

# Práctica 3 Sistemas Informáticos

Ignacio Serena y Marcos Muñoz

## 1. NoSQL

### 1.1 Base de datos documental

#### Apartado A

En este apartado únicamente nos vamos a encargar de iniciar el contenedor y la base de datos. Tras esto, nos conectaremos a la base de datos PostgreSQL y comprobaremos que está dotada de las tablas necesarias para realizar la carga a MongoDB están presentes. Esta comprobación la haremos con los siguientes comandos:

```
sudo docker-compose up
psql -h localhost -U alumnodb -d si1
\dt
```

#### Apartado B

En este apartado vamos a realizar un script para automatizar la creación de la Base de Datos documental. Este script extraerá la información de la BD PostgreSQL y se encargará de crear otra BD en MongoDB llamada si1, que contendrá la colección llamada france donde se almacenarán los documentos. EL script implementado es el siguiente:

```
create_mongodb_from_postgresqldb.py
```

```
import pymongo
from sqlalchemy import create_engine, text

# Configuración para PostgreSQL
```

```

POSTGRES_URL = "postgresql://alumnodb:1234@localhost:5432/s

# Configuración para MongoDB
MONGO_URL = "mongodb://localhost:27017/"
DB_NAME = "si1"
COLLECTION_NAME = "france"

# Parte 1: Creamos la base de datos y la colección en Mongo
def create_mongo():
    # Conexión a MongoDB
    mongo_client = pymongo.MongoClient(MONGO_URL)
    mongo_db = mongo_client[DB_NAME]

    # Crea la colección "France"
    try:
        mongo_db.create_collection(COLLECTION_NAME)
    except pymongo.errors.CollectionInvalid:
        print(f"La colección {COLLECTION_NAME} ya existe.")

# Define el esquema JSON para la validación de los documentos
vexpr = {
    "$jsonSchema": {
        "required": ["title", "year", "genres", "directors"],
        "properties": {
            "title": {"bsonType": "string"},
            "year": {"bsonType": "int"},
            "genres": {"bsonType": "array", "items": {"bsonType": "string"}},
            "directors": {"bsonType": "array", "items": {"bsonType": "string"}},
            "actors": {"bsonType": "array", "items": {"bsonType": "string"}},
            "most_related_movies": {
                "bsonType": "array",
                "items": {
                    "bsonType": "object",
                    "properties": {
                        "title": {"bsonType": "string"},
                        "year": {"bsonType": "int"}
                    }
                }
            }
        }
    }
}

```

```

        },
        "related_movies": {          # Lista de películas
            "bsonType": "array",
            "items": {
                "bsonType": "object",
                "properties": {
                    "title": {"bsonType": "string"},
                    "year": {"bsonType": "int"}
                }
            }
        }
    }
}

```

# Comando para aplicar el esquema de validación

```

cmd = {
    "collMod": COLLECTION_NAME,
    "validator": vexpr,
    "validationLevel": "moderate"
}

```

```

mongo_db.command(cmd)    # Aplica el esquema de validación
print("Validador aplicado correctamente.")

```

# Parte 2: Extraemos datos de PostgreSQL y cargarlos en MongoDB

def load\_mongo():

```

    # Conexión a PostgreSQL
    engine = create_engine(POSTGRES_URL)

```

```

    # Conexión a MongoDB

```

```

    mongo_client = pymongo.MongoClient(MONGO_URL)
    mongo_db = mongo_client[DB_NAME]
    collection = mongo_db[COLLECTION_NAME]

```

```

    # Consulta para obtener las películas francesas

```

```

    query = text("""
        SELECT p.movieid, p.movietitle, p.year, mg.genre, c
        FROM imdb_movies p

```

```

        JOIN imdb_moviecountries mc ON p.movieid = mc.movieid
        JOIN imdb_moviegenres mg ON p.movieid = mg.movieid
        LEFT JOIN imdb_directormovies dm ON p.movieid = dm.movieid
        LEFT JOIN imdb_directors d ON dm.directorid = d.directorid
        LEFT JOIN imdb_actormovies am ON p.movieid = am.movieid
        LEFT JOIN imdb_actors a ON am.actorid = a.actorid
        WHERE mc.country = 'France';
    """
)

```

```

# Extrae y transforma los datos

```

```

with engine.connect() as conn:

```

```

    result = conn.execute(query) # Ejecuta la consulta
    movies = {} # Diccionario para almacenar datos traídos

```

```

# Itera sobre los resultados asegurándote de que se procesen todos

```

```

for row in result.mappings():

```

```

    movie_id = row["movieid"]

```

```

    if movie_id not in movies:

```

```

        # El título está en el formato "Nombre de película (año)"
        title_with_date = row["movietitle"]

```

```

        # Separar por '(' para aislar la fecha
        parts = title_with_date.split('(')

```

```

        # El título limpio es la primera parte antes de la fecha
        clean_title = parts[0].strip()

```

```

        # Convertimos el año a un número entero
        try:

```

```

            movie_year = int(row["year"]) # Convertir a entero
        except ValueError:

```

```

            print(f"Error: año faltante o no válido")
            continue # Si no se puede convertir, saltar

```

```

        # Nos aseguramos de que el año sea válido

```

```

        if movie_year <= 0:

```

```

            print(f"Error: año faltante o no válido")

```

```

        continue # Si el año es inválido, saltar

    movies[movie_id] = {
        "title": clean_title,
        "year": movie_year,
        "genres": set(),
        "directors": set(),
        "actors": set(),
        "most_related_movies": [],
        "related_movies": []
    }
    movies[movie_id]["genres"].add(row["genre"] or "")
    if row["director"]:
        movies[movie_id]["directors"].add(row["director"])
    if row["actor"]:
        movies[movie_id]["actors"].add(row["actor"])

# Inserta películas
for movie in movies.values():
    movie["genres"] = list(movie["genres"])
    movie["directors"] = list(movie["directors"])
    movie["actors"] = list(movie["actors"])

    try:
        collection.insert_one(movie)
    except pymongo.errors.WriteError as e:
        print(f"Error al insertar documento: {movie}")
        print(f"Detalle del error: {e}")

print("Películas cargadas en MongoDB.")

# Parte 3: Detección de películas relacionadas
def calculate_related_movies():

```

```

# Conexión a MongoDB
mongo_client = pymongo.MongoClient(MONGO_URL)
mongo_db = mongo_client[DB_NAME]
collection = mongo_db[COLLECTION_NAME]

# Obtenemos todas las películas
all_movies = list(collection.find())

# Procesamos cada película
for movie in all_movies:
    title = movie["title"]
    genres_set = set(movie["genres"])
    most_related = []
    related = []

    # Compara con todas las demás películas
    for other_movie in all_movies:
        if movie["_id"] == other_movie["_id"]: # No co
            continue

        #Extraemos los géneros de la película que vamos
        other_genres_set = set(other_movie["genres"])

        # Calculamos la compatibilidad de géneros
        genre_overlap = len(genres_set & other_genres_s

        if genre_overlap == 1.0: # 100% coincidencia
            most_related.append({"title": other_movie["
        elif genre_overlap >= 0.5: # 50% o más coincid
            related.append({"title": other_movie["title

# Actualizamos las películas relacionadas en la bas
collection.update_one(
    {"_id": movie["_id"]},
    {"$set": {
        "most_related_movies": most_related[:10],
        "related_movies": related[:10] # Máximo 10
    }}

```

```

    )
    print(f"Películas relacionadas calculadas para '{ti

# Ejecución principal
if __name__ == "__main__":
    create_mongo()
    load_mongo()
    calculate_related_movies()

```

Este script que hemos implementado, realiza las siguientes tareas:

1. Se conecta a PostgreSQL y extrae los datos.
2. Crea la base de datos `si1` y la colección `france` en MongoDB.
3. Transforma los datos de PostgreSQL a la estructura de MongoDB.
4. Inserta los documentos en MongoDB.

Este archivo tiene dos funciones principales:

1. `create_mongo()` : Define la base de datos y la colección con validadores en MongoDB.
2. `load_mongo()` : Extrae datos de PostgreSQL, transforma los datos y los carga en MongoDB.

## ***Pruebas de validación***

Para comprobar este apartado hemos ejecutado el script y, posteriormente hemos accedido al cliente de MongoDB desde la terminal. Una vez dentro ejecutamos el siguiente comando:

```

show collections
db.france.find().limit(5).pretty()

```

```

si1> db.france.find().limit(5).pretty()
[
  {
    _id: ObjectId('674452a2f49842c64806a011'),
    title: '1871',
    year: 1990,
    genres: [ 'History', 'Drama' ],
    directors: [ 'McMullen, Ken' ],
    actors: [
      'Hondo, Med',          'Spall, Timothy',
      'Klaff, Jack',         'Lynch, John (I)',
      'Toscano, Maria João', 'de Sousa, Alexandre',
      'Dankworth, Jacqui',   'César, Carlos (I)',
      'de Medeiros, Maria',  'Braine, Alan',
      'Padrão, Ana',         'McNeice, Ian',
      'Maia, André (I)',     'Ruivo, João Pedro',
      'Pinon, Dominique',    'Shaw, Bill (I)',
      'Argüelles, José',     'Seth, Roshan',
      'Michaels, Cedric'
    ],
    most_related_movies: [
      { title: 'Colonel Chabert, Le', year: 1994 },
      { title: 'Hour of the Pig, The', year: 1993 },
      { title: 'JFK', year: 1991 },
      { title: 'Jing ke ci qin wang', year: 1999 },
      { title: 'Last September, The', year: 1999 },
      { title: 'Sacco e Vanzetti', year: 1971 },
      { title: '1871', year: 1990 },
      { title: 'Colonel Chabert, Le', year: 1994 },
      { title: 'Hour of the Pig, The', year: 1993 },
      { title: 'JFK', year: 1991 }
    ],
    related_movies: [
      { title: '2 ou 3 choses que je sais d'elle", year: 1967 },
      { title: 'Abre los ojos', year: 1997 },
      { title: 'Ai no corrida', year: 1976 },
      { title: 'Albino Alligator', year: 1996 },
      { title: 'Alice et Martin', year: 1998 },
      {
        title: 'Alphaville, une étrange aventure de Lemmy Caution',
        year: 1965
      },
      { title: 'Amantes del Círculo Polar, Los', year: 1998 },
      { title: 'Amants criminels, Les', year: 1999 },
      { title: 'Amants du Pont-Neuf, Les', year: 1991 },
      { title: 'Amateur', year: 1994 }
    ]
  },
  {
    _id: ObjectId('674452a2f49842c64806a012'),
    title: "2 ou 3 choses que je sais d'elle",
    year: 1967
  }
]

```

Como podemos observar, los datos se han cargado correctamente en la base de datos documental, y cumplen todas las especificaciones pedidas en el enunciado.

