Esse arquivo é uma pequena introdução ao trabalho que foi desenvolvido para o EP2 da disciplina.

Geração dos dados:

1. Uma lista de cerca de 43 mil nomes foi baixada da internet e armazenada em ‘name.json’.

2. O script ‘cria_pessoas.py’ gera um arquivo ‘person.txt’ com o subconjunto de nomes a serem efetivamente utilizados no banco. Esse arquivo pode ser facilmente modificado para aumentar o número de pessoas do banco.

   Para os experimentos aqui expostos, utilizamos um banco de 2 mil pessoas.

3. O script Python ‘adiciona_pessoas.py’ foi utilizado para gerar o arquivo DML da tabela ‘PERSON’ de pessoas do banco.

4. O script Python ‘adiciona_relacao.py’ foi utilizado para gerar o arquivo DML de relações entre as pessoas do banco, expresso na tabela ‘PERSON_FRIEND’.

   Para os experimentos aqui descritos, ‘Alice’ e ‘Bob’ têm 1000 amigos - amigos entre si e amigos das pessoas de índice 3 até 1001. Toda relação de ‘Alice’ e ‘Bob’ é recíproca, o que significa que, para todos os seus amigos, eles também são amigos de volta. O resultado prático disso, é que tanto ‘Alice’ quanto ‘Bob’ estão diretamente envolvidos em 2 mil registros da tabela ‘PERSON_FRIEND’, cumprindo as exigências do enunciado.

   Para criar relações mais profundas - de pelo menos 5 níveis de indireção como descrito no EP -, para cada pessoa que não ‘Alice’ e ‘Bob’, conectamos 5 pessoas, mas sempre pulando 5 para que as amizades não se repitam. O exemplo abaixo esclarece a abordagem.

ID 3	<- amigo de -> ID 4		ID 4	<- amigo de -> ID 9
	<- amigo de -> ID 5			<- amigo de -> ID 10
	<- amigo de -> ID 6	. . .		<- amigo de -> ID 11
	<- amigo de -> ID 7			<- amigo de -> ID 12
	<- amigo de -> ID 8			<- amigo de -> ID 13

   Vale notar, ainda, que a mesma estratégia de amizade sempre recíproca foi utilizada. Além disso, no exemplo acima, foram ocultadas as amizades com ‘Alice’ e ‘Bob’ e também a amizade com ID 3 na tabela do ID 4.

A última tabela de conexões gerada como a tabela do exemplo acima foi para o ID 402, tabela essa incompleta uma vez que essa pessoa relaciona-se apenas com ID 1999.

No total foram geradas 7990 relações. E é fácil perceber que existem relações com nível de indireção maior ou igual a 5.

5. Para rodar os 3 scripts Python automaticamente, basta rodar o script bash 'gera_dmls.sh'.

Implementação relacional:

1. Para implementar as tabelas, o PostgreSQL foi utilizado. As tabelas são bem descritas através do arquivo 'DDL.sql' e a confirmação de geração das mesmas é dada pelos *screenshots* abaixo.

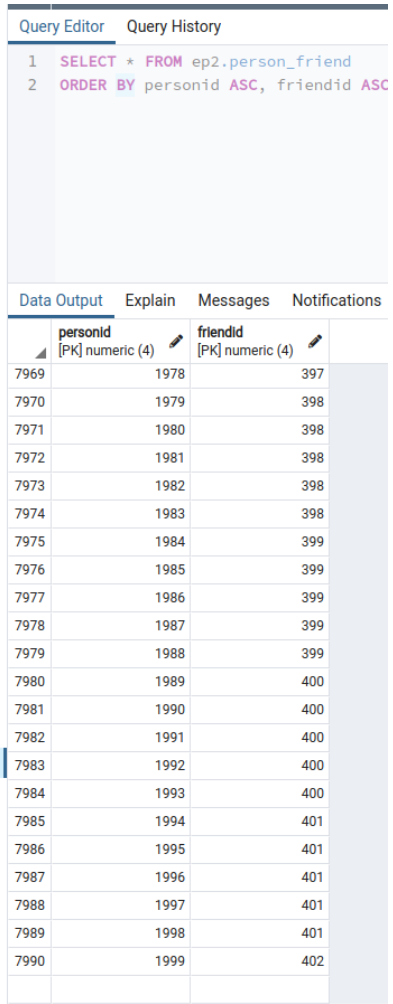

Figura 1: Imagens mostrando o final das tabelas geradas para o EP. Através das duas, fica fácil observar a estrutura das relações e pode-se comprovar a criação do grande número de entradas.

