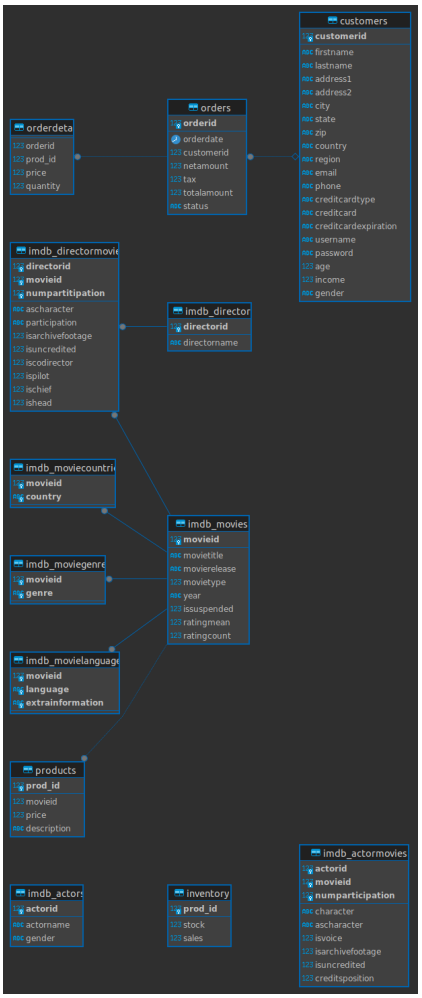
P3 SISTEMAS INFORMÁTICOS

Daniel Birsan y Jorge Paniagua Moreno

1) NoSQL

1.1 Base de datos documental

Partimos de la BBDD dumpeada desde el archivo de la práctica:



Para calcular las películas más relacionadas (f) o relacionadas (g), guardamos los géneros de la película a devolver y la del resto de películas, si el tamaño de la intersección de los conjuntos es el mismo que el del conjunto 1, es f, si es mayor que la mitad del conjunto 1 es g, lo añadimos a mongo, limitamos a 10 películas al final:

```python
# Cálculo de películas relacionadas
num_films = len(films)
for i in range(num_films):
    for j in range(num_films):
        if i != j:
            set1 = set(films[i]["genres"])
            set2 = set(films[j]["genres"])
            intersection = set1.intersection(set2)
            if len(intersection) == len(set1):
                films[i]["most_related_movies"].append(
                    {"title": films[j]["title"], "year": films[j]["year"]}
                )
            elif len(intersection) >= round(len(set1) / 2, 0):
                films[i]["related_movies"].append(
                    {"title": films[j]["title"], "year": films[j]["year"]}
                )
    if len(films[i]["most_related_movies"]) > 10:
        films[i]["most_related_movies"] = films[i]["most_related_movies"][:10]
    if len(films[i]["related_movies"]) > 10:
        films[i]["related_movies"] = films[i]["related_movies"][:10]
```

Lo primero que hay que hacer es lanzar mondongo:

Para crear la BBDD documental con MongoDB hemos implementado un script en python usando SQLAlchemy y la librería pymongo. Primero, usando las librerías anteriores vamos a realizar las conexiones a PostgreSQL y MongoDB:

```python
# Conexión a MongoDB
mongo_client = MongoClient("mongodb://localhost:27017/")
if "si1" in mongo_client.list_database_names():
    print("Removing the old si1 database")
    mongo_client.drop_database('si1')

# Conexión a PostgreSQL
postgres_engine = create_engine("postgresql://alumnodb:1234@localhost/si1", echo=False)
postgres_conn = postgres_engine.connect()
```

Usamos el puerto 27017 ya que es el por defecto de MongoDB, en cuanto a postgres tenemos que usar la autenticación por defecto que se nos ha pedido en las prácticas.

Ahora vamos a recopilar la información para rellenar la BBDD documental, en este caso, películas francesas o de colaboración francesa, para conseguir la información hay que lanzar una query desde el script y almacenar la salida en una variable:

```python
# Consulta SQL para obtener información de películas francesas
query = """
    SELECT
        m.movietitle AS title,
        ARRAY(SELECT genre FROM imdb_moviegenres WHERE movieid = m.movieid) AS genres,
        m.year,
        ARRAY(SELECT imd.directorname FROM imdb_directormovies idm
        NATURAL JOIN imdb_directors imd WHERE idm.movieid = m.movieid) AS directors,
        ARRAY(SELECT ima.actorname FROM imdb_actormovies iam
        NATURAL JOIN imdb_actors ima WHERE iam.movieid = m.movieid) AS actors
    FROM
        imdb_movies m
    JOIN
        imdb_moviecountries mc ON m.movieid = mc.movieid
    WHERE
        mc.country = 'France'
    ORDER BY
        m.year;
"""
postgres_result = postgres_conn.execute(text(query))
movies_data = list(postgres_result)
```

Ahora vamos a crear el formato de la BBDD documental (se nos da en el enunciado de la práctica) y guardamos con el formato la información obtenida con el resultado de la consulta en SQL:

```
films = []
for row in movies_data:
    title = row["title"][:row["title"].rfind("(") - 1]
    year = int(row["year"])
    pelicula = {
        "title": title,
        "genres": row["genres"],
        "year": year,
        "directors": row["directors"],
        "actors": row["actors"],
        "most_related_movies": [],
        "related_movies": []
    }
    films.append(pelicula)
```

Por último, realizamos la conexión con MongoDB e insertamos las películas:

```
# Conexión y carga de datos en MongoDB
mongodb_db = mongo_client["si1"]
mongodb_collection = mongodb_db["france"]
mongodb_collection.insert_many(films)

mongo_client.close()
print("Base de datos documental creada correctamente")
return
```