## Apartado C

En este apartado se nos solicita realizar otro script en python que muestre los resultados de unas queries propuestas por el profesorado. Estas queries son las siguientes:



a. Películas de ciencia ficción entre 1994 y 1998

```
def query_sci_fi_movies(collection):
    query = {
        "genres": "Sci-Fi", # Género: ciencia ficción
        "year": {"$gte": 1994, "$lte": 1998} # Años entre 1994 y 1998
    }
    results = list(collection.find(query))
    return pd.DataFrame(results)
```

b. Películas de drama del año 1998 que empiecen por "The"

```
def query_drama_the_movies(collection):
    query = {
        "genres": "Drama", # Género: drama
        "year": 1998, # Año 1998
        "title": {"$regex": r"^(The |, The$)"} # Título empieza con "The" o termina con ", The"
    }
    results = list(collection.find(query))
    return pd.DataFrame(results)
```

c. Películas en las que Faye Dunaway y Viggo Mortensen hayan compartido reparto

```
def query_shared_cast_movies(collection):
    query = {
        "actors": {"$all": ["Dunaway, Faye", "Mortensen, Viggo"]}
    }
    results = list(collection.find(query))
    return pd.DataFrame(results)
```

La salida de ejecutar el script es la siguiente:

```

marcos1701@marcos1701-VirtualBox:~/Escritorio/Marcos/M/SI/SI/P3/app-mongodb-et1$ python3 mongodb_querues.py

Consulta (a): Peliculas de ciencia ficción entre 1994 y 1998
  _id                                     title ...                               most_related_movles                               related_movles
0  674595524a660acea4d34f68             Abre los ojos ... [[{'title': 'Alphaville, une étrange aventure d... [[{'title': 'Basic Instinct', 'year': 1992}, {'t...
1  674595524a660acea4d34fc7             Fifth Element, The ... [[{'title': 'Moonraker', 'year': 1979}, {'title... [[{'title': 'Abre los ojos', 'year': 1997}, {'t...
2  674595524a660acea4d34fe4             Highlander III: The Sorcerer ... [[{'title': 'Stargate', 'year': 1994}] [[{'title': 'Barbarella', 'year': 1968}, {'titl...
3  674595524a660acea4d35031             Nowhere ... [[{'title': 'Abre los ojos', 'year': 1997}, {'t... [[{'title': '1871', 'year': 1990}, {'title': '2...
4  674595524a660acea4d3506e             Stargate ... [] [[{'title': 'Adventures of Pinocchio, The', 'ye...

[5 rows x 8 columns]

Consulta (b): Peliculas de drama del año 1998 que comienzan con 'The'
  _id                                     title ...                               most_related_movles                               related_movles
0  674595524a660acea4d35001             Land Girls, The ... [[{'title': '1871', 'year': 1990}, {'title': '2... []
1  674595524a660acea4d35015             Man with Rain in His Shoes, The ... [[{'title': 'Collectionneuse, La', 'year': 1967... [[{'title': 'Abre los ojos', 'year': 1997}, {'t...
2  674595524a660acea4d3504a             Quarry, The ... [[{'title': '1871', 'year': 1990}, {'title': '2... []

[3 rows x 8 columns]

Consulta (c): Peliculas en las que Faye Dunaway y Viggo Mortensen compartieron reparto
  _id                                     title ...                               most_related_movles                               related_movles
0  674595524a660acea4d34f6b             Albino Alligator ... [[{'title': 'Amants criminels, Les', 'year': 19... [[{'title': 'Abre los ojos', 'year': 1997}, {'t...

[1 rows x 8 columns]

```

Como podemos observar los resultados son los esperados en las 3 consultas, lo que sugiere que su implementación es correcta.

## 1.1 BBDD basadas en Grafos

En este apartado, desarrollamos una solución práctica para implementar una Base de Datos basada en grafos a partir de una base relacional de PostgreSQL. El objetivo principal fue migrar y transformar los datos para Neo4j, una plataforma de grafos que permite analizar relaciones complejas de forma más eficiente y visual. A continuación, describiremos los pasos realizados y los resultados obtenidos.

### Tareas Realizadas

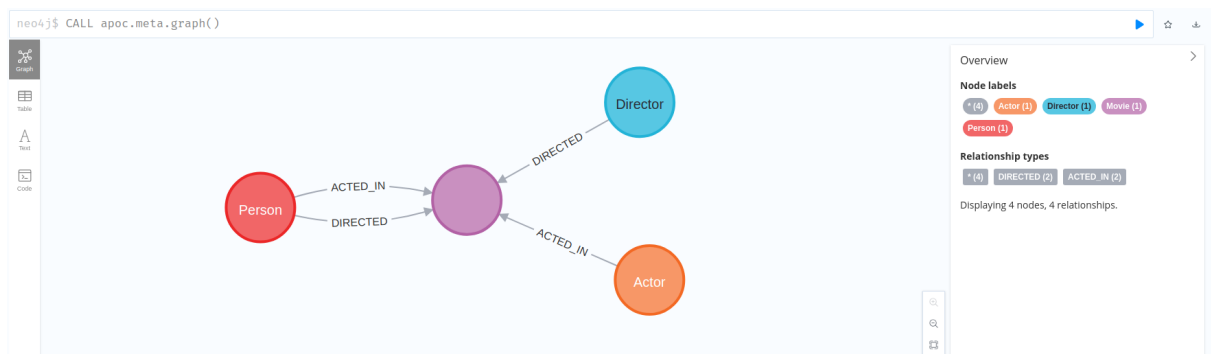
#### A. Creación de la Base de Datos basada en grafos

Diseñamos un grafo que representara las relaciones entre las películas, sus actores y directores, tomando como base las 20 películas estadounidenses más vendidas de la base de datos en PostgreSQL. La estructura del grafo incluye:

- **Nodos:**
  - Películas.
  - Actores.
  - Directores.
  - Personas.
- **Relaciones:**

- Los actores actúan en películas ( `ACTS_IN` ).
- Los directores dirigen películas ( `DIRECTED` ).
- Las personas pueden desempeñar un de las funciones previas.

Este modelo relacional se transformó a un modelo de grafos para Neo4j, asegurando que al ejecutar el procedimiento `apoc.meta.graph` en Neo4j, la estructura del grafo reflejara la relación deseada, como podemos observar en el resultado de ejecución:



## B. Implementación del Script

Para automatizar el proceso de creación de la Base de Datos en Neo4j, hemos desarrollado un script en Python llamado `create_neo4jdb_from_postgreslodb.py`. Este script se encuentra en la carpeta `app-neo4j-etl` y utiliza las siguientes tecnologías:

- **SQLAlchemy:** Para conectarse a la base de datos PostgreSQL, realizar consultas y extraer datos.
- **Librería Neo4j:** Para conectar y cargar los datos en Neo4j.
- **Transformaciones:**
  - Consolidación de géneros, actores y directores en listas para reducir redundancias.
  - Optimización de consultas SQL mediante funciones como `ARRAY_AGG` para agrupar información relacionada.

El script realiza las siguientes operaciones:

1. Se conecta a PostgreSQL y selecciona las 20 películas con mayor promedio de calificaciones ( `ratingmean` ).
2. Transforma los datos en una estructura adecuada para grafos.

3. Carga los datos en Neo4j, creando nodos y relaciones según el modelo diseñado.
4. Configura la base de datos en Neo4j con las credenciales proporcionadas ( `si1` como nombre de la base de datos y `si1-password` como contraseña).

A continuación proporcionamos el código implementado para la automatización de la migración:

```
"""Module to transform the movie database from relational to graph database
import time

from neo4j import GraphDatabase
from consultas import SQL

class EtlFromPostgresToNeo4j:
    """Class to Transform Movie relational data to graph database

    def __init__(self, uri, user, password):
        self.driver = GraphDatabase.driver(uri, auth=(user, password))
        self.delete_all_nodes()
        self.create_all_constraints()
        self.sql = SQL()

    def transform_postgres_to_neo4j(self):
        """
        Transform Movie Relational database to Graph database
        """
        # Get all the data from PostgreSQL
        # get the 20 most selling movies from USA
        db_result = self.best_selling_movies_usa()
        print("Processing DB rows ...")
        start_time = time.time()

        for row_movie in db_result:
            dict_movie = row_movie
            node_movie = self.create_and_return_movie(dict_movie)
```

```

        if not node_movie:
            print(f"[ERROR] No se pudo crear el nodo para
                continue

        self.insert_all_actors(dict_movie, node_movie)
        self.insert_all_directors(dict_movie, node_movie)

    print("End ETL process...")
    f_string_order = f"Process finished --- {time.time()}"
    print(f_string_order)

def insert_all_actors(self, dict_movie, node_movie):
    """Inserta todos los actores que participaron en la p
    # TO-DO SQL
    db_actors = []
    query = """
        SELECT a.actorid, a.actorname
        FROM imdb_actors a
        JOIN imdb_actormovies am ON a.actorid = am.actorid
        WHERE am.movieid = :movieid
    """
    db_actors = self.sql.execute_query(query, {"movieid":

    # Insert all actors
    for row_actor in db_actors:
        dict_actor = row_actor
        self.create_and_return_actor(dict_actor, node_mov

def insert_all_directors(self, dict_movie, node_movie):
    """Insert the directors that directed the movie"""
    # TO-DO SQL
    db_directors = []
    query = """
        SELECT d.directorid, d.directorname
        FROM imdb_directors d
        JOIN imdb_directormovies dm ON d.directorid = dm.
        WHERE dm.movieid = :movieid
    """

```

```

        db_directors = self.sql.execute_query(query, {"moviei

# Insert all directors
for row_director in db_directors:
    dict_director = row_director
    self.create_and_return_director(dict_director, no

def create_and_return_director(self, dict_director, node_)
    """ Create Node Person Director"""
    with self.driver.session() as session:
        # Creamos nodo Person
        person_node = session.execute_write(self._create_
        if not person_node:
            print("[ERROR] No se pudo crear el nodo Perso
            return None

        # Creamos nodo Director
        node_director = session.execute_write(self._creat
        if not node_director:
            print("[ERROR] No se pudo crear el nodo Direc
            return None

        # Relacionamos Person con Movie
        if person_node and node_director:
            session.execute_write(self._create_person_dir

        #Create Relationship DIRECTED
            session.execute_write(self._create_and_return

        return node_director

def create_and_return_actor(self, dict_actor, node_movie)
    with self.driver.session() as session:
        # Creamos nodo Person
        person_node = session.execute_write(self._create_
        if not person_node:
            print("[ERROR] No se pudo crear el nodo Perso