2. As consultas são expressas em Python através do script ‘consulta.py’ e destacadas aqui no bloco de código a seguir.

```
SELECT PERSON
FROM ep2.PERSON
INNER JOIN (SELECT FRIENDID
             FROM ep2.PERSON_FRIEND
             INNER JOIN (SELECT ID
                         FROM ep2.PERSON
                         WHERE PERSON='Bob') AS ID_BOB
              ON ep2.PERSON_FRIEND.PERSONID=ID_BOB.ID) AS ID_AMIGOS_BOB
ON ep2.PERSON.ID=ID_AMIGOS_BOB.FRIENDID;
```

Consulta 2.1 em SQL

```
SELECT PERSON
FROM ep2.PERSON
INNER JOIN (SELECT PERSONID
             FROM ep2.PERSON_FRIEND
             INNER JOIN (SELECT ID
                         FROM ep2.PERSON
                         WHERE PERSON='Bob') AS ID_BOB
              ON ep2.PERSON_FRIEND.FRIENDID=ID_BOB.ID) AS ID_BOB_AMIGO
ON ep2.PERSON.ID=ID_BOB_AMIGO.PERSONID;
```

Consulta 2.2 em SQL

```
SELECT PERSON
FROM ep2.PERSON
INNER JOIN (SELECT DISTINCT ep2.PERSON_FRIEND.FRIENDID
             FROM ep2.PERSON_FRIEND
             INNER JOIN (SELECT FRIENDID
                         FROM ep2.PERSON_FRIEND
                         INNER JOIN (SELECT ID
                                     FROM ep2.PERSON
                                     WHERE PERSON='Alice') AS ID_ALICE
                         ON ep2.PERSON_FRIEND.PERSONID=ID_ALICE.ID) AS
ID_AMIGOS_ALICE
              ON ep2.PERSON_FRIEND.PERSONID=ID_AMIGOS_ALICE.FRIENDID) AS
ID_AMIGOS_AMIGOS_ALICE
ON ep2.PERSON.ID=ID_AMIGOS_AMIGOS_ALICE.FRIENDID;
```

Consulta 2.3 em SQL

Alguns experimentos relacionados a tempo de execução foram realizados nesse script, mas o valor efetivamente utilizado para a comparação pedida no quinto item do

enunciado foi o tempo de consulta obtido no pgAdmin 4. Essa escolha foi feita para deixar a comparação mais justa, uma vez que a implementação do banco de dados orientado a grafo foi feita num *sandbox* online do Neo4j e o tempo foi computado na própria plataforma. Caso a consulta para o banco de dados orientado a grafo fosse feita em Python, o consumo de tempo computado contava o tempo das requisições HTTPS.

3. Para o banco de dados relacional, a consulta 2.3 foi feita 10 vezes, resultando numa média de cerca de 120 milissegundos. Um *screenshot* do campo do pgAdmin 4 utilizado para contagem é exibido abaixo.