Nuestra implementación realiza controles de errores y también una comparativa para asegurar la correcta inserción de los datos, ejecutamos la query en postgres para conseguir los títulos de todas las películas francesas para después comparar este título (sin el año) con el insertado en la BBDD documental, esto nos asegura que no haya pérdidas de datos durante la ejecución de nuestro programa.

```
# Comparación de resultados
for postgres_row, mongodb_document in zip(postgres_results, mongodb_results):
    postgres_title_without_year = postgres_row["title"][:postgres_row["title"].rfind("(") - 1]

    if postgres_title_without_year != mongodb_document["title"]:
        print(f"Discrepancia en el título: {postgres_title_without_year} vs {mongodb_document['title']}")

print("Los datos en Mongodb se han guardado correctamente")
```

Salida final:

```
❯ python3 create_mongodb_from_postgresqldb.py
Removing the old si1 database
Base de datos documental creada correctamente
Los datos en Mongodb se han guardado correctamente
 ~/E/SI/P3/app-mongodb-etl  main !1 ?2                                    ✔  6s
```

Consultas

1) Una tabla con toda la información de aquellas películas (documentos) de ciencia ficción comprendidas entre 1994 y 1998:

```python
def query_a(mongodb_collection):
    query = {
        "genres": "Sci-Fi",
        "year": {"$gte": 1994, "$lte": 1998}
    }
    result = list(mongodb_collection.find(query))
    return result
```

```sql
-- QUERY A
SELECT imm.movietitle
FROM imdb_movies imm NATURAL JOIN imdb_moviecountries imc NATURAL JOIN imdb_moviegenres img
WHERE
    imc.country = 'France' and img.genre = 'Sci-Fi' AND CAST(imm.year AS INT) BETWEEN 1994 AND 1998
```

5 rows returned

	movietitle character varying
1	Abre los ojos (1997)
2	Fifth Element, The (1997)
3	Highlander III: The Sorcerer (1994)
4	Nowhere (1997)
5	Stargate (1994)

```
> python3 mongodb_queries.py
Consulta A: Películas de ciencia ficción entre 1994 y 1998
{'_id': ObjectId('657e5adefbe9c89865940adc'), 'title': 'Stargate', 'genres': ['Actior
ohn', 'Giannelli, Steve', 'Gilmore, Jerry', 'Gray, George (II)', 'Holland, Erik', 'Ho
n-Philippe, Michel', 'Kind, Richard', 'Lauchu, Carlos', 'Lee, Kieron', 'Loffler, Gian
nch', 'Storey, John (I)', 'Til, Roger', 'Vint, Kelly', 'Webster, Derek (I)', 'Welker,
est, Christopher (I)'], 'most_related_movies': [], 'related_movies': [{'title': 'Barb
 {'title': 'Cliffhanger', 'year': 1993}, {'title': 'Highlander III: The Sorcerer', 'y
: 1997}, {'title': 'Man in the Iron Mask, The', 'year': 1998}]}
{'_id': ObjectId('657e5adefbe9c89865940ae9'), 'title': 'Highlander III: The Sorcerer'
er', 'Do, Daniel', 'Doucet, Charles S.', 'Dunn-Hill, John', 'Fierry, Patrick', 'Franc
 'Boy, Gouchy', 'Cantin, Martin', 'Cavalier, Jason', 'Chuipka, Chip', 'Inoue, Akira (
Emidio', 'Neufeld, Martin', 'Okimura, Frederick Y.', 'Oumansky, André', 'Ozores, Robe
tzakis, Georges', 'Vrana, Vlasta', 'Cameron, Morven', 'Larkin, Sheena', 'MacRae, Liz'
barella', 'year': 1968}, {'title': 'Moonraker', 'year': 1979}, {'title': 'Gandahar',
{'_id': ObjectId('657e5adefbe9c89865940b3f'), 'title': 'Abre los ojos', 'genres': ['D
I)', 'Gabella, Ion', 'García, Luis (II)', 'Amenábar, Alejandro', 'Barray, Gérard', 'C
', 'Noriega, Eduardo (II)', 'Otegui, Raúl', 'Palenzuela, Miguel', 'Pepe (III)', 'Prie
 une étrange aventure de Lemmy Caution', 'year': 1965}], 'related_movies': [{'title':
rbourg, Les', 'year': 1964}, {'title': 'Collectionneuse, La', 'year': 1967}, {'title'
e': 'Fanny och Alexander', 'year': 1982}]}
{'_id': ObjectId('657e5adefbe9c89865940b41'), 'title': 'Fifth Element, The', 'genres'
', 'Douglas, Sam (I)', 'Dunwell, Peter', 'Ellwood, Eddie', 'Evans, Lee (I)', 'Ezekiel
Wilson, Stewart', 'Heng, Ivan', 'Holm, Ian', 'Hughes, John (VI)', 'Adamson, Christoph
ewerton, Kevin', 'Brooks, Colin', 'Bryan, Clifton Lloyd', 'Bryce, Roberto', 'Buckley,
n, Robert', 'Creed-Miles, Charlie', 'Culkin, Michael', 'James, Brion (I)', 'Janjua, K
 'Matthews, Al', 'McCrary, Vladimir', 'McDonald, Mac (I)', 'McDougall, Martin', 'McLe
bert', 'Oldman, Gary', 'Paramor, Aron', 'Pellegrino, Vincenzo', 'Perry, Luke (I)', 'P
Tricky (I)', 'Tucker, Chris (I)', 'Tyrell, Tyrone', 'Williams, Fred (VI)', 'Williams,
 Gin', 'De Palma, Laura', 'Frye, Mia', 'Goth, Sophia', 'Graff, Zeta', 'Guillard, Mari
 'Ové, Indra', 'Perez, Josie', 'Richardson, Stina', 'Salvail, Éve', 'Wallace, Julie T
], 'related_movies': [{'title': 'Alphaville, une étrange aventure de Lemmy Caution',
r': 1992}, {'title': 'Cliffhanger', 'year': 1993}, {'title': 'Stargate', 'year': 1994
{'_id': ObjectId('657e5adefbe9c89865940b4d'), 'title': 'Nowhere', 'genres': ['Drama',
Brian', 'Alexander, Peter (III)', 'Bexton, Nathan', 'Blue, Travis William', 'Boyce, A
hyme', 'Light, Kevin', 'Mayweather, Joshua Gibran', 'Nicholas, Petro', 'Phillippe, Ry
te, Christina', "D'Angelo, Beverly", 'Doherty, Shannen', 'Gato, Nicolette', 'Graham,
wan, Rose', 'Odessa, Devon', 'Plumb, Eve', 'Rae, Charlotte', 'Richards, Denise', 'Rob
', 'year': 1965}, {'title': 'Bis ans Ende der Welt', 'year': 1991}, {'title': 'Delica
', 'year': 1937}, {'title': 'Enfants du paradis, Les', 'year': 1945}, {'title': 'Symp
tti di Cabiria, Le', 'year': 1957}, {'title': 'Quatre cents coups, Les', 'year': 1959
```