```

```

        return None

    # Creamos nodo Actor
    node_actor = session.execute_write(self._create_a

    if not node_actor:
        print("[ERROR] No se pudo crear el nodo Actor")
        return None

    # Relacionar Person con Actor
    if person_node and node_actor:
        session.execute_write(self._create_person_act

    # Creamos relación ACTED_IN solo si el nodo_movie
    if node_movie:
        session.execute_write(self._create_and_return
    return node_actor

def create_and_return_movie(self, dict_movie: dict):
    """ Create Node Movie """
    with self.driver.session() as session:
        movie = session.execute_write(self._create_and_re
    return movie

def delete_all_nodes(self):
    """Delete all nodes from database"""
    with self.driver.session() as session:
        session.execute_write(self._delete_all_nodes)

def create_all_constraints(self):
    """ Create Genre Node """
    with self.driver.session() as session:
        # Creación de constraints UNIQUE
        session.run(
            "CREATE CONSTRAINT actor_id_unique IF NOT EXI
        )

```

```

        session.run(
            "CREATE CONSTRAINT director_id_unique IF NOT EXISTS"
        )
        session.run(
            "CREATE CONSTRAINT movie_id_unique IF NOT EXISTS"
        )

# Creación de índices
        session.run(
            "CREATE INDEX actor_id IF NOT EXISTS FOR (a:Actor)"
        )
        session.run(
            "CREATE INDEX director_id IF NOT EXISTS FOR (d:Director)"
        )
        session.run(
            "CREATE INDEX movie_id IF NOT EXISTS FOR (m:Movie)"
        )

    @staticmethod
    def _create_and_return_directed(tx, node_movie, node_director):
        if not node_movie or not node_director:
            print("[ERROR] node_movie o node_director es None")
            return None

        # Accedemos directamente a las propiedades del nodo
        movieid = node_movie["movieid"]
        directorid = node_director["directorid"]

        if not movieid or not directorid:
            print(f"[ERROR] Faltan datos en node_movie o node_director")
            return None

        query = """
            MATCH (d:Director {directorid: $directorid}), (m:Movie {movieid: $movieid})
            MERGE (d)-[:DIRECTED]->(m)
            RETURN d, m
        """
        result = tx.run(query, directorid=directorid, movieid=movieid)

```



```

        return result.single()

    @staticmethod
    def _create_and_return_acted_in(tx, node_movie, node_actor):
        if not node_movie or not node_actor:
            print("[ERROR] node_movie o node_actor es None.")
            return None

        # Accede directamente a las propiedades del nodo
        movieid = node_movie.get("movieid")
        actorid = node_actor.get("actorid")

        if not movieid or not actorid:
            print(f"[ERROR] Faltan datos en node_movie o node_actor")
            return None

        query = """
            MATCH (a:Actor {actorid: $actorid}), (m:Movie {movieid: $movieid})
            MERGE (a)-[:ACTED_IN]->(m)
            RETURN a, m
        """

        result = tx.run(query, actorid=actorid, movieid=movieid)
        return result.single()

    @staticmethod
    def _create_person_actor_relationship(tx, node_movie, node_actor):
        """Create ACTED_IN relationship between Person and Movie"""
        if not node_movie or not node_actor:
            print("[ERROR] node_movie o node_actor es None.")
            return None

        # Accede directamente a las propiedades del nodo
        movieid = node_movie["movieid"]
        person_id = node_actor["actorid"]

        if not movieid or not person_id:

```

```

        print(f"[ERROR] Faltan datos en node_movie o pers
        return None

query = """
    MATCH (p:Person {personid: $personid}), (m:Movie
    MERGE (p)-[:ACTED_IN]->(m)
    """
result = tx.run(query, personid=person_id, movieid=mo
return result.single()

@staticmethod
def _create_person_director_relationship(tx, node_movie,
    """Create DIRECTED relationship between Person and Mo
    if not node_movie or not node_director:
        print("[ERROR] node_movie o node_director es None
        return None

    # Accede directamente a las propiedades del nodo
    movieid = node_movie["movieid"]
    person_id = node_director["directorid"]

    if not movieid or not person_id:
        print(f"[ERROR] Faltan datos en node_movie o pers
        return None

    query = """
        MATCH (p:Person {personid: $personid}), (m:Movie
        MERGE (p)-[:DIRECTED]->(m)
        """
    result = tx.run(query, personid=person_id, movieid=mo
    return result.single()

```

```

@staticmethod
def _create_and_return_actor(tx, dict_actor):
    query = """
        MERGE (a:Actor {actorid: $actorid})
        SET a.name = $name
        RETURN a
    """
    result = tx.run(query, actorid=dict_actor["actorid"],

    # Obtener el primer resultado (registro)
    record = result.single() # Devuelve un solo registro
    if record:
        return record["a"] # Devuelve el nodo
    else:
        print(f"[ERROR] No se pudo crear o encontrar el n
        return None

```

```

@staticmethod
def _create_and_return_director(tx, dict_director):

    # Verificar que las claves existen
    if "directorid" not in dict_director or "directorname"
        raise KeyError("Faltan claves necesarias en dict_

    # Extraemos los valores
    directorid = dict_director["directorid"]
    name = dict_director["directorname"]

    # Consulta Cypher para crear o encontrar al director
    query = """
        MERGE (d:Director {directorid: $directorid})
        SET d.name = $name
        RETURN d
    """

    # Ejecutar la consulta

```

```

        result = tx.run(query, directorid=directorid, name=name)
        # Devuelve directamente el nodo en lugar del Record
        record = result.single()
        if record:
            return record["d"] # 'd' es el alias del nodo en
        else:
            print(f"[ERROR] No se pudo crear o encontrar el n
            return None

    @staticmethod
    def _create_and_return_movie(tx, dict_movie):
        """Crea un nodo Movie"""
        # Consulta para crear o encontrar una película
        query = """
            MERGE (m:Movie {movieid: $movieid})
            SET m.title = $title, m.year = $year, m.genre = $
            RETURN m
        """

        result = tx.run(query,
                        movieid=dict_movie["movieid"],
                        title=dict_movie["movietitle"],
                        year=dict_movie["year"],
                        genre=dict_movie.get("genre"))

        # Obtener el primer resultado (registro)
        record = result.single() # Devuelve un solo registro
        if record:
            return record["m"] # Devuelve el nodo
        else:
            print(f"[ERROR] No se pudo crear o encontrar el n
            return None

    @staticmethod
    def _create_and_return_person(tx, person_id, name):

```

```

        """Create or return a Person node"""
        query = """
            MERGE (p:Person {personid: $personid})
            SET p.name = $name
            RETURN p
        """
        result = tx.run(query, personid=person_id, name=name)
        record = result.single()
        if record:
            return record["p"]
        else:
            print(f"[ERROR] No se pudo crear o encontrar el n
            return None

    @staticmethod
    def _delete_all_nodes(tx):
        tx.run("MATCH (n) DETACH DELETE n")

    def best_selling_movies_usa(self):
        """Method that execute the query on the database to g
        # TO-DO SQL
        query = """
            SELECT DISTINCT
                m.movieid,
                m.movietitle,
                m.year,
                c.country,
                SUM(i.sales) AS total_sales
            FROM imdb_movies m
            JOIN imdb_moviecountries c ON m.movieid = c.movie
            JOIN products p ON m.movieid = p.movieid
            JOIN inventory i ON p.prod_id = i.prod_id
            WHERE c.country = 'USA' -- Filtrar películas de l
            GROUP BY m.movieid, m.movietitle, m.year, c.coun
            ORDER BY total_sales DESC -- Ordenar por ventas t
            LIMIT 20; -- Seleccionar las 20 películas más ven

```

```

        """
        result = self.sql.execute_query(query)
        return result

    def close(self):
        """
        Close driver connection
        """
        self.driver.close()

if __name__ == "__main__":
    # Neoj Configuration
    URI = "bolt://127.0.0.1:7687"
    USER = "neo4j"
    PASSWORD = "si1-password"
    convert = EtlFromPostgresToNeo4j(URI, USER, PASSWORD)
    try:
        convert.transform_postgres_to_neo4j()
    finally:
        convert.close()

```

Para comprobar si el proceso de migración ha sido exitoso, es importante realizar una serie de **pruebas y validaciones** sobre los datos migrados en Neo4j. Para ello, realizaremos las siguientes pruebas.

## ***Validación de creación de nodos***

- **Prueba de películas:**

```

MATCH (m:Movie)
RETURN m.movieid, m.title, m.year
LIMIT 10;

```

Salida:

neo4j\$ MATCH (m:Movie) RETURN m.movieid, m.title, m.year LIMIT 10;

	m.movieid	m.title	m.year
1	229764	"Life Less Ordinary, A (1997)"	"1997"
2	291835	"Only You (1994)"	"1994"
3	156403	"Glimmer Man, The (1996)"	"1996"
4	189256	"Illtown (1996)"	"1996"
5	425473	"Very Thought of You, The (1945)"	"1945"
6	49493	"Blob, The (1958)"	"1958"
7			

- **Prueba de actores:**

```
MATCH (a:Actor)
RETURN a.actorid, a.name
LIMIT 10;
```

Salida:

neo4j\$ MATCH (a:Actor) RETURN a.actorid, a.name LIMIT 10;

	a.actorid	a.name
1	96754	"Gorham, Christopher"
2	98198	"Gowdy, Chuck"
3	127665	"Hedaya, Dan"
4	140937	"Holm, Ian"
5	261467	"Chaykin, Maury"
6	324886	"Kanig, Frank"
7		

Started streaming 10 records after 14 ms and completed after 15 ms.

- **Prueba de directores:**

```
MATCH (d:Director)
RETURN d.directorid, d.name
LIMIT 10;
```

Salida:

```
neo4j$ MATCH (d:Director) RETURN d.directorid, d.name LIMIT 10;
```

	d.directorid	d.name
1	12079	"Boyle, Danny"
2	50473	"Jewison, Norman"
3	40196	"Gray, John (I)"
4	38986	"Gomez, Nick (I)"
5	10199	"Blake, Ben K."
6	27810	"Doughten Jr., Russell S."
7		

Started streaming 10 records after 7 ms and completed after 7 ms.

