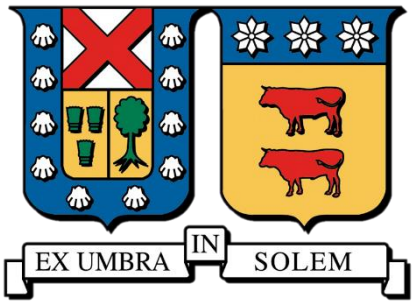

Tarea 2: Neo4j

Integrantes/Rol/Paralelo:

Benjamín Nicolas Daza Jiménez/202173574-7/201

Sebastián Andrés Von Kunowsky Lepe/202173560-7/201

Curso: Bases de datos avanzadas

Profesor: José Luis Martí Lara

Ayudante: Daniela Sánchez Nizza

En el archivo “Tarea2.py” se encuentra la implementación de las respuestas (querys) de cada una de las preguntas, las cuales están en funciones de la clase “PonyDatabase”.

1.

```
def add_pony(self):
    nombre = input("Ingresa el nombre del pony: ")
    color = input("Ingresa el color del pony: ")
    tipo = input("Ingresa el tipo del pony: ")
    habilidad = input("Ingresa la habilidad del pony: ")
    cutiemark = input("Ingresa la cutiemark del pony: ")
    gusto = input("Ingresa los gustos del pony: ")
    bebida = input("Ingresa la bebida del pony: ")
    nodo = nombre.lower().replace(" ", "")
    query = '''
        MERGE ("'''+nodo+'''":Pony { nombre: '"'''+nombre+'''"', color: '"'''+color+'''"', tipo: '"'''+tipo+'''"', habilidad:
        MERGE ("'''+nodo+'''")-[:AMIGOS]->(vinyl)
    '''
    self.run_query(query)
```

En la función de clase “add_pony”, se le pregunta por consola al usuario los atributos del pony a insertar; la variable “nodo” contiene el nombre del nodo en Neo4j; en la query se hacen dos MERGE, el primero añadirá el nuevo pony y el segundo creará la relación de AMIGOS solicitada; la última línea ejecuta la query en la base de datos.

2.

```
def cant_ponys(self):
    ciudad = input("Ingrese el nombre de la ciudad: ")
    query = '''
        MATCH ((pony)-[:VIVE_EN]->(ciudad:Ciudad{nombre:"'''+ciudad+'''"}))
        RETURN COUNT(pony) AS total_ponys
    '''
    result = self.return_query(query)
    print(result[0]['total_ponys'] if result else 0)
```

En la función de clase “cant_ponys”, se le pregunta por consola al usuario de que ciudad desea ver la cantidad de ponys; la query contiene un MATCH donde se busca todos los ponys que viven en el nodo ciudad solicitado, luego se retorna la cantidad de ponys encontrados; en la variable “result” se guarda el resultado de ejecutar la query; luego se muestra en pantalla la cantidad encontrada, en caso contrario se muestra 0.

```
def return_query(self, query, parameters = None):
    with self.driver.session(database=self.database_name) as session:
        result = session.run(query, parameters)
        return [record for record in result]
```

Función de clase ”return query” es similar a ”run_query” solo que esta retorna el valor obtenido de ejecutar la query en la base de datos.

3.

```
def add_anexo(self):
    query = '''
        MATCH (p:Pony)
        OPTIONAL MATCH (p)-[:AMIGOS]->(a:Pony)
        WITH p, COUNT(a) AS total_amigos
        SET p.anexo = CASE
            WHEN p.tipo = "Unicornio" AND total_amigos >= 3 THEN "Sociable"
            WHEN p.tipo = "Unicornio" AND total_amigos = 2 THEN "Reservado"
            WHEN p.tipo = "Unicornio" AND total_amigos = 1 THEN "Solitario"
            WHEN p.tipo = "Pony terrestre" AND total_amigos >= 4 THEN "Hipersociable"
            WHEN p.tipo = "Pony terrestre" AND total_amigos <= 2 THEN "Reservado"
            WHEN p.tipo = "Alicornio" THEN "Realeza"
            ELSE "Por Completar"
        END
    '''
    self.run_query(query)
```

La función de clase ”add_anexo” la query contiene un MATCH para encontrar a todos los ponys, la siguiente línea es un OPTIONAL MATCH para encontrar además todos los ponys con amigos, luego se realiza un WITH para guardar los ponys y su cantidad total de cada pony, luego se realiza un SET para agregar a cada pony el atributo anexo con valor CASE, en el cual se comparan y se asigna el valor pedido (esto utilizando WHEN y THEN), en caso de que no se cumpla ninguna condición se le asignará el valor de ELSE (”Por Completar”).