Podemos observar como los títulos de las consultas coinciden. Para la implementación de la consulta para mongodb hemos seguido la misma lógica que en SQL, aplicando la sintaxis correcta, guardamos el resultado de la query para devolverlo y que un main nos lo muestre por pantalla.

2) Una tabla con toda la información de aquellas películas (documentos) que sean dramas del año 1998, y que empiecen por la palabra “The” (por ejemplo "Governess, The").

```python
def query_b(mongodb_collection):
    query = {
        "genres": "Drama",
        "year": 1998,
        "title": {"$regex": ", The", "$options": "i"}
    }
    result = list(mongodb_collection.find(query))
    return result
```

```sql
-- QUERY B
SELECT imm.*
FROM imdb_movies imm
NATURAL JOIN imdb_moviecountries imc
NATURAL JOIN imdb_moviegenres img
WHERE
    imc.country = 'France'
    AND img.genre = 'Drama'
    AND imm.year = '1998'
    AND imm.movietitle LIKE '%, The%';
```

4 rows returned

	movieid integer	movietitle character varying
1	222714	Land Girls, The (1998)
2	245525	Man in the Iron Mask, The (1998/I)
3	246071	Man with Rain in His Shoes, The (1998)
4	324246	Quarry, The (1998)


```
Consulta B: Dramas que empiezan con 'The' en 1998
{'_id': ObjectId('657e5adefbe9c89865940b5a'), 'title': 'Land Girls, The', 'genres': ['Dra
lie', 'Barr, Russell', 'Bennett, Reverend Alan', 'Bettany, Paul', 'Brown, Arnold', 'Layfi
l', 'Akhurst, Lucy', 'Bannerman, Celia', 'Bell, Ann (I)', 'Friel, Anna', 'Hall, Esther',
related_movies': [{'title': 'Grande illusion, La', 'year': 1937}, {'title': 'Enfants du p
 en ce jardin, La', 'year': 1956}, {'title': 'Notti di Cabiria, Le', 'year': 1957}, {'tit
 Jim', 'year': 1962}], 'related_movies': [{'title': 'À nous la liberté', 'year': 1931}, {
istoires extraordinaires', 'year': 1968}, {'title': 'Planète sauvage, La', 'year': 1973},
a nave va', 'year': 1983}]}
{'_id': ObjectId('657e5adefbe9c89865940b61'), 'title': 'Quarry, The', 'genres': ['Drama']
lcke, Serge-Henri', 'Esau, Sylvia'], 'most_related_movies': [{'title': 'Grande illusion,
a la femme', 'year': 1956}, {'title': 'Mort en ce jardin, La', 'year': 1956}, {'title': '
 5 à 7', 'year': 1961}, {'title': 'Jules et Jim', 'year': 1962}], 'related_movies': [{'ti
: 'Barbarella', 'year': 1968}, {'title': 'Histoires extraordinaires', 'year': 1968}, {'ti
 temps, Les', 'year': 1982}, {'title': 'E la nave va', 'year': 1983}]}
{'_id': ObjectId('657e5adefbe9c89865940b7e'), 'title': 'Man with Rain in His Shoes, The',
Antonio', 'Gold, Max (I)', 'Griggs, Tim', 'Henshall, Douglas', 'Lázaro, Eusebio', 'Meacoc
'Coleman, Charlotte', 'Cruz, Penélope', 'Freud, Emma', 'Headey, Lena', 'Hillier, Emily (I
'year': 1991}, {'title': 'Postino, Il', 'year': 1994}, {'title': 'Rough Magic', 'year': 1
 "She's So Lovely", 'year': 1997}, {'title': 'Train de vie', 'year': 1998}], 'related_mov
{'title': 'Parapluies de Cherbourg, Les', 'year': 1964}, {'title': 'Alphaville, une étran
ntic City', 'year': 1980}, {'title': 'Diva', 'year': 1981}, {'title': 'Tenue de soirée',
{'_id': ObjectId('657e5adefbe9c89865940b7f'), 'title': 'Man in the Iron Mask, The', 'genr
on, Christian', 'Guttierez, Emmanuel', 'Hofland, Michael', 'Atterton, Edward', 'Brissart,
 François', 'Morris, Michael (II)', 'Nemeth, Vincent', 'Patron, Emmanuel', 'Sarsgaard, Pe
er, Laura (II)', 'Godrèche, Judith', 'Parillaud, Anne', 'Varela, Leonor (I)'], 'most_rela
ar': 1979}, {'title': 'Diva', 'year': 1981}, {'title': 'Ran', 'year': 1985}, {'title': 'T
'year': 1993}, {'title': 'Heaven & Earth', 'year': 1993}]}
```


Volvemos a confirmar que el título de las películas que cumplen ambas consultas coinciden, seguimos el mismo procedimiento que en la consulta anterior, para esta consulta hemos tenido que especificar que el título puede contener lo que sea pero si encuentra “, The” y que le siga otros valores, significa que el título de la película empieza por “The”.