Como podemos observar, todos los nodos han sido creados perfectamente.

## Validación de relaciones entre nodos

- Películas con actores:

```
MATCH (a:Actor)-[:ACTED_IN]->(m:Movie)
RETURN a.name, m.title
LIMIT 10;
```

Salida:

```
neo4j$ MATCH (a:Actor)-[:ACTED_IN]->(m:Movie) RETURN a.name, m.title LIMIT 10;
```

	a.name	m.title
1	"Schaub, Mary-Cristina"	"Life Less Ordinary, A (1997)"
2	"Martinez, Crystal (I)"	"Life Less Ordinary, A (1997)"
3	"Gorham, Christopher"	"Life Less Ordinary, A (1997)"
4	"Gowdy, Chuck"	"Life Less Ordinary, A (1997)"
5	"Hedaya, Dan"	"Life Less Ordinary, A (1997)"
6	"Holm, Ian"	"Life Less Ordinary, A (1997)"
7		

Started streaming 10 records in less than 1 ms and completed after 1 ms.

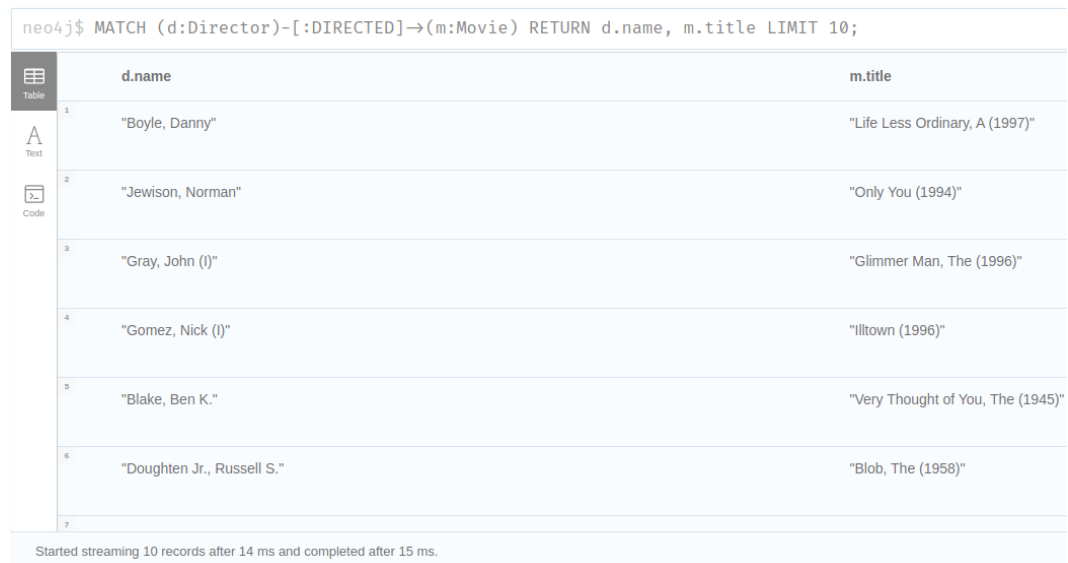


Podemos confirmar que cada actor esté relacionado con las películas correspondientes.

- **Películas con directores:**

```
MATCH (d:Director)-[:DIRECTED]->(m:Movie)
RETURN d.name, m.title
LIMIT 10;
```

Salida:



The screenshot shows the Neo4j query interface. The query entered is: `neo4j$ MATCH (d:Director)-[:DIRECTED]->(m:Movie) RETURN d.name, m.title LIMIT 10;`. The results are displayed in a table with two columns: `d.name` and `m.title`. The table contains 6 rows of data. On the left side of the interface, there are icons for 'Table', 'Text', and 'Code', with 'Table' being the active view. At the bottom, a status message reads: 'Started streaming 10 records after 14 ms and completed after 15 ms.'

	d.name	m.title
1	"Boyle, Danny"	"Life Less Ordinary, A (1997)"
2	"Jewison, Norman"	"Only You (1994)"
3	"Gray, John (I)"	"Glimmer Man, The (1996)"
4	"Gomez, Nick (I)"	"Illtown (1996)"
5	"Blake, Ben K."	"Very Thought of You, The (1945)"
6	"Doughten Jr., Russell S."	"Blob, The (1958)"
7		

Comprobamos que los directores están asociados correctamente con las películas.


## Verificación de la integridad de datos


- **Actores sin relaciones:**

```
MATCH (a:Actor)
WHERE NOT (a)-[:ACTED_IN]->( )
RETURN a.name;
```

Salida:

```
neo4j$ MATCH (a:Actor) WHERE NOT (a)-[:ACTED_IN]→() RETURN a.name;
```


  
Table


  
Code

(no changes, no records)


Completed in less than 1 ms.


- **Directores sin relaciones:**

```
MATCH (d:Director)
WHERE NOT (d)-[:DIRECTED]->()
RETURN d.name;
```

Salida:

```
neo4j$ MATCH (d:Director) WHERE NOT (d)-[:DIRECTED]→() RETURN d.name;
```


  
Table


  
Code

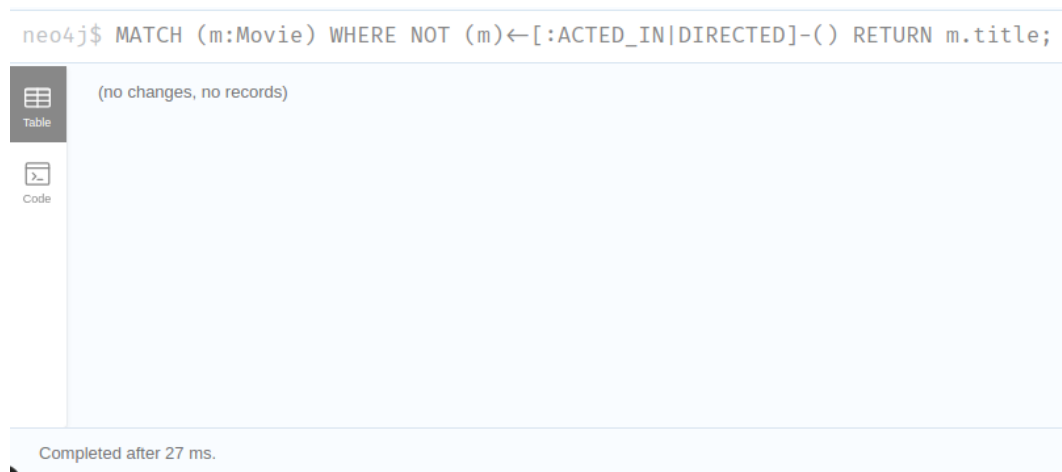
(no changes, no records)

Completed after 1 ms.

- **Películas sin relaciones:**

```
MATCH (m:Movie)
WHERE NOT (m)-[:ACTED_IN|DIRECTED]-()
RETURN m.title;
```

Salida:



Como hemos podido comprobar todas las consultas devuelven un conjunto vacío, ya que todos los nodos deben estar vinculados a alguna película. Con todas estas comprobaciones, podemos concluir en que la migración de la base de datos ha sido exitosa.

## C. Consultas sobre la Base de Datos en Neo4j

Para este último apartado diseñamos y ejecutamos consultas avanzadas en Cypher para explorar las relaciones en el grafo. Las consultas incluyen:

### 1. Actores que no han trabajado con "Winston, Hattie" pero tienen un tercero en común:

- Devuelve una lista ordenada alfabéticamente de 10 actores que cumplen esta condición.
- Script: `winston-hattie-co-co-actors.cypher`.

```
MATCH (actor1:Actor)-[:ACTED_IN]->(:Movie)<-[:ACTED_IN]-(common)-[:ACTED_IN]->(:Movie)<-[:ACTED_IN]-(actor2:Actor)
WHERE actor1.name <> "Winston, Hattie"
      AND actor2.name = "Winston, Hattie"
      AND NOT (actor1)-[:ACTED_IN]->(:Movie)<-[:ACTED_IN]-(actor2)
RETURN DISTINCT actor1.name AS ActorName
ORDER BY actor1.name
LIMIT 10;
```

## 2. Pares de personas que han trabajado juntas en más de una película:

- Identifica y lista los pares (actores y/o directores) que han colaborado en múltiples proyectos.
- Script: `pair-persons-most-occurrences.cypher`.

```
MATCH (person1:Person)-[:ACTED_IN|DIRECTED]->(movie:Movie)
WHERE person1.personid <> person2.personid
WITH person1, person2, COUNT(movie) AS collaborations
WHERE collaborations > 1
RETURN person1.name AS Person1, person2.name AS Person2, c
ORDER BY collaborations DESC;
```

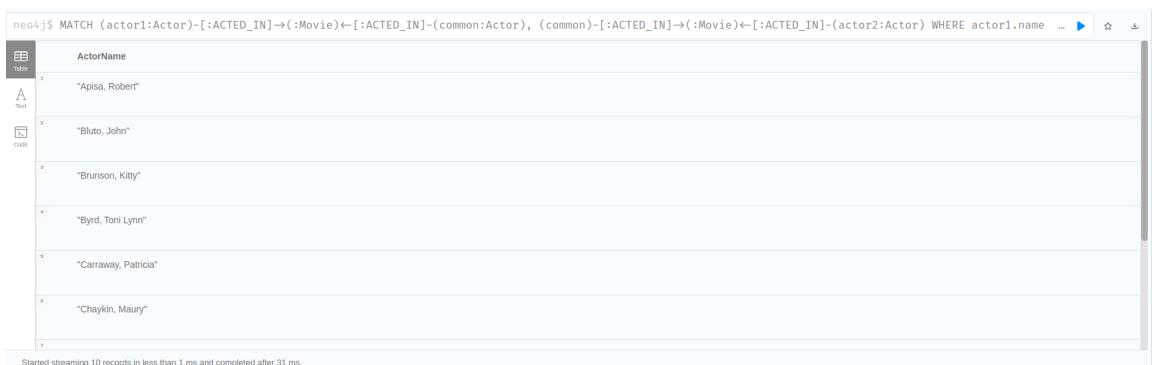
## 3. Camino más corto entre el director "Reiner, Carl" y la actriz "Smyth, Lisa (I)":

- Encuentra la ruta de menor número de conexiones entre el director y la actriz en el grafo.
- Script: `degrees-reiner-to-smyth.cypher`.

```
MATCH p = shortestPath((reiner:Director {name: "Reiner, Ca
RETURN p AS ShortestPath, length(p) AS PathLength;
```

## Pruebas de ejecución:

- 1ª consulta:



The screenshot shows a Neo4j Cypher query execution interface. The query is: `neo4j$ MATCH (actor1:Actor)-[:ACTED_IN]->(movie:Movie)-[:ACTED_IN]-(common:Actor), (common)-[:ACTED_IN]->(movie:Movie)-[:ACTED_IN]-(actor2:Actor) WHERE actor1.name ...`. The results are displayed in a table with the header 'ActorName'. The table contains 7 rows of actor names. At the bottom, a status message reads: 'Started streaming 10 records in less than 1 ms and completed after 31 ms.'