4.

```
def camino_corto(self):
    pony1 = input("Ingrese el nombre del primer pony: ")
    pony2 = input("Ingrese el nombre del segundo pony: ")
    query = '''
        MATCH path = shortestPath((pony1:Pony{nombre:"'''+pony1+'''"})-[:AMIGOS*]->(pony2:Pony{nombre:"'''+pony2+'''"}))
        RETURN nodes(path) AS path, length(path) AS length
    '''
    result = self.return_query(query)
    if result:
        short_path = (result[0]['path'] if result else None)
        length = result[0]['length']
        print(f"Longitud del camino: {length}")
        print("Nodos en el camino:")
        for nodo in short_path:
            print(f"{nodo['nombre']} ({nodo['tipo']})")
    else:
        print("No existe relacion")
```

En la función de clase ”camino_corto”, se le pregunta al usuario por medio de consola los nombres de los ponys de los cuales desea obtener el camino; en la query se realiza el MATCH y la función ”shortestPath” para encontrar el camino más corto entre los nodos, luego se retorna los nodos del camino encontrado y la longitud de este. Luego en la variable ”result” se guarda el resultado de ejecutar la query; luego si existe un valor en ”result” se guardarán en variables y se mostrarán por consola cada nodo que conforma el camino encontrado y su longitud.

5.

```
def friend_of_friend(self):
    nombre = input("Ingrese nombre del pony: ")
    query = '''
        MATCH ((pony:Pony{nombre:"'''+nombre+'''"})-[:AMIGOS]->(amigo:Pony)-[:AMIGOS]->(amigo_de_amigo:Pony))
        WHERE amigo_de_amigo <> amigo AND amigo_de_amigo <> pony
        RETURN DISTINCT amigo_de_amigo.nombre AS amigo_de_amigo
    '''
    result = self.return_query(query)
    if result:
        print("Amigos de amigos:")
        for record in result:
            print(f"{record['amigo_de_amigo']}")
    else:
        print("No se encontraron amigos de amigos.")
```

La función de clase “friend_of_friend”, solicita al usuario por medio de consola el nombre del pony al que desea encontrar los amigos de sus amigos; en la query se realiza un MATCH para encontrar los amigos de los amigos del pony solicitado, luego se realiza un WHERE para solo obtener los amigos de amigos que no sean el pony solicitado ni ninguno de sus amigos directos, posteriormente se retornan todos los distintos nombres de los amigos de amigos encontrados del pony (se utiliza DISTINCT). Luego en la variable “result” se guarda el resultado de ejecutar la query, si existe un valor en “result” se mostrará por consola el nombre de cada amigo de amigo de los ponys.

6.

```
def magic_ponys(self):
    query = '''
        MATCH (pony:Pony)
        WHERE toLower(pony.habilidad) CONTAINS 'magia'
        RETURN pony.nombre AS pony
    '''
    result = self.return_query(query)
    if result:
        print("Lista de ponys con habilidades relacionadas a la magia:")
        for record in result:
            print(f"{record['pony']}")
    else:
        print("No se encontraron ponys.")
```

En la función de clase “magic_pony”, se guarda la query que realiza un MATCH para encontrar todos los ponys, luego se usa la función “toLower” para obtener la habilidad de cada pony en minúsculas para hacer un WHERE y un CONTAINS ‘magia’ lo que filtrará todos los ponys que tengan habilidades con dicha palabra, en la siguiente linea se retorna el nombre de cada pony solicitado. Luego en la variable “result” se guarda el resultado de ejecutar la query, si existe un valor en “result” se mostrará por consola el nombre de cada pony que tenga la palabra ‘magia’ en su atributo habilidad.

7.

```python
def friend_uni(self):
    query = '''
    MATCH ((pony:Pony)-[:AMIGOS]->(amigo:Pony))
    WHERE NOT (amigo)-[:AMIGOS]->(pony)
    RETURN pony.nombre AS pony_amigo, amigo.nombre AS amigo_no_amigo
    '''
    result = self.return_query(query)
    if result:
        for record in result:
            print(f"{record['pony_amigo']} es amigo unidireccional de {record['amigo_no_amigo']}")
    else:
        print("No se encontraron ponys con amigos unidereccionales.")
```

En la función de clase “friend_uni”, se guarda la query que realiza un MATCH para filtrar los ponys que tienen amigos, luego se realiza un WHERE NOT para volver a filtrar, dando como resultado los ponys que tienen un amigo unidireccional, en la siguiente línea retorna el nombre del pony y su amigo unidireccional. Luego en la variable “result” se guarda el resultado de ejecutar la query, si existe un valor en “result” se mostrará por consola el nombre de cada pony y de su amigo unidireccional.