3) Una tabla con toda la información de aquellas películas (documentos) en las que Faye Dunaway y Viggo Mortensen hayan compartido reparto.

```python
def query_c(mongodb_collection):
    query = {
        "actors": {"$all": ["Dunaway, Faye", "Mortensen, Viggo"]}
    }
    result = list(mongodb_collection.find(query))
    return result
```

```sql
-- QUERY C
SELECT DISTINCT m.*
FROM imdb_movies m
NATURAL JOIN imdb_moviecountries mc
NATURAL JOIN imdb_actormovies af
NATURAL JOIN imdb_actors a
WHERE
    mc.country = 'France'
    AND m.movieid IN (
        SELECT maux.movieid
        FROM imdb_movies maux
        NATURAL JOIN imdb_actormovies af
        NATURAL JOIN imdb_actors a
        WHERE a.actorname = 'Dunaway, Faye'
    )
    AND m.movieid IN (
        SELECT maux.movieid
        FROM imdb_movies maux
        NATURAL JOIN imdb_actormovies af
        NATURAL JOIN imdb_actors a
        WHERE a.actorname = 'Mortensen, Viggo'
    )
```

```
Consulta C: Películas en las que Faye Dunaway
 y Viggo Mortensen compartieron reparto
{'_id': ObjectId('657e5adefbe9c89865940b1e'),
 'title': 'Albino Alligator', 'genres': ['Cri
```

1 row returned

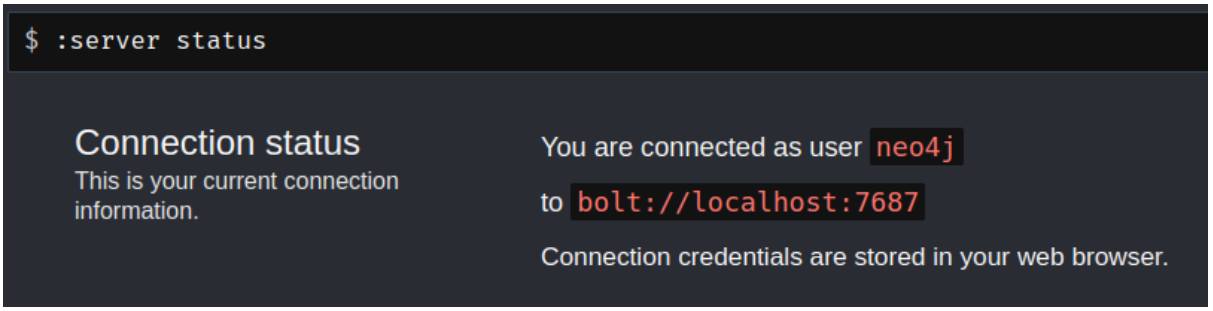
	movieid integer	movietitle character varying
1	12494	Albino Alligator (1996)

El resultado coincide, en este caso vemos las ventajas de la BBDD documental, comparando la diferencia de simplicidad de las consultas.

Para la ejecución de las consultas hemos creado una función principal que organiza la salida del programa así como ejecutar las consultas, este programa se conecta a nuestra BBDD documental y almacena la colección “france” en una variable, además, llama a las consultas que reciben la colección por parámetro. Para probar que las queries funcionan correctamente, como ya hemos podido observar, hemos implementado las consultas equivalentes en nuestra BBDD PostgreSQL y hemos ido comparando las salidas.

1.2 BBDD basadas en Grafos

Primero, tenemos que lanzar el servidor de neo4j con sudo neo4j start, podemos acceder a la interfaz web con el puerto por defecto o confirmar el estado del servidor con sudo neo4j status, en mi caso en la interfaz web:



Para crear la BBDD basada en grafos hemos implementado un script en python usando SQLALchemy y la librería neo4j para python, establecemos conexiones y autenticaciones:

```python
# Configuración de conexión a PostgreSQL
postgres_engine = create_engine("postgresql://alumnodb:1234@localhost/si1", echo=False)
postgres_conn = postgres_engine.connect()

# Configuración de conexión a Neo4j
neo4j_uri = "bolt://localhost:7687"
neo4j_user = "neo4j"
neo4j_password = "si1-password"
```

Ahora, para crear la BBDD con neo4j ejecutamos una consulta sobre nuestra BBDD PostgreSQL para recopilar la información solicitada, esta vez, las 20 películas estadounidenses más vendidas:

```python
query = """
    SELECT
        m.movieid,
        m.movietitle,
        SUM(od.quantity) AS total_quantity
    FROM
        imdb_movies m
    JOIN
        imdb_moviecountries imm ON m.movieid = imm.movieid
    JOIN
        products p ON m.movieid = p.movieid
    JOIN
        orderdetail od ON p.prod_id = od.prod_id
    WHERE
        imm.country = 'USA'
    GROUP BY
        m.movieid, m.movietitle
    ORDER BY
        total_quantity DESC
    LIMIT 20;
"""
movies_result = postgres_conn.execute(text(query))
movies_data = list(movies_result)
```

Esta consulta se aplica sobre la BBDD del ER del apartado anterior.

