SQL Para Manejo de Datos.

SQL - Structured Query language (lenguaje estructurado de consultas), es un lenguaje diseñado para administrar y manipular bases de datos relacionales, permite realizar una variedad de operaciones en los datos, tales como consultar, insertar, actualizar y eliminar registros en una base de datos.

En este caso, se pretende utilizar MySQL, el cual es un sistema de gestión de base de datos de código abierto que utiliza SQL como lenguaje de consulta. Este motor de base de datos es el candidato perfecto a utilizar con Python debido a la gran compatibilidad que estos tiene gracias a las librerías y dependencias para su conexión. Además, es uno de los motores de base de datos más populares en el mundo, utilizado por empresas de todos los tamaños (lo cual nos beneficia para el desarrollo de análisis de datos), y en una amplia gama de aplicaciones, desde sitios web simples

hasta sistemas empresariales complejos.

Además, la razón principal de usar SQL es debido a su gran nivel de **capacidad de ser declarativo** (lo veremos un poco más adelante).

Para el propósito de este documento, se utilizará una base de datos llamada Sakila, la cual es ofrecida por la página oficial de MySQL para fines educativos.

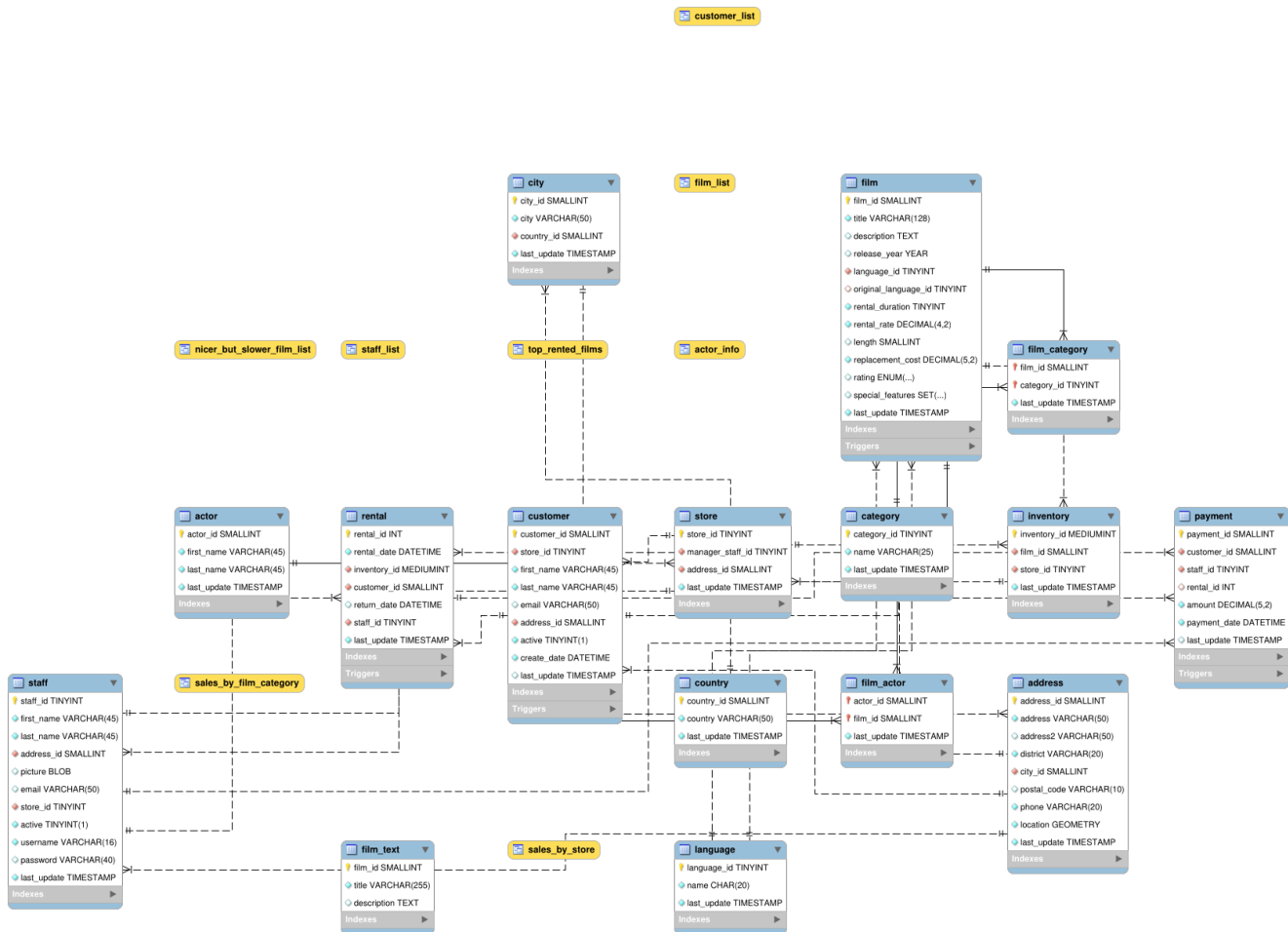


Figure 1: Base datos sakila.