8.

```python
def ponys_pref(self):
    tipo = input("Ingresar el tipo de pony: ")
    query = '''
        MATCH (pony:Pony {tipo:"'''+tipo+'''"})
        WHERE pony.bebida IN ['Coca Cola', 'Sprite']
        RETURN pony.bebida AS bebida, COUNT(pony) AS cantidad
    '''
    result = self.return_query(query)
    suma = 0
    if result:
        print("Preferencias de bebida para ponis del tipo: "+ tipo)
        for record in result:
            bebida = record['bebida']
            cantidad = record['cantidad']
            suma += cantidad
            print(f"Bebida: {bebida}, Cantidad: {cantidad}")
        print(f"Cantidad total: {suma}")
    else:
        print("No se encontraron ponys.")
```

La función “ponys_pref”, solicita al usuario ingresar por consola el tipo de pony a los que desea contar las preferencias de bebidas; en la query se realiza un MATCH para encontrar todos los ponys del tipo seleccionado, luego se realiza un WHERE para filtrar a los ponys que prefieren la bebida Coca Cola o Sprite, para posteriormente retornar la cantidad de ponys que prefieren una bebida y los que prefieren la otra. Luego en la variable “result” se guarda el resultado de ejecutar la query, si existe un valor en “result” se guardarán en variables la cantidad de ponys que prefieren Coca Cola, en otra la cantidad que prefieren Sprite y en otra la suma de ambas, posteriormente se mostrarn por consola estas cantidades.

9.

```python
def more_enemys(self):
    query = '''
        MATCH (pony:Pony)
        OPTIONAL MATCH (pony)-[:ENEMIGOS]->(enemigo:Pony)
        WITH pony, COUNT(enemigo) AS enemigos
        OPTIONAL MATCH (pony)-[:COLABORACION]->(colaborador:Pony)
        WITH pony, enemigos, COUNT(colaborador) AS colaboradores
        WHERE enemigos > colaboradores
        RETURN pony.nombre AS pony
    '''
    result = self.return_query(query)
    if result:
        print("Ponys que tienen mas enemigos que colaboraciones:")
        for record in result:
            print(f"{record['pony']}")
    else:
        print("No se encontraron ponys.")
```

En la función de clase “more_enemys”, se guarda la query que realiza un MATCH para encontrar todos los ponys, luego un OPTIONAL MATCH para encontrar todos los ponys con enemigos, posteriormente se hace un WITH para guardar cada pony y la cantidad de enemigos de este, luego se realiza otro OPTIONAL MATCH para encontrar los ponys con colaboradores, se realiza otro WITH para guardar cada pony y la cantidad de colaboradores de este, luego se hace el WHERE para filtrar los ponys con más enemigos que colaboradores y se retorna el nombre de cada uno. Luego en la variable “result” se guarda el resultado de ejecutar la query, si existe un valor en “result” se mostrará por consola el nombre de cada pony que cumple con la condición dada.

10.

```python
def pref_and_friend(self):
    query = '''
        MATCH ((pony:Pony {bebida:"Coca Cola"})-[:AMIGOS]->(amigo:Pony{tipo:"Pony terrestre", bebida:"Sprite"}))
        RETURN DISTINCT pony.nombre as pony
    '''
    result = self.return_query(query)
    if result:
        print("Ponys que prefieren Coca Cola y tienen al menos un amigo que prefiere Sprite:")
        for record in result:
            print(f"{record['pony']}")
    else:
        print("No se encontraron ponys.")
```

En la función “pref_and_friend”, se guarda una query donde se hace un MATCH para filtrar los ponys con preferencia de Coca Cola y que tenga un amigo con preferencia de Sprite,

luego se retorna los nombres de estos ponys. Luego en la variable “result” se guarda el resultado de ejecutar la query, si existe un valor en “result” se mostrará por consola el nombre de cada pony que cumple con la condición dada.

11. Los índices mejoran el rendimiento de las consultas de lectura, esto se debe a que permite encontrar rápidamente los nodos y relaciones que siguen una condición específica. Mientras que, al no tener índices, sería muy tardío las consultas de lectura, especialmente en bases de datos grandes al tener que escanearla completamente para obtener el nodo o la relación.

Por otro lado, con respecto al costo de mantenimiento, el mantener índices actualizados a la hora de hacer operaciones de escritura consume tiempo y recursos, ya que, en cada una de las escrituras, el índice debe actualizarse. Por ende, mientras más índices se tiene, más recursos se utilizarán para mantenerlos consistentes.

Para mantener un equilibrio entre rendimiento de consultas y el costo de equilibrio se puede hacer uso de índices selectos, esto refiriéndose a propiedades que se consulten frecuentemente. Sin embargo, es importante saber si el uso de la aplicación está enfocado a escritura o lectura, ya que se podría permitir mayor o menor uso de índices según estas necesidades. Finalmente, hacer uso de EXPLAIN o PROFILE permite tener mejor entendimiento en cómo se están gastando los recursos para así tomar una mejor decisión.

12. EXPLAIN es adecuado usarlo cuando solo se quiere ver el plan de ejecución sin ejecutar la consulta, ahorrando tiempo y recursos de estas consultas. Por otro lado, PROFILE es preferible cuando se necesita información detallada respecto al rendimiento de la consulta, ya que necesita ejecutarla con el fin de obtener detalles de estadísticas de ejecución.

PROFILE sería preferible sobre EXPLAIN en situaciones donde se ha optimizado una consulta y se quiere medir el impacto de esta en términos de recursos y tiempo, ya que es ideal para análisis más detallado. También es preferible usar PROFILE cuando se modifica una consulta y se quiere verificar cómo afecta la implementación en el tiempo de ejecución en tiempo real.