Tras almacenar la información necesaria, vamos a crear los nodos siguiendo el esquema proporcionado en el enunciado.
Para cada película en la lista devuelta por la query vamos a crear un nodo “Movie” que contendrá los campos movieid y movietitle, estos contienen la información recopilada en la query SQL y se le asignan a un nodo:

```
# Crear nodos y relaciones en Neo4j para cada película
for movie in movies_data:
    movie_id = movie["movieid"]
    movie_title = movie["movietitle"]

    # Crear nodo de película
    neo4j_session.run(
        "MERGE (movie:Movie {id: $id, title: $title})",
        id=movie_id, title=movie_title
```

Cada película tiene sus actores y directores, por tanto, dentro del bucle anterior tenemos que, desde nuestra BBDD PostgreSQL, conseguir las personas implicadas en la película, los actores serán guardados en un nodo y los directores en otro.
Ejecutamos dos queries sql sobre la BBDD original como hemos hecho hasta ahora:

```
# Consulta SQL para obtener actores de cada película
actors_query = """
    SELECT
        ima.actorid,
        ima.actorname
    FROM
        imdb_actormovies imam
    JOIN imdb_actors ima ON imam.actorid = ima.actorid
    WHERE imam.movieid = :movieid;
"""
actors_data = postgres_conn.execute(text(actors_query), movieid=movie_id).fetchall()

# Consulta SQL para obtener directores de cada película
directors_query = """
    SELECT
        imd.directorid,
        imd.directorname
    FROM
        imdb_directormovies imdm
    JOIN imdb_directors imd ON imdm.directorid = imd.directorid
    WHERE imdm.movieid = :movieid;
"""
directors_data = postgres_conn.execute(text(directors_query), movieid=movie_id).fetchall()
```

Habiendo obtenido ids y nombres de actores y directores creamos sus respectivos nodos relacionándolos con la película en cuestión mediante las relaciones neo4j, en este caso, ACTED_IN y DIRECTED, además “rellenamos” los nodos con la información pertinente:

```
# Para cada actor, fusionar el nodo del actor y crear la relación ACTED_IN hacia la película
for actor in actors_data:
    neo4j_session.run(
        "MERGE (actor:Person:Actor {id: $actor_id, name: $actor_name}) "
        "WITH actor "
        "MATCH (movie:Movie {id: $movie_id}) "
        "MERGE (actor)-[:ACTED_IN]->(movie)",
        actor_id=actor['actorid'], actor_name=actor['actorname'], movie_id=movie_id
    )

# Para cada director, fusionar el nodo del director y crear la relación DIRECTED hacia la película
for director in directors_data:
    neo4j_session.run(
        "MERGE (director:Person:Director {id: $director_id, name: $director_name}) "
        "WITH director "
        "MATCH (movie:Movie {id: $movie_id}) "
        "MERGE (director)-[:DIRECTED]->(movie)",
        director_id=director['directorid'], director_name=director['directorname'], movie_id=movie_id
    )
```

Ejemplo BBDD neo4j película Lethal Weapon 4 (desde interfaz web neo4j)

En este caso, el nodo central es el referido a la película, como ya hemos comentado, contiene el título y el id de esta, todos los nodos rojos se unen (apuntan) a Movie mediante la relación ACTED_IN, por otro lado, el nodo naranja, se une (apunta) a Movie con la relación DIRECTED, estos dos últimos tipos de nodo son de tipo Persona y almacenan el id y nombre de los actores/directores.

Consultas

Las consultas para neo4j han sido ejecutadas con “cat nombre.cypher | cypher-shell -u neo4j -p si1-password”

1) Una tabla que devuelva 10 actores ordenados alfabéticamente que no han trabajado con “Winston, Hattie”, pero que en diferentes momentos han trabajado con un tercero en común.

```
MATCH (hattie:Actor {name: "Winston, Hattie"})-[:ACTED_IN]->(m:Movie)
WITH hattie, collect(m) AS hattieMovies
MATCH (m)<-[:ACTED_IN]-(coactor:Actor)
WHERE NOT coactor = hattie
WITH coactor, hattie
MATCH (coactor)-[:ACTED_IN]->(m2:Movie)<-[:ACTED_IN]-(otherActor:Actor)
WHERE NOT (otherActor)-[:ACTED_IN]->()<-[:ACTED_IN]-(hattie)
AND NOT otherActor = hattie
RETURN DISTINCT otherActor.name AS ActorName
ORDER BY ActorName
LIMIT 10;
```

```
-- QUERY 1
SELECT a.actorname
FROM imdb_actors a
JOIN imdb_actormovies am ON a.actorid = am.actorid
JOIN imdb_actormovies am2 ON am.movieid = am2.movieid AND am.actorid != am2.actorid
JOIN imdb_actors a2 ON am2.actorid = a2.actorid
WHERE a2.actorname != 'Winston, Hattie'
AND a2.actorid NOT IN (
    SELECT am3.actorid
    FROM imdb_actormovies am3
    JOIN imdb_actors a3 ON am3.actorid = a3.actorid
    WHERE a3.actorname = 'Winston, Hattie'
)
AND am.movieid IN (
    SELECT m.movieid
    FROM imdb_movies m
    JOIN imdb_moviecountries imm ON m.movieid = imm.movieid
    JOIN products p ON m.movieid = p.movieid
    JOIN orderdetail od ON p.prod_id = od.prod_id
    WHERE imm.country = 'USA'
    GROUP BY m.movieid, m.movietitle
    ORDER BY SUM(od.quantity) DESC
    LIMIT 20
)
GROUP BY a.actorname
HAVING COUNT(DISTINCT am2.actorid) > 1
ORDER BY a.actorname
LIMIT 10;
```

Como vamos a observar en las siguientes imágenes los resultados de las consultas coinciden, apreciamos la gran diferencia de consultas y los beneficios de neo4j.

```
❯ cat winston-hattie-co-c
ActorName
"Adam, Joel"
"Adams, Catlin"
"Adams, Lillian"
"Adams, Melanie (II)"
"Addington, Constance"
"Addota, Kip"
"Ahern, Alston"
"Alcañiz, Luana"
"Alderman, Jane"
"Allen, Sarita"
```

10 rows returned

	actorname character varying
1	Adam, Joel
2	Adams, Catlin
3	Adams, Lillian
4	Adams, Melanie (II)
5	Addington, Constance
6	Addota, Kip
7	Ahern, Alston
8	Albright, Gerald
9	Alcañiz, Luana
10	Alderman, Jane