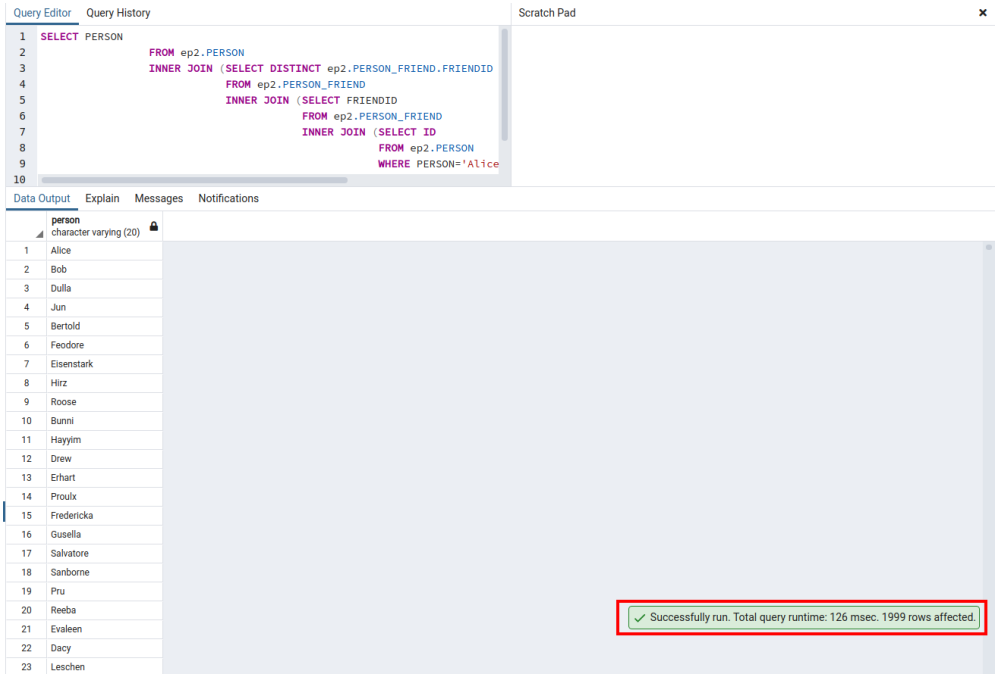

Figura 2: Imagem ilustrativa da consulta 2.3. É possível ver uma fração da tabela de resposta e, destacado no retângulo em vermelho, o tempo necessário para realizar a consulta.

Implementação em grafos:

1. Para fazer a implementação do banco em grafo, foi utilizada a ferramenta '*Blank Sandbox*' do Neo4j. Tendo um servidor implantado, utilizei Python para ler os dados do banco relacional previamente construído e gerar o grafo desejado com Cypher, o que pode ser visto no script 'gera_grafo.py'. Abaixo, encontram-se figuras comprovando a existência do banco , descrevendo o modelo do grafo pensado e, por fim, exemplificando uma consulta de 'amigos da pessoa de ID 3' para mostrar a correta implementação do modelo de grafo pensado. Nessa consulta, os nós resposta - e suas respectivas relações - foram pedidos, não apenas os nomes, isso para melhor ilustrar a estrutura pensada para o banco.