Ahora, se mostrará una consulta que devuelve el título de la película, el número total de alquileres, la duración promedio de la renta o alquiler de la película, y los ingresos totales generados por la película. Los resultados están ordenados por ingresos totales en orden descendente y se limitan a las 10 películas con mayores ingresos.

```
SELECT
    film.title AS film_title,
    COUNT(*) AS rental_count,
```

```

    AVG(DATEDIFF(rental.return_date, rental.rental_date)) AS avg_rental_duration,
    SUM(payment.amount) AS total_revenue
FROM
    rental
JOIN
    inventory ON rental.inventory_id = inventory.inventory_id
JOIN
    film ON inventory.film_id = film.film_id
JOIN
    payment ON rental.rental_id = payment.rental_id
GROUP BY
    film.film_id
ORDER BY
    total_revenue DESC
LIMIT 10;

```

A simple vista puede verse muy complejo, sin embargo, es mucho más sencillo de lo que parece. Si intentamos traducir un poco más literal lo que la consulta intenta extraer sería algo como:

```

SELECCIONAR
    pelicula.titulo COMO titulo_pelicula,
    CONTAR(*) COMO cantidad_alquileres,
    PROM(DATEDIFF(alquiler.fecha_devolucion, alquiler.fecha_alquiler)) COMO
        duracion_promedio_alquiler,
    SUM(pago.monto) COMO ingreso_total
DESDE
    alquiler
UNIR
    inventario EN alquiler.id_inventario = inventario.id_inventario
UNIR
    pelicula EN inventario.id_pelicula = pelicula.id_pelicula
UNIR
    pago EN alquiler.id_alquiler = pago.id_alquiler
AGRUPAR POR
    pelicula.id_pelicula
ORDENAR POR
    ingreso_total DESC
LIMITAR 10;

```

Para no dejar cabos sueltos, podemos observar el resultado de la consulta a continuación:

film_title	rental_count	avg_rental_duration	total_revenue
TELEGRAPH VOYAGE	27	6.4815	231.73
WIFE TURN	31	4.8387	223.69
ZORRO ARK	31	4.5161	214.69
GOODFELLAS SALUTE	31	5.2581	209.69
SATURDAY LAMBS	28	5.1786	204.72
TITANS JERK	29	5.5714	201.71
TORQUE BOUND	27	5.0000	198.72
HARRY IDAHO	30	5.5172	195.70
INNOCENT USUAL	26	4.9615	191.74
HUSTLER PARTY	22	6.5000	190.78

4 Programación Declarativa con Python

La programación declarativa es un paradigma de programación donde el código describe qué debe hacerse en lugar de cómo hacerlo. En este enfoque, el programador declara o especifica el resultado deseado y confía en que el sistema encuentre la mejor manera de lograrlo (librerías, frameworks, plugins, etc.).

Sin embargo, para entender un poco mejor el enfoque de la Programación Declarativa, es necesario contrastar con la Programación Imperativa.

4.1 Programación Declarativa.

En la programación funcional, se **enfatisa** el uso de funciones puras y evita los efectos secundarios.

```
# Ejemplo de map y lambda
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x ** 2, numbers))
print(squared) # Output: [1, 4, 9, 16, 25]
```

4.2 Expresiones Generadoras.

Las expresiones generadoras permiten construir secuencias de manera perezosa (lazy) utilizando la sintaxis de compresión de listas.

```
# Ejemplo de map y lambda
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x ** 2, numbers))
print(squared) # Output: [1, 4, 9, 16, 25]
```

La expresión `(x ** 2 for x in numbers)` crea un generador que produce los cuadrados de los números en la lista `numbers`, cuando se solicita.

4.3 SQL con SQLAlchemy

Gracias al alto nivel declarativo de SQLAlchemy, podemos construir instrucciones en Python muy similares a las instrucciones SQL.

```
from sqlalchemy import create_engine, Column, Integer, String
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

# Crear una clase que representa una tabla
Base = declarative_base()

class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    name = Column(String)
    age = Column(Integer)

# Crear una instancia de motor y conectar a la base de datos
engine = create_engine('mysql://usuario:contraseña@localhost/nombre_base_de_datos',
    echo=True)

Base.metadata.create_all(engine)

# Crear una sesión para interactuar con la base de datos
Session = sessionmaker(bind=engine)
session = Session()

# Insertar un nuevo usuario
new_user = User(name='John', age=30)
session.add(new_user)
```

```
session.commit()

# Consultar usuarios
users = session.query(User).all()
for user in users:
    print(user.name, user.age)
```