2) Una tabla que muestre en cada fila pares de personas que han trabajado juntas en más de una película, sin distinguir categoría de actores o directores.

```
MATCH (director:Person {name: "Reiner, Carl"}), (actor:Person {name: "Smyth, Lisa (I)"}),
      path = shortestPath((director)-[:ACTED_IN|DIRECTED*]-(actor))
RETURN path
```

```
-- QUERY 2
WITH TopMovies AS (
    SELECT
        m.movieid
    FROM
        imdb_movies m
    JOIN
        imdb_moviecountries imm ON m.movieid = imm.movieid
    JOIN
        products p ON m.movieid = p.movieid
    JOIN
        orderdetail od ON p.prod_id = od.prod_id
    WHERE
        imm.country = 'USA'
    GROUP BY
        m.movieid
    ORDER BY
        SUM(od.quantity) DESC
    LIMIT 20
)
SELECT
    a1.actorname AS Person1,
    a2.actorname AS Person2,
    COUNT(DISTINCT am1.movieid) AS NumberOfMovies
FROM
    imdb_actormovies am1
JOIN
    imdb_actormovies am2 ON am1.movieid = am2.movieid
    AND am1.actorid != am2.actorid
JOIN
    imdb_actors a1 ON am1.actorid = a1.actorid
JOIN
    imdb_actors a2 ON am2.actorid = a2.actorid
JOIN
    TopMovies tm ON am1.movieid = tm.movieid
WHERE
    a1.actorid < a2.actorid
GROUP BY
    a1.actorname, a2.actorname
HAVING
    COUNT(DISTINCT am1.movieid) > 1
ORDER BY
    NumberOfMovies DESC,
    Person1,
    Person2
```

1 row returned

	person1 character varying	person2 character varying	numberofmovies bigint
1	Walsh, M. Emmet	Macey, Elizabeth	2

```
❯ cat pair-persons-mostoccurrences.cypher| cypher-shell -u n
eo4j -p si1-password
Person1, Person2, sharedMovies
"Walsh, M. Emmet", "Macey, Elizabeth", ["Jerk, The (1979)",
"Bound for Glory (1976)"]
```

Vemos que los resultados vuelven a ser iguales, en esta consulta nos ha extrañado la cantidad de resultados, pero hemos llegado a la conclusión de que al ser 20 películas tiene sentido el resultado. Volvemos a apreciar la gran diferencia de consultas entre sql y neo4j, este último gana muchos puntos a la hora de buscar la intersección entre dos o más campos.

3) Hallar el camino mínimo por el cual el director “Reiner, Carl” podría conocer a la actriz “Smyth, Lisa (I)”.

```
MATCH (director:Person {name: "Reiner, Carl"}), (actor:Person {name: "Smyth, Lisa (I)"}),
      path = shortestPath((director)-[:ACTED_IN|DIRECTED*]-(actor))
RETURN path
```

```
❯ cat degrees-reiner-to-smyth.cypher| cypher-shell -u neo4j -p si1-password
path
(:Person:Actor {name: "Reiner, Carl", id: 536686})-[:ACTED_IN]->(:Movie {title: "Jerk, The (1979)", id: 201944})<-
[:ACTED_IN]-(:Person:Actor {name: "Walsh, M. Emmet", id: 682727})-[:ACTED_IN]->(:Movie {title: "Ordinary People (1
980)", id: 293247})<-[:ACTED_IN]-(:Person:Actor {name: "Smyth, Lisa (I)", id: 1083181})
(:Person:Director {name: "Reiner, Carl", id: 87280})-[:DIRECTED]->(:Movie {title: "Jerk, The (1979)", id: 201944})
<-[:ACTED_IN]-(:Person:Actor {name: "Walsh, M. Emmet", id: 682727})-[:ACTED_IN]->(:Movie {title: "Ordinary People
(1980)", id: 293247})<-[:ACTED_IN]-(:Person:Actor {name: "Smyth, Lisa (I)", id: 1083181})
 ~/Escritorio/SI/P3/app-neo4j-etl  main !2 ?2
 ~/Escritorio/SI/P3/app-neo4j-etl  main !2 ?2
```

Esta consulta se nos ha hecho imposible implementarla en SQL, sin embargo, hemos comprobado los datos a través de la interfaz web de neo4j para asegurar el correcto funcionamiento de la consulta.
En cuanto a su implementación ha sido muy sencilla gracias a la funcionalidad shortestPath() que, literalmente, da el resultado a la consulta solicitada en el enunciado.

2) Uso de tecnología caché de acceso rápido

```python
import redis
import random
from sqlalchemy import create_engine, MetaData, Table, select

# Conexión a la base de datos PostgreSQL
print("Conectando a la base de datos PostgreSQL...")
db_url = 'postgresql://alumnodb:1234@localhost/si1'
engine = create_engine(db_url)
connection = engine.connect()

print("Conexión exitosa.")

metadata = MetaData()

# Cargar la estructura de la tabla desde PostgreSQL
print("Cargando la estructura de la tabla 'customers'...")
customers = Table('customers', metadata, autoload_with=engine, schema='public')
print("Estructura de la tabla cargada correctamente.")

# Consulta para obtener usuarios de España
query = select(customers).where(customers.c.country == 'Spain')

result = connection.execute(query)
users_spain = result.fetchall()

# Cerrar la conexión a PostgreSQL
connection.close()

# Conexión a Redis
redis_db = redis.StrictRedis(host='localhost', port=6379, db=0)

# Iterar sobre los usuarios de España obtenidos de PostgreSQL y almacenar en Redis
for user in users_spain:
    email = user[10]  # Índice de la columna 'email'
    name = f"{user[1]} {user[2]}"  # Índices de 'firstname' y 'lastname'
    phone = user[11]  # Índice de la columna 'phone'
    visits = random.randint(1, 99)  # Generar número aleatorio de visitas

    # Crear el hash en Redis con email, nombre, teléfono y visitas
    data = {'email': email, 'name': name, 'phone': phone, 'visits': visits}
    redis_db.hmset(f'customers:{email}', data)

# Función para incrementar una visita dado el correo electrónico
def increment_by_email(email):
    key = f'customers:{email}'
    db.hincrby(key, 'visits', 1)

# Función para obtener el email del usuario con más visitas
def customer_most_visits():
    keys = redis_db.keys('customers:*')
    max_visits = -1
    email_max_visits = None

    for key in keys:
        visits = redis_db.hget(key, 'visits')
        if visits and int(visits) > max_visits:
            max_visits = int(visits)
            email_max_visits = key.split(':')[-1]  # Obtener el email del key

    return email_max_visits

# Función para mostrar nombre, teléfono y número de visitas dado el email
def get_field_by_email(email):
    key = f'customers:{email}'
    data = redis_db.hmget(key, 'name', 'phone', 'visits')
    if all(data):
        return {
            'name': data[0].decode(),
            'phone': data[1].decode(),
            'visits': int(data[2])
        }
    return None

# Mostrar datos recuperados de Redis
print("Datos recuperados de Redis:")
keys = redis_db.keys('customers:*')
for key in keys:
    data = redis_db.hgetall(key)
    decoded_data = {key.decode(): value.decode() for key, value in data.items()}
    print(decoded_data)
```