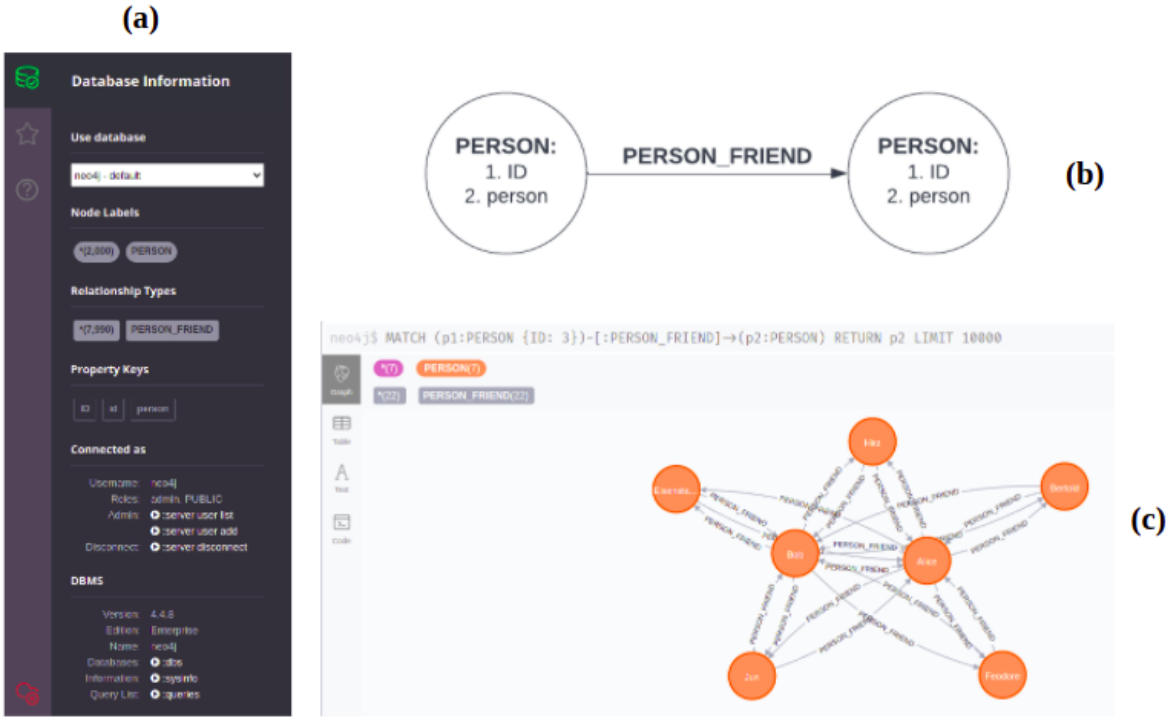

Figura 3: Imagem subdividida em três diferentes ilustrações. Em (a), pode-se observar a efetiva criação do banco, comprovada pelos 2 mil nós 'PERSON' e 7990 relações 'PERSON_FRIEND' identificados. Já em (b), fica clara a estrutura de grafo pensada para bem representar os dados. Em (c), por fim, uma consulta em Cypher foi feita para melhor ilustrar a forma do grafo.

2. Replicando a consulta 2.3 agora em Cypher, temos uma *query* bastante mais simples.

```
MATCH (p1:PERSON {person: 'Alice'})-[:PERSON_FRIEND*2]->(p2:PERSON) RETURN
p2.person LIMIT 10000
```

Consulta 2.3 em Cypher

Fazendo o mesmo experimento das 10 consultas, temos uma redução significativa do tempo de *query* com a implementação do banco em grafo, demorando cerca 30 milissegundos na média. Esse resultado já era esperado e decorre diretamente da exponencialmente maior velocidade da busca em grafo quando comparada com execução de operações 'JOIN' aninhadas. Um *screenshot* do campo do Neo4j utilizado para contagem é exibido abaixo.

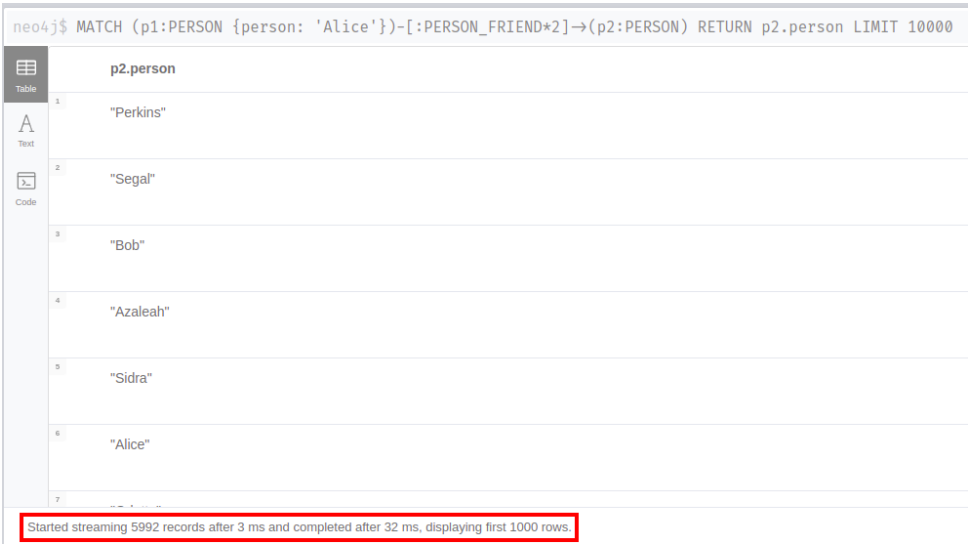

Figura 4: Ilustração da consulta 2.3 em Cypher com sua saída e, destacado em vermelho, o tempo de resposta.

Vale notar alguns detalhes sobre a implementação em Neo4j. Em primeiro lugar, o tempo de resposta das consultas é calculado usando o segundo *timestamp* menos o primeiro *timestamp*. Na figura 4, por exemplo, o tempo é dado por 32 - 3 = 29 milissegundos.

Para além disso, percebe-se que o número de registros gerados é bastante maior do que o esperado. Esse fenômeno ocorre porque a implementação do banco em grafo não filtra saídas repetidas por padrão. Para gerar esse conjunto, portanto, deve-se substituir ‘RETURN p2.person’ por ‘RETURN collect(distinct p2.person)’. Essa operação, contudo, é muito custosa em Neo4j e, sendo uma forma de pós-processamento da consulta, não foi utilizada quando da medição de tempo da *query*.