ActorName
"Apisa, Robert"
"Bluto, John"
"Brunson, Kitty"
"Byrd, Toni Lynn"
"Carraway, Patricia"
"Chaykin, Maury"

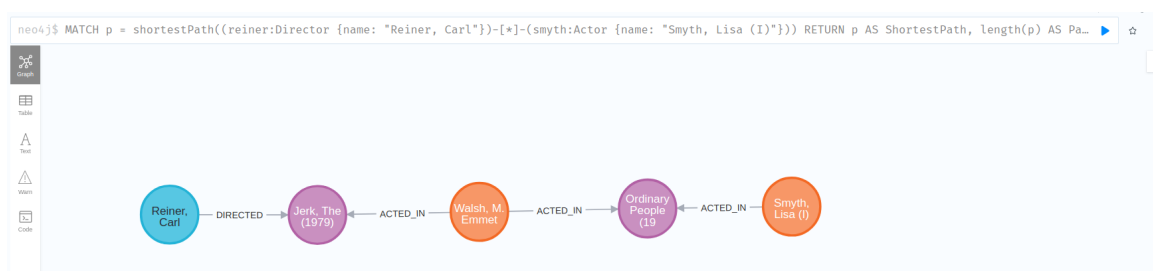
- 2ª consulta:

```
neo4j$ MATCH (person1:Person)-[:ACTED_IN|DIRECTED]-(movie:Movie)-[:ACTED_IN|DIRECTED]-(person2:Person) WHERE person1.personid <> person2.personid WITH pers...
```

	Person1	Person2	collaborations
1	"Walsh, M. Emmet"	"Macey, Elizabeth"	2
2	"Macey, Elizabeth"	"Walsh, M. Emmet"	2

Started streaming 2 records in less than 1 ms and completed after 20 ms.

- 3ª consulta:



Como podemos observar, los resultados son correctos y se corresponden totalmente con los datos de la base de datos original (postgres).

## 2. Transacciones

### Apartado A

Para este apartado se nos solicita que implementemos una API REST en Python que gestione transacciones en una base de datos PostgreSQL, específicamente para borrar clientes de una ciudad junto con toda la información asociada, como pedidos, detalles de pedidos y carritos. El endpoint principal, `/borraCiudad`, permite realizar esta operación de manera controlada, asegurando que los cambios sean consistentes y que cualquier error en el proceso se maneje adecuadamente mediante un rollback.

La API utiliza SQLAlchemy para gestionar la conexión con la base de datos, deshabilitando el autocommit para asegurar que las transacciones sean manuales. Esto permite un control preciso sobre el flujo de operaciones, implementando tanto un orden correcto de borrado como simulando errores al ejecutar la eliminación en un orden incorrecto. Además, se han añadido opciones como commits intermedios para explorar cómo afectan al comportamiento del rollback.

El diseño garantiza la integridad de los datos y ofrece retroalimentación clara al cliente sobre el resultado de las operaciones, ya sea exitoso o fallido. Este enfoque no solo cumple con los requisitos funcionales, sino que también demuestra el manejo robusto de transacciones en un entorno realista, considerando escenarios de error y proporcionando mecanismos de recuperación efectivos.

api.py

```
from quart import Quart, request, jsonify
from sqlalchemy.ext.asyncio import create_async_engine, AsyncSession
from sqlalchemy.exc import IntegrityError, SQLAlchemyError
from sqlalchemy.sql import text

app = Quart(__name__)

# Configuración de la base de datos
DATABASE_URI = "postgresql+asyncpg://alumnodb:1234@localhost:5432"
engine = create_async_engine(DATABASE_URI, execution_options={"isolation_level": "AUTOCOMMIT"})

@app.route("/borraCiudad", methods=["POST"])
async def borra_ciudad():
    """
    Punto de acceso que borra todos los clientes de una ciudad
    """
    data = await request.json
    city = data.get("city")
    use_wrong_order = data.get("use_wrong_order", False)  # P
    commit_before_error = data.get("commit_before_error", False)
```

```

if not city:
    return jsonify({"error": "Falta el parámetro 'city'"})

# Verificamos si hay clientes en la ciudad especificada
async with engine.connect() as conn:
    result = await conn.execute(
        text("SELECT COUNT(*) FROM customers WHERE city =
            {"city": city}
        )
    )
    count = result.scalar() # Obtiene el valor de la pri

    if count == 0:
        return jsonify({"error": f"No hay clientes en la

async with engine.connect() as conn: # Usamos conexión a
    trans = await conn.begin() # Iniciamos la transacción
    try:
        print(f"Iniciando transacción para borrar cliente

        if commit_before_error:
            print("Haciendo un COMMIT intermedio...")
            await trans.commit()
            trans = await conn.begin()

        if use_wrong_order:
            # Intento de borrado en orden incorrecto para
            print("Usando orden incorrecto para el borrado
            await conn.execute(
                text("DELETE FROM customers WHERE city =
                    {"city": city}
                )

            # Esto debería fallar si hay restricciones de
            await conn.execute(
                text("DELETE FROM orders WHERE customerid
                    {"city": city}
                )

```

```

else:
    # Orden correcto de borrado
    print("Usando orden correcto para el borrado.")
    await conn.execute(
        text("DELETE FROM orderdetail WHERE orderid = "
              "(SELECT orderid FROM orders WHERE customerid = "
              "(SELECT customerid FROM customers WHERE city = "
              "{city": city}
              )
              )
    )

    await conn.execute(
        text("DELETE FROM orders WHERE customerid = "
              "{city": city}
              )
    )
    await conn.execute(
        text("DELETE FROM customers WHERE city = "
              "{city": city}
              )
    )

    # Commit de la transacción si todo salió bien
    await trans.commit()
    print("Transacción completada con éxito.")
    return jsonify({"message": f"Clientes de la ciudad {city} eliminados"})

except IntegrityError as ie:
    print(f"Error de integridad detectado: {ie}")
    await trans.rollback()
    print("Rollback realizado debido a un error de integridad")
    return jsonify({"error": "Error de integridad durante la transacción"})

except SQLAlchemyError as e:
    print(f"Error de la base de datos detectado: {e}")
    await trans.rollback()
    print("Rollback realizado debido a un error en la base de datos")
    return jsonify({"error": "Error en la base de datos"})

except Exception as e:

```



```

        print(f"Error desconocido: {e}")
        await trans.rollback()
        print("Rollback realizado debido a un error desco
        return jsonify({"error": "Error desconocido, camb

if __name__ == "__main__":
    app.run(debug=True)

```

A parte de la API se nos solicita que implementemos un script SQL que se encarga de modificar las restricciones de clave foránea en la base de datos para gestionar de manera más controlada la eliminación de registros relacionados. Inicialmente, eliminamos las restricciones `ON DELETE CASCADE` de las tablas `orders` y `orderdetail`, lo que implica que, en caso de eliminar un cliente o un pedido, los registros relacionados no se eliminarán automáticamente. Esto permite tener un control más preciso sobre qué datos se deben borrar en una transacción.

A continuación, el script añade nuevamente las mismas restricciones de clave foránea, pero sin la opción `ON DELETE CASCADE`. De esta manera, aunque se mantendrá la integridad referencial entre las tablas, las eliminaciones de clientes o pedidos no afectarán automáticamente a los registros relacionados en `orders` y `orderdetail`. Esto es esencial para realizar operaciones de borrado más manuales, como las requeridas en el ejercicio de la API, donde los registros asociados deben eliminarse explícitamente en un orden específico para evitar errores de integridad referencial.

Este enfoque permite un mayor control durante las operaciones de borrado, especialmente cuando se está trabajando con transacciones que implican varias tablas interrelacionadas.

`actualiza.sql`

```

-- Eliminamos restricciones ON DELETE CASCADE
ALTER TABLE orders DROP CONSTRAINT IF EXISTS rel_orders_custome
ALTER TABLE orderdetail DROP CONSTRAINT IF EXISTS rel_orderde

-- Añadimos restricciones sin ON DELETE CASCADE
ALTER TABLE orders

```

```

ADD CONSTRAINT rel_orders_customers
FOREIGN KEY (customerid)
REFERENCES customers (customerid);

ALTER TABLE orderdetail
ADD CONSTRAINT rel_orderdetail_orders
FOREIGN KEY (orderid)
REFERENCES orders (orderid);

```

Para probar las implementaciones anteriores, hemos realizado un cliente python para probar la funcionalidad completa:

`cliente.py`

```

import requests
import json

# Configuración del servidor
API_URL = "http://localhost:5000/borraCiudad"

def test_borra_ciudad(city, use_wrong_order=False, commit_before_error=False):
    """
    Envía una solicitud POST al servidor para probar el borrado de una ciudad.

    Args:
        city (str): Ciudad cuyos clientes se desean borrar.
        use_wrong_order (bool): Si se usa el orden incorrecto para el borrado.
        commit_before_error (bool): Si se realiza un COMMIT antes de borrar.

    """
    data = {
        "city": city,
        "use_wrong_order": use_wrong_order,
        "commit_before_error": commit_before_error
    }

    print(f"\n=== Probando borrado para ciudad: '{city}'")
    print(f"Parámetros: Orden incorrecto: {use_wrong_order}, Commit antes de borrar: {commit_before_error}")