En primer lugar realizamos la conexión a PostgreSQL, en concreto usamos SQLAlchemy para conectarnos a nuestra base de datos. Lo primero que hacemos es cargar la tabla ‘customers’ para una posterior consulta sobre aquellos clientes de España. Tras esto, cerramos la conexión con PostgreSQL e iniciamos la conexión a Redis.

Para el siguiente paso, almacenamos los datos obtenidos en Redis iterando sobre los datos obtenidos previamente. Estos se guardan de la forma propuesta en el enunciado (name, phone, visits, email) y se crea un hash en Redis utilizando el email como clave y almacena los datos como valores asociados a esa clave.

Finalizando, se llevan a cabo la implementación de las 3 funciones pedidas en el enunciado:

- **increment_by_email(email):** Incrementa el número de visitas para un usuario dado su correo electrónico en Redis.
- **customer_most_visits():** Encuentra el correo electrónico del usuario con más visitas en Redis.
- **get_field_by_email(email):** Obtiene nombre, teléfono y número de visitas dado el email de un usuario en Redis.

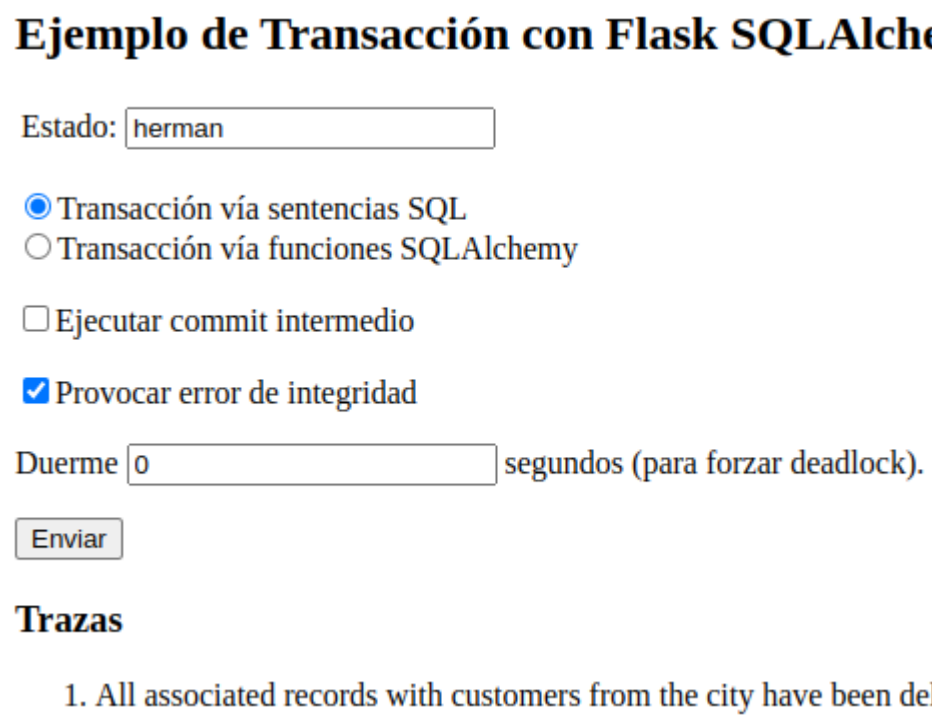
Para confirmar que todo ha ido bien, mostramos los datos almacenados en Redis.

3) Transacciones

A) Aquí tenemos un ejemplo de una ejecución de nuestra función implementada:

4 rows returned

	city character varying
1	mts
2	mts
3	mts
4	mts



Con mts, después volvemos a ejecutar select city from customers where city = 'mts':