```

```

try:
    response = requests.post(API_URL, json=data)
    print("Respuesta del servidor:", response.json())
    if response.status_code == 200:
        print("✓ Éxito: ", response.json()["message"])
    else:
        print("✗ Error: ", response.json()["error"])
except requests.RequestException as e:
    print(f"Error de conexión con el servidor: {e}")

def menu():
    """
    Menú principal para probar las funcionalidades del cliente
    """
    while True:
        print("\n--- Cliente de Pruebas para borraCiudad")
        print("1. Borrar clientes con orden correcto")
        print("2. Borrar clientes con orden incorrecto (rollback)")
        print("3. Borrar clientes con COMMIT intermedio")
        print("4. Salir")

        choice = input("Seleccione una opción: ")

        if choice == "1":
            city = input("Ingrese la ciudad: ").strip()
            test_borra_ciudad(city, use_wrong_order=False)

        elif choice == "2":
            city = input("Ingrese la ciudad: ").strip()
            test_borra_ciudad(city, use_wrong_order=True)

        elif choice == "3":
            city = input("Ingrese la ciudad: ").strip()
            test_borra_ciudad(city, use_wrong_order=True)

        elif choice == "4":

```

```

        print("Saliendo del cliente.")
        break

    else:
        print("Opción no válida. Por favor, elija un número")

if __name__ == "__main__":
    print("Iniciando cliente para pruebas de API...")
    menu()

```

Para probar el correcto funcionamiento del cliente y, por tanto, de la api, vamos a realizar la siguiente prueba de validación:

## Prueba de validación

- **Borrado en orden correcto:**

Vamos a eliminar lo relativo a los clientes que viven en la ciudad sadie  
Estado de la base de datos previo a la eliminación:

```

sql=# SELECT * FROM customers WHERE city = 'sadie';
 customerid | firstname | lastname | address1 | address2 | city | state | zip | country | region | email | phone | creditcardtype | creditcard | cre
-----
2731 | shroud | mirror | judea starve 108 | expect jew | sadie | fax | 42065 | Spain | | elias | shroud.mirror@nanoot.com | +39 815826855 | American | 4411802203434666 | 201
285 | | altair | smock | 52 | 45780 | M | | | | | | | | | |
5999 | bank | nelsen | tensor cozen 298 | from dude | sadie | cream | 37551 | Czech Republic | slay | bank.nelsen@jmail.com | +15 595171852 | Mastercard | 4757984816474883 | 201
309 | | abe | macao | 24 | 30262 | F | | | | | | | | | |
8809 | gushy | velor | alton plate 154 | manna pipplin | sadie | jazzy | 22722 | Hong Kong | basso | gushy.veor@potnail.com | +6 867070128 | VISA | 4855047020313569 | 201
283 | | groove | marine | 59 | 19246 | F | | | | | | | | | |
13057 | decant | yokel | fount retort 19 | slant butick | sadie | sydney | 31887 | Denmark | kirby | decant.yokel@nanoot.com | +30 286845978 | VISA | 4195065254145986 | 201
309 | | bet | aton | 40 | 40385 | F | | | | | | | | | |
(4 rows)

```

```

si1=# SELECT * FROM orders WHERE customerid = 2731;

```

orderid	orderdate	customerid	netamount	tax	totalamount	status
35716	2021-08-26	2731	92.5	18	109.15	Paid
35710	2020-10-26	2731	140.5825242718446600	18	165.89	Processed
35711	2019-07-12	2731	14.4244105409153952	15	16.59	Shipped
35717	2017-06-20	2731	123.7170596393897366	15	142.27	Shipped
35713	2017-08-17	2731	114.1007859454461397	15	131.22	Shipped
35714	2017-08-07	2731	37.9103097549699491	15	43.60	Shipped
35709	2017-06-20	2731	71.3823393435043922	15	82.09	Processed
35715	2020-11-23	2731	38.8349514563106796	18	45.83	Paid
35707	2016-09-12	2731	77.6699029126213591	15	89.32	Shipped
35708	2020-12-29	2731	157.2815533980582524	18	185.59	Shipped
35712	2018-07-13	2731	164.6786870087840964	15	189.38	Processed

(11 rows)

```

si1=# SELECT * FROM orderdetail WHERE orderid = 35716;

```

orderid	prod_id	price	quantity
35716	2328	13.2	1
35716	4858	10	1
35716	5803	22.8	1
35716	4163	19.5	1
35716	6564	13	1
35716	4382	14	1

(6 rows)

Pd: hemos elegido como usuario de prueba específico al usuario con `customerid = 2731` y como pedido el que tiene `orderid = 35716`.

Proceso de eliminación:

```

marcosi701@marcosi701-VirtualBox:~/Escritorio/Marcos.M/SI/SI/P3/transacciones$ python3 cliente.py
Iniciando cliente para pruebas de API...

--- Cliente de Pruebas para borraCiudad ---
1. Borrar clientes con orden correcto
2. Borrar clientes con orden incorrecto (provocar error)
3. Borrar clientes con COMMIT intermedio y luego error
4. Salir
Seleccione una opción: 1
Ingrese la ciudad: sadie

=== Probando borrado para ciudad: 'sadie' ===
Parámetros: Orden incorrecto: False, Commit intermedio: False
Respuesta del servidor: {'message': "Clientes de la ciudad 'sadie' borrados con éxito."}
✓ Éxito: Clientes de la ciudad 'sadie' borrados con éxito.

```

Estado de la base de datos previo a la eliminación:

```

si1=# SELECT * FROM customers WHERE city = 'sadie';

```

customerid	firstname	lastname	address1	address2	city	state	zip	country	region	email	phone	creditcardtype	creditcard	creditcardexpiration	username	password	age	income
gender																		

(0 rows)

```

si1=# SELECT * FROM orders WHERE customerid = 2731;
 orderid | orderdate | customerid | netamount | tax | totalamount | status
-----+-----+-----+-----+-----+-----+-----
(0 rows)

si1=# SELECT * FROM orderdetail WHERE orderid = 35716;
 orderid | prod_id | price | quantity
-----+-----+-----+-----
(0 rows)

```

Como podemos comprobar, la eliminación se ha realizado perfectamente, por lo que podemos concluir que la implementación es correcta.

- **Borrado en orden incorrecto:**

Estado de la base de datos previo a la eliminación:

```

si1=# SELECT * FROM customers WHERE city = 'sadie';
customerid | firstname | lastname | address1 | address2 | city | state | zip | country | region | email | phone | creditcardtype | creditcard | cre
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
(4 rows)

```

customerid	firstname	lastname	address1	address2	city	state	zip	country	region	email	phone	creditcardtype	creditcard	cre
2731	shroud	mirror	judea starve	108   expect jew	sadie	fax	42065	Spain	elias	shroud.mirror@nanoot.com	+39 815826855	American	4411802203434666	201
5999	bank	nelsen	tensor cozen	298   from dude	sadie	cream	37551	Czech Republic	slay	bank.nelsen@nail.com	+15 505171852	Mastercard	4757984816474883	201
8809	gushy	velor	alton plate	154   manna pippin	sadie	jazzy	22722	Hong Kong	basso	gushy.velor@potnail.com	+6 807070128	VISA	4855047020313569	201
13057	decant	yokel	fount retort	19   slant buick	sadie	sydney	31887	Denmark	kirby	decant.yokel@nanoot.com	+30 286845978	VISA	4195065254145986	201

Proceso de eliminación:

```

--- Cliente de Pruebas para borraCiudad ---
1. Borrar clientes con orden correcto
2. Borrar clientes con orden incorrecto (provocar error)
3. Borrar clientes con COMMIT intermedio y luego error
4. Salir
Seleccione una opción: 2
Ingrese la ciudad: sadie

=== Probando borrado para ciudad: 'sadie' ===
Parámetros: Orden incorrecto: True, Commit intermedio: False
Respuesta del servidor: {'error': 'Error de integridad durante el borrado, cambios deshechos.'}
✗ Error: Error de integridad durante el borrado, cambios deshechos.

```

Estado de la base de datos previo a la eliminación:

```

si1=# SELECT * FROM customers WHERE city = 'sadie';
customerid | firstname | lastname | address1 | address2 | city | state | zip | country | region | email | phone | creditcardtype | creditcard | cre
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
(4 rows)

```

customerid	firstname	lastname	address1	address2	city	state	zip	country	region	email	phone	creditcardtype	creditcard	cre
2731	shroud	mirror	judea starve	108   expect jew	sadie	fax	42065	Spain	elias	shroud.mirror@nanoot.com	+39 815826855	American	4411802203434666	201
5999	bank	nelsen	tensor cozen	298   from dude	sadie	cream	37551	Czech Republic	slay	bank.nelsen@nail.com	+15 505171852	Mastercard	4757984816474883	201
8809	gushy	velor	alton plate	154   manna pippin	sadie	jazzy	22722	Hong Kong	basso	gushy.velor@potnail.com	+6 807070128	VISA	4855047020313569	201
13057	decant	yokel	fount retort	19   slant buick	sadie	sydney	31887	Denmark	kirby	decant.yokel@nanoot.com	+30 286845978	VISA	4195065254145986	201

Como podemos observar, aunque introducamos datos correctos para la eliminación de usuarios, si lo hacemos en un orden incorrecto, la función fallará debido a la restricción de integridad. Al fallar, la función realiza un rollback y deja los datos tal y como estaban.

- **Borrado con commit intermedio:**

Cuando se realiza un commit intermedio en medio de una transacción, los cambios hasta ese punto se confirman en la base de datos. Si luego ocurre un error, los cambios posteriores al commit no serán guardados, y la transacción se revertirá para esas operaciones posteriores. Esto permite preservar la integridad de los datos hasta el momento del commit intermedio, mientras que los intentos fallidos después de ese punto se deshacen, garantizando que no se queden datos inconsistentes en la base de datos.

## ***Apartado B***

Para la primera parte del apartado, hemos creado un script que añada la columna promo a la tabla customers y que a su vez contenga un trigger que se active en el momento en el que se modifique la columna anteriormente mencionada y actualice el precio de los pedidos pertinentes

updPromo.sql

```
-- Añadimos la columna 'promo' para descuentos en clientes
ALTER TABLE customers ADD COLUMN promo DECIMAL(5, 2) DEFAULT 0;

-- Función para actualizar los precios en el carrito del cliente
CREATE OR REPLACE FUNCTION update_cart_price() RETURNS TRIGGER AS $$
BEGIN
    -- Pausa de 5 segundos durante la ejecución del trigger
    PERFORM pg_sleep(5);

    -- Actualizamos el precio de los pedidos basándonos en el
```

```

        UPDATE orders
        SET netamount = netamount * (1 - NEW.promo / 100)
        WHERE orderid IN (SELECT orderid FROM orders WHERE custom
        RETURN NEW;
    END;
$$ LANGUAGE plpgsql;

-- Creamos el trigger que se activará después de actualizar '
CREATE TRIGGER after_promo_update
AFTER UPDATE OF promo ON customers
FOR EACH ROW
EXECUTE FUNCTION update_cart_price();

```

Para implementar la nueva función `borraCiudad`, que añade un `pg_sleep` durante la ejecución, hemos codificado el siguiente fichero:

`borraCiudad.py`

```

from sqlalchemy.ext.asyncio import create_async_engine, Async
from sqlalchemy.exc import IntegrityError, SQLAlchemyError
from sqlalchemy.sql import text
import asyncio

# Configuración de la base de datos
DATABASE_URI = "postgresql+asyncpg://alumnodb:1234@localhost:
engine = create_async_engine(DATABASE_URI, execution_options=

async def borra_ciudad(city, use_wrong_order=False, commit_be
    """
    Función que borra todos los clientes de una ciudad y su i
    :param city: Ciudad cuyos clientes se desean eliminar.
    :param use_wrong_order: Si se debe usar un orden incorrec
    :param commit_before_error: Si se debe hacer un commit in
    """

    # Verificar si existen clientes en la ciudad
    async with engine.connect() as conn:
        result = await conn.execute(

```



```

        text("SELECT COUNT(*) FROM customers WHERE city =
            {"city": city}
        )
count = result.scalar()

if count == 0:
    print(f"No hay clientes en la ciudad '{city}'.")
    return

async with engine.connect() as conn: # Iniciamos una con
    trans = await conn.begin() # Iniciamos la transacción
    try:
        print(f"Iniciando transacción para borrar cliente

        if commit_before_error:
            print("Haciendo un COMMIT intermedio...")
            await trans.commit()
            trans = await conn.begin()

        if use_wrong_order:
            # Intento de borrado en orden incorrecto
            print("Usando orden incorrecto para el borrado")
            await conn.execute(
                text("DELETE FROM customers WHERE city =
                    {"city": city}
                )
            )
            await conn.execute(
                text("DELETE FROM orders WHERE customerid
                    "(SELECT customerid FROM customers W
                    {"city": city}
                )
            )
        else:
            # Orden correcto de borrado
            print("Usando orden correcto para el borrado.")
            await conn.execute(
                text("DELETE FROM orderdetail WHERE order.
                    "(SELECT orderid FROM orders WHERE c
                    "(SELECT customerid FROM customers W

```

```

        {"city": city}
    )

    # Simulamos un retraso con pg_sleep
    print("Simulando latencia con pg_sleep...")
    await conn.execute(text("SELECT pg_sleep(10)")

    await conn.execute(
        text("DELETE FROM orders WHERE customerid
              "(SELECT customerid FROM customers W
              {"city": city}
        )
    await conn.execute(
        text("DELETE FROM customers WHERE city =
              {"city": city}
        )

    # Commit de la transacción
    await trans.commit()
    print(f"Clientes de la ciudad '{city}' borrados c

except IntegrityError as ie:
    print(f"Error de integridad detectado: {ie}")
    await trans.rollback()
    print("Rollback realizado debido a un error de in

except SQLAlchemyError as e:
    print(f"Error de la base de datos detectado: {e}")
    await trans.rollback()
    print("Rollback realizado debido a un error en la

except Exception as e:
    print(f"Error desconocido: {e}")
    await trans.rollback()
    print("Rollback realizado debido a un error desco

# Ejecución principal para realizar pruebas
if __name__ == "__main__":

```

```
# Modificar estos parámetros según los casos de prueba ne
city_to_delete = "timur"
use_wrong_order = False # Cambiar a True para usar un or
commit_before_error = False # Cambiar a True para probar

asyncio.run(borra_ciudad(city_to_delete, use_wrong_order,
```

Introducimos los siguientes pedidos con status a NULL mediante la sentencia UPDATE:

Estado de la base de datos previo al cambio:

```

si1=# SELECT * FROM orders WHERE customerid = 1;

```

orderid	orderdate	customerid	netamount	tax	totalamount	status
108	2020-01-31	1	41.6088765603328709	15	47.85	Shipped
107	2016-10-28	1	130.7443365695792881	15	150.36	Shipped
104	2019-03-07	1	26.6296809986130374	15	30.62	Shipped
105	2016-11-29	1	12.2052704576976422	15	14.04	Processed
106	2019-12-31	1	26.8146093388811836	15	30.84	Shipped
103	2016-12-28	1	25.1502542764678688	15	28.92	Shipped

(6 rows)

```

si1=# SELECT * FROM orders WHERE customerid = 2;

```

orderid	orderdate	customerid	netamount	tax	totalamount	status
114	2019-05-30	2	127.6930189551548773	15	146.85	Paid
115	2019-06-21	2	122.4225612575127138	15	140.79	Shipped
111	2017-03-01	2	55.4785020804438278	15	63.80	Shipped
113	2018-02-07	2	131.4840499306518725	15	151.21	Shipped
109	2018-07-08	2	68.0536292186777623	15	78.26	Shipped
112	2021-02-21	2	117.1	18	138.18	Paid
110	2021-10-02	2	87.8	18	103.60	Processed

(7 rows)

Comando ejecutado:

```
UPDATE orders SET status = NULL WHERE customerid = 1;
```

```
UPDATE orders SET status = NULL WHERE customerid = 2;
```

Estado de la base de datos tras el cambio:

```

si1=# UPDATE orders SET status = NULL WHERE customerid = 1;
UPDATE 6
si1=# UPDATE orders SET status = NULL WHERE customerid = 2;
UPDATE 7
si1=# SELECT * FROM orders WHERE customerid = 1;

```

orderid	orderdate	customerid	netamount	tax	totalamount	status
108	2020-01-31	1	41.6088765603328709	15	47.85	
107	2016-10-28	1	130.7443365695792881	15	150.36	
104	2019-03-07	1	26.6296809986130374	15	30.62	
105	2016-11-29	1	12.2052704576976422	15	14.04	
106	2019-12-31	1	26.8146093388811836	15	30.84	
103	2016-12-28	1	25.1502542764678688	15	28.92	

(6 rows)

```

si1=# SELECT * FROM orders WHERE customerid = 2;

```

orderid	orderdate	customerid	netamount	tax	totalamount	status
114	2019-05-30	2	127.6930189551548773	15	146.85	
115	2019-06-21	2	122.4225612575127138	15	140.79	
111	2017-03-01	2	55.4785020804438278	15	63.80	
113	2018-02-07	2	131.4840499306518725	15	151.21	
109	2018-07-08	2	68.0536292186777623	15	78.26	
112	2021-02-21	2		117.1	18	138.18
110	2021-10-02	2		87.8	18	103.60

(7 rows)

Tras haber cambiado los valor de status a null, el enunciado nos solicita que comprobemos si al realizar el cambio en la columna promo, también se actualiza el valor del netamount de los pedidos correspondientes a ese usuario. Para ello hemos hecho la siguiente comprobación:

Estado de la base de datos previo al update:

```

si1=# SELECT * FROM orders WHERE customerid = 1;

```

orderid	orderdate	customerid	netamount	tax	totalamount	status
108	2020-01-31	1	41.6088765603328709	15	47.85	
107	2016-10-28	1	130.7443365695792881	15	150.36	
104	2019-03-07	1	26.6296809986130374	15	30.62	
105	2016-11-29	1	12.2052704576976422	15	14.04	
106	2019-12-31	1	26.8146093388811836	15	30.84	
103	2016-12-28	1	25.1502542764678688	15	28.92	

(6 rows)

Comando ejecutado:

```
UPDATE customers SET promo = 10 WHERE customerid = 1;
```

Estado de la base de datos tras el update:

```
sll=# SELECT * FROM orders WHERE customerid = 1;
orderid | orderdate | customerid | netamount | tax | totalamount | status
```

108	2020-01-31	1	37.447988904299583810000000000000000000	15	47.85	
107	2016-10-28	1	117.669902912621359290000000000000000000	15	150.36	
104	2019-03-07	1	23.966712898751733660000000000000000000	15	30.62	
105	2016-11-29	1	10.984743411927877980000000000000000000	15	14.04	
106	2019-12-31	1	24.133148404993065240000000000000000000	15	30.84	
103	2016-12-28	1	22.635228848821081920000000000000000000	15	28.92	

```
(6 rows)
```

Como podemos observar, el valor del precio final (netamount) ha cambiado al actualizar el valor del campo promo del usuario con customerid = 1, lo que nos indica que el trigger está desempeñando su tarea a la perfección.

A continuación vamos a estudiar el bloqueo de ciertos datos de la base de datos cuando ocurren actualizaciones simultáneas. Para ello, vamos a realizar pruebas con dos sesiones diferentes que intentan acceder y modificar los mismos datos al mismo tiempo, prestando especial atención a los bloqueos que se generan y cómo se resuelven.

Estado de la base de datos previo al update:

```

si1=# SELECT customerid, city, promo FROM customers WHERE customerid = 1;
 customerid | city  | promo
-----+-----+-----
           1 | timur | 20.00
(1 row)

```

Comando de actualizacion ejecutado:

```
UPDATE customers SET promo = 10 WHERE customerid = 1;
```

Comprobación del estado del campo alterado, durante la ejecución del trigger:

```

si1=# SELECT customerid, city, promo FROM customers WHERE customerid = 1;
 customerid | city  | promo
-----+-----+-----
           1 | timur | 20.00
(1 row)

```

Como podemos observar, durante la ejecución de la función de del trigger (`update_cart_price`), si accedemos al valor que se ha solicitado modificar, observamos que el cambio aún no se ha reflejado en la base de datos debido a que PostgreSQL sigue el modelo de MVCC (Control de Concurrency Multiversion). Las transacciones en curso no exponen sus cambios hasta que se hace un `COMMIT`.