0 rows returned

city character varying

```python
# -*- coding: utf-8 -*-

import os
import sys, traceback, time
from sqlalchemy import create_engine, text
from pymongo import MongoClient

# configurar el motor de sqlalchemy
db_engine = create_engine("postgresql://alumnodb:1234@localhost/si1", echo=False, execution_options={"autocommit":False})

# Crea la conexión con MongoDB
mongo_client = MongoClient()

def getMongoCollection(mongoDB_client):
    mongo_db = mongoDB_client.si1
    return mongo_db.topUK

def mongoDBCloseConnect(mongoDB_client):
    mongoDB_client.close()

def dbConnect():
    return db_engine.connect()

def dbCloseConnect(db_conn):
    db_conn.close()

def delState(state, bFallo, bSQL, duerme, bCommit):
    # Array de trazas a mostrar en la página
    dbr=[]

    conn = dbConnect()

    try:
        # Begin transaction
        if bSQL:
            conn.execute("BEGIN")

        # Select customers from the given city
        stmt = text("SELECT customerid FROM customers WHERE city = :city")
        customers = conn.execute(stmt, city=state)
        # If no customers are found, no action is taken
        if not customers:
            dbr.append("No customers found in the specified city.")
            if bSQL:
                conn.execute("COMMIT")
            return dbr

        customer_ids = [c['customerid'] for c in customers]

        # Delete orders and order details
        stmt = text("DELETE FROM orderdetail WHERE orderid IN (SELECT orderid FROM orders WHERE customerid = ANY(:customer_ids))")
        conn.execute(stmt, customer_ids=customer_ids)
        stmt = text("DELETE FROM orders WHERE customerid = ANY(:customer_ids)")
        conn.execute(stmt, customer_ids=customer_ids)

        # If bFallo is True, attempt to delete customers before deleting their orders, which should cause an error due to foreign key constraints
        if bFallo:
            stmt = text("DELETE FROM customers WHERE customerid = ANY(:customer_ids)")
            conn.execute(stmt, customer_ids=customer_ids)
        # Intermediate commit if bCommit is True
        if bCommit:
            conn.execute("COMMIT")
            time.sleep(duerme)  # Sleep to simulate delay or deadlock
            if bSQL:
                conn.execute("BEGIN")

        # Final commit if no error occurred
        if not bFallo:
            conn.execute("COMMIT")
            dbr.append("All associated records with customers from the city have been deleted.")

    except Exception as e:
        # Rollback in case of error
        conn.execute("ROLLBACK")
        dbr.append(f"Transaction failed: {str(e)}. Rollback executed.")

    finally:
        # Close the database connection
        dbCloseConnect(conn)

    return dbr
```

Hemos implementado la función ‘delState’ cuyos argumentos son:

- **state:** Representa la ciudad cuyos registros se eliminarán.
- **bFallo:** Es un booleano que indica si se debe intentar eliminar clientes antes que sus órdenes (para simular un error por restricciones de clave externa).
- **bSQL:** Es un booleano que indica si se debe utilizar SQL explícito (**BEGIN** y **COMMIT**) para transacciones.

- **duerme**: Representa el tiempo de espera en segundos para simular un retraso o un bloqueo.
- **bCommit**: Es un booleano que indica si se debe hacer un commit intermedio durante el proceso.

A partir de aquí, se inicializa una lista vacía ‘dbr’ que se usa para almacenar trazas o mensajes de estados. Para empezar, realizamos la conexión a la db mediante ‘dbConnect()’ y usamos la variable ‘conn’ para referirnos a esta. Si bSQL es verdadero, se inicia una transacción utilizando ‘conn.execute("BEGIN")’. Tras esto, se ejecuta una consulta SQL para obtener los IDs de clientes que pertenecen a la ciudad especificada como argumento y se eliminan el carrito y los pedidos correspondientes a los clientes encontrados en el paso anterior.

Si bFallo es verdadero, se intenta eliminar los clientes antes que sus pedidos, lo que podría generar un error por restricciones de clave externa.

Si bCommit es verdadero, se realiza un commit intermedio en la transacción y se introduce un tiempo de espera para simular un retraso o un posible bloqueo. Si no hay errores, se realiza un commit final para confirmar los cambios en la base de datos.

En caso de que se produzca algún error durante el proceso, se ejecuta un rollback para deshacer cualquier cambio realizado en la base de datos y se registra el error en dbr.

Finalmente, se devuelve la lista dbr, que contiene mensajes sobre el estado de la transacción (éxito, fallo, errores, etc.).

B)

b.

```
alter table customers add promo integer;
```

Se crea el script ‘updPromo.sql’ y para crear la columna promo en la tabla customers usamos esa línea de código.

c.

```
CREATE OR REPLACE FUNCTION apply_promo()
RETURNS TRIGGER AS $$
BEGIN
    UPDATE orders
    SET totalamount = totalamount - (totalamount * NEW.promo / 100)
    FROM customers
    WHERE customers.customerid = NEW.customerid
    AND orders.customerid = NEW.customerid
    AND orders.status IS NULL;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER updPromo
AFTER UPDATE OR INSERT ON customers
FOR EACH ROW
EXECUTE FUNCTION apply_promo();
```

Cuando se inserta o actualiza una fila en la tabla customers, el trigger ‘updPromo’ se activa y ejecuta la función ‘apply_promo’, que a su vez actualiza los campos ‘totalamount’ en la tabla orders basándose en el descuento especificado en la columna promo de la tabla customers.

d.

```
CREATE OR REPLACE FUNCTION apply_promo()
RETURNS TRIGGER AS $$
BEGIN
    UPDATE orders
    SET totalamount = totalamount - (totalamount * NEW.promo / 100)
    FROM customers
    WHERE customers.customerid = NEW.customerid
    AND orders.customerid = NEW.customerid
    AND orders.status IS NULL;

    RETURN NEW;
    perform pg_sleep(20);
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER updPromo
AFTER UPDATE OR INSERT ON customers
FOR EACH ROW
EXECUTE FUNCTION apply_promo();
```

Insertamos la sentencia ‘perform pg_sleep()’ en el lugar adecuado de la creación de nuestro trigger.

LOS APARTADOS SIGUIENTES NO SE HAN COMPRENDIDO Y, POR LO TANTO, NO SE HAN REALIZADO.

Explicación deadlocks y posibilidad de evitarlos:
Esto se produce cuando realizamos una transacción de borrado debido a que se intenta borrar datos contenidos en la tabla ‘customers’ pero, encontramos esta tabla bloqueada por el trigger previamente creado, ya que se actualiza. Para evitar estos deadlocks, podemos reproducir el tiempo de ejecución de las transacciones y hacer una optimización mediante índices.