Para estudiar los bloqueos que suceden mientras duren los sleep vamos a ejecutar la siguiente consulta, que nos permitirá obtener información sobre los bloqueos actuales:

```
SELECT
pg_locks.locktype,
pg_stat_activity.datname,
pg_stat_activity.username,
pg_stat_activity.pid,
pg_stat_activity.query,
pg_locks.mode,
pg_locks.granted
FROM pg_locks
JOIN pg_stat_activity
ON pg_locks.pid = pg_stat_activity.pid
WHERE pg_stat_activity.datname = 'si1';
```

Tras la ejecución de la consulta (mientras se ejecuta el sleep), obtenemos la siguiente salida:

locktype	datname	username	pid	
relation	si1	alumnodb	326	SELECT pg_sleep(1
relation	si1	alumnodb	326	SELECT pg_sleep(1
relation	si1	alumnodb	326	SELECT pg_sleep(1
relation	si1	alumnodb	326	SELECT pg_sleep(1
relation	si1	alumnodb	326	SELECT pg_sleep(1
virtualxid	si1	alumnodb	326	SELECT pg_sleep(1
relation	si1	alumnodb	146	SELECT
				pg_locks.lockty
				pg_stat_activit

				pg_stat_activit
				pg_stat_activit
				pg_stat_activit
				pg_locks.mode,
				pg_locks.grante
				FROM pg_locks
				JOIN pg_stat_acti
				ON pg_locks.pid =
				WHERE pg_stat_act
relation	si1	alumnodb	146	SELECT
				pg_locks.lockty
				pg_stat_activit
				pg_stat_activit
				pg_stat_activit
				pg_stat_activit
				pg_locks.mode,
				pg_locks.grante
				FROM pg_locks
				JOIN pg_stat_acti
				ON pg_locks.pid =
				WHERE pg_stat_act
virtualxid	si1	alumnodb	146	SELECT
				pg_locks.lockty
				pg_stat_activit
				pg_stat_activit
				pg_stat_activit
				pg_stat_activit
				pg_locks.mode,
				pg_locks.grante
				FROM pg_locks
				JOIN pg_stat_acti
				ON pg_locks.pid =
				WHERE pg_stat_act
transactionid	si1	alumnodb	326	SELECT pg_sleep(1
relation	si1	alumnodb	146	SELECT
				pg_locks.lockty
				pg_stat_activit
				pg_stat_activit

				pg_stat_activit
				pg_stat_activit
				pg_locks.mode,
...skipping 1 line				
				FROM pg_locks
				JOIN pg_stat_acti
				ON pg_locks.pid =
				WHERE pg_stat_act
relation	si1	alumnodb	146	SELECT
				pg_locks.lockty
				pg_stat_activit
				pg_stat_activit
				pg_stat_activit
				pg_stat_activit
				pg_locks.mode,
				pg_locks.grante
				FROM pg_locks
				JOIN pg_stat_acti
				ON pg_locks.pid =
				WHERE pg_stat_act
relation	si1	alumnodb	146	SELECT
				pg_locks.lockty
				pg_stat_activit
				pg_stat_activit
				pg_stat_activit
				pg_stat_activit
				pg_locks.mode,
				pg_locks.grante
				FROM pg_locks
				JOIN pg_stat_acti
				ON pg_locks.pid =
				WHERE pg_stat_act
relation	si1	alumnodb	146	SELECT
				pg_locks.lockty
				pg_stat_activit
				pg_stat_activit
				pg_stat_activit



				pg_stat_activit
				pg_locks.mode,
				pg_locks.grante
				FROM pg_locks
				JOIN pg_stat_acti
				ON pg_locks.pid =
				WHERE pg_stat_act
relation	si1	alumnodb	146	SELECT
				pg_locks.lockty
				pg_stat_activit
				pg_stat_activit
				pg_stat_activit
				pg_stat_activit
				pg_locks.mode,
				pg_locks.grante
				FROM pg_locks
				JOIN pg_stat_acti
				ON pg_locks.pid =
				WHERE pg_stat_act
...skipping 1 line				
				pg_locks.lockty
				pg_stat_activit
				pg_stat_activit
				pg_stat_activit
				pg_stat_activit
				pg_locks.mode,
				pg_locks.grante
				FROM pg_locks
				JOIN pg_stat_acti
				ON pg_locks.pid =
				WHERE pg_stat_act
(16 rows)				

La salida muestra los bloqueos actuales en la base de datos

`si1` para las consultas activas. Basandose en la información de las vistas del sistema `pg_locks` y `pg_stat_activity`.

## Bloqueos observados durante la ejecución del trigger

Cuando el trigger introduce un retraso ( `pg_sleep` ), analizamos los bloqueos en la base de datos. Durante este tiempo:

### 1. Tipos de bloqueos:

- **RowExclusiveLock** : Este es el principal bloqueo usado por la transacción que activa el trigger, ya que intenta modificar filas en las tablas `customers` y `orders`.
- **AccessShareLock** : Ocurre cuando otras sesiones consultan tablas afectadas por el trigger. Estas consultas no se bloquean mientras no haya modificaciones concurrentes.
- **ExclusiveLock** : Se aplica a la transacción para garantizar la consistencia de las operaciones.

### 2. Propagación de bloqueos:

- El trigger afecta no solo a la tabla `customers`, sino también a `orders`, debido a la consulta de actualización que realiza internamente.
- Si una segunda transacción intenta modificar los mismos datos, quedará bloqueada hasta que la transacción activa libere los recursos.

### 3. Interacción entre sesiones:

- Mientras una transacción está activa con `pg_sleep`, otra sesión puede leer los datos bloqueados, pero no verá los cambios no confirmados debido al modelo de control de concurrencia multiversión (MVCC).
- Si se intenta escribir en recursos bloqueados, la operación se aplazará hasta que se liberen los bloqueos.

**Conclusión:** Este comportamiento es esencial para la integridad de los datos, pero puede introducir problemas de concurrencia en sistemas con alta carga. Por ello, es importante diseñar triggers y transacciones con cuidado, minimizando los tiempos de bloqueo ( en nuestro caso, simplemente quitando el sleep jajaj).

Para discutir la última parte de la práctica, se nos solicita ajustar el punto en el que se realizan los scripts con la finalidad de crear un deadlock. Para ello vamos a utilizar los dos `pg_sleep` que implementamos en nuestro trigger y en la función `borraCiudad`, para que ambos esperen recursos que están bloqueados por la otra, creando un ciclo de dependencias imposible de resolver.

Para lograrlo vamos a lanzar dos transacciones simultáneas en diferentes sesiones:

- **Sesión 1:** Actualizamos `promo` de un cliente con pedidos en curso:

```
BEGIN;
```

```
UPDATE customers SET promo = 20 WHERE city = 'timur';
```

- **Sesión 2:** Llamamos a `borraCiudad` para la misma ciudad:  
`python3 borraCiudad.py`

```
marcos1701@marcos1701-VirtualBox:~/Escritorio/Marcos.M/SI/SI/P3/transacciones/Apartado B$ python3 borraCiudad.py
Iniciando transacción para borrar clientes de la ciudad 'timur'
Usando orden correcto para el borrado.
Simulando latencia con pg_sleep...
```

```
si1=# BEGIN;
UPDATE customers SET promo = 20 WHERE city = 'timur';
BEGIN
```

Como vemos, ambos procesos se quedan interbloqueados a la espera de que se liberen los recursos que el otro tiene bloqueados. Vamos a detallar más en profundidad lo que ocurre:

## Secuencia de Eventos

### 1. Primera Sesión (UPDATE):

- La sentencia `UPDATE customers SET promo = 30 WHERE city = 'timur';` intenta actualizar todos los registros de la tabla `customers` donde la ciudad sea `'timur'`.

- Esta operación activa un **trigger** llamado `update_cart_price` definido en la base de datos, que a su vez actualiza los registros relacionados en la tabla `orders` para reflejar los nuevos valores de promoción.
- Durante este proceso, se adquieren varios bloqueos:
  - **Bloqueos en las filas de la tabla** `customers` .
  - **Bloqueos en las filas de las tablas** `orders` y `orderdetail` .

## 2. Segunda Sesión (Función `borraCiudad`):

- La función intenta borrar todos los datos relacionados con los clientes de `'timur'`, comenzando por los detalles de las órdenes ( `orderdetail` ), luego las órdenes ( `orders` ), y finalmente los clientes ( `customers` ).
- Se ejecuta un `pg_sleep(10)` que simula latencia, lo que mantiene bloqueada la transacción por más tiempo.
- Esta operación también adquiere bloqueos:
  - **Bloqueos en las filas de la tabla** `orderdetail` .
  - **Bloqueos en las filas de la tabla** `orders` .
  - **Bloqueos en las filas de la tabla** `customers` .

## 3. Conflicto:

- **Sesión 1 (UPDATE):** Mientras está actualizando las filas de `orders`, intenta adquirir un bloqueo que la **sesión 2** ya posee en las filas de `customers` o relacionadas.
- **Sesión 2 (borraCiudad):** Mientras está borrando datos, intenta adquirir un bloqueo que la **sesión 1** ya posee en las filas de `orders` o relacionadas.

Todo esto supone, el famoso:...

## 4. Deadlock:

- Ninguna de las dos transacciones puede continuar porque están esperando que la otra libere los bloqueos necesarios.

Debido al interbloqueo creado, el mecanismo utilizado por PostgreSQL cuando detecta un deadlock es elegir una de las transacciones y abortarla:

```

si1=# BEGIN;
UPDATE customers SET promo = 30 WHERE city = 'timur';
BEGIN
ERROR:  deadlock detected
DETAIL:  Process 84 waits for ShareLock on transaction 858; blocked by process 92.
Process 92 waits for ShareLock on transaction 857; blocked by process 84.
HINT:   See server log for query details.
CONTEXT: while updating tuple (9,74) in relation "orders"
SQL statement "UPDATE orders
        SET netamount = netamount * (1 - NEW.promo / 100)
        WHERE orderid IN (SELECT orderid FROM orders WHERE customerid = NEW.customerid)"
PL/pgSQL function update_cart_price() line 7 at SQL statement

```

```

marcos1701@marcos1701-VirtualBox:~/Escritorio/Marcos.M/SI/SI/P3/transacciones/Apartado b$ python3 borraCiudad.py
Iniciando transacción para borrar clientes de la ciudad 'timur'
Usando orden correcto para el borrado.
Simulando latencia con pg_sleep...
Clientes de la ciudad 'timur' borrados con éxito.

```

## 5. Resolución del Deadlock:

- PostgreSQL termina la transacción `UPDATE` con el error **deadlock detected**.
- Esto permite que la transacción de `borraCiudad` finalice con éxito, ya que los recursos que estaba esperando se liberan al abortar la otra transacción.

Por último, como soluciones para afrontar este tipo de problemas, aportamos las siguientes:

1. Mantener un orden consistente de acceso a tablas. Es decir, acceder en un orden que no genere interbloqueados.
2. Optimización de la lógica de la aplicación eliminando latencias innecesarias. En nuestro caso, eliminando los sleep java.
3. Usar herramientas de la base de datos como timeouts, bloqueos explícitos, o la desactivación temporal de triggers. Para evitar que los bloques, si es que ocurren, supongan un colapso total de la base de datos.
4. Implementar reintentos automáticos en caso de fallos.

**Esperemos que haya disfrutado de esta memoria tanto como a nosotros realizarla 😊**