ALUNO: FLÁVIO AUGUSTO ALÓ TORRES – 2020030477

BANCO DE DADOS 2: TRABALHO PRÁTICO 1

Primeira parte: desenvolver uma aplicação que consome dados de uma API de Dados

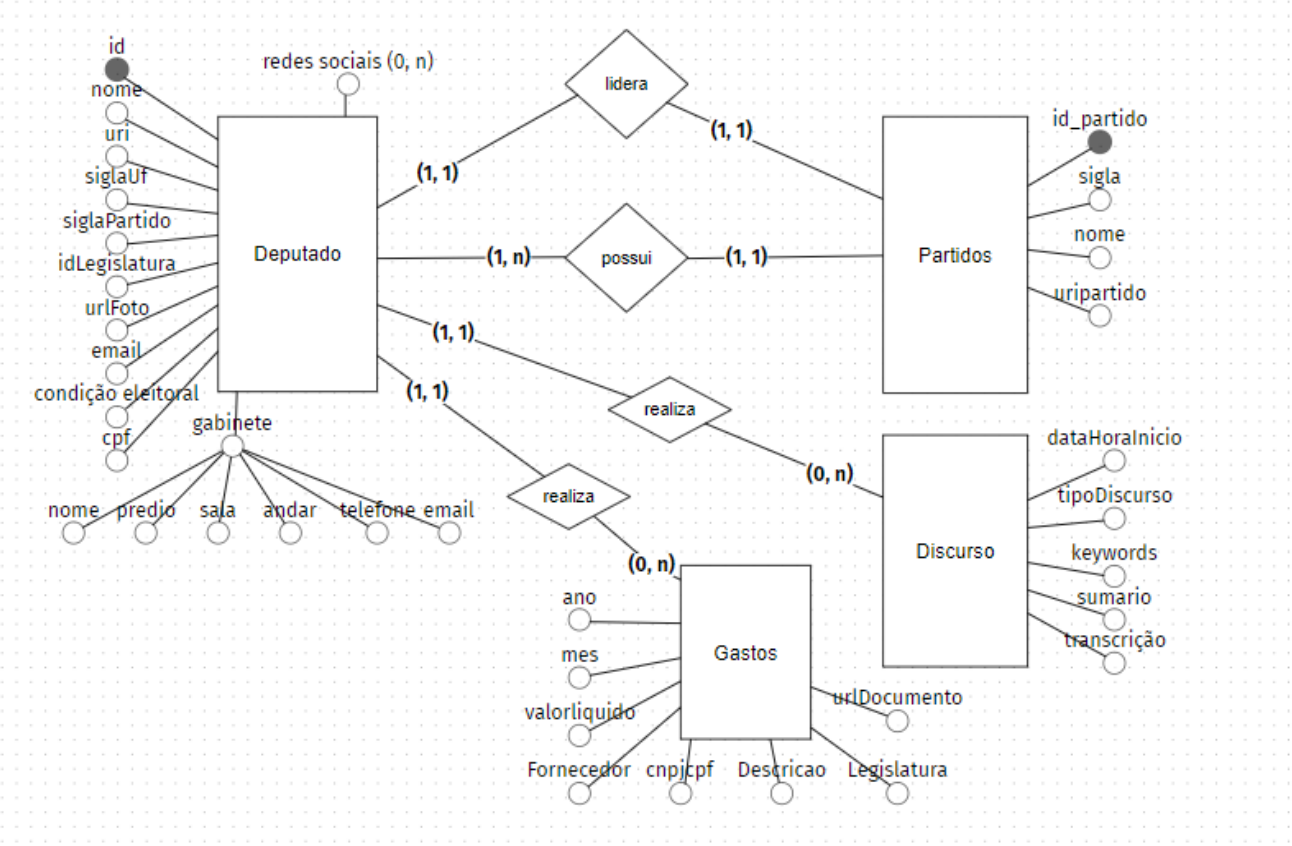
Para o desenvolvimento deste trabalho, eu escolhi consumir dados de uma API da Câmara dos Deputados, do governo. Para obter mais dados sobre a API RESTful utilizada, o link é: https://dadosabertos.camara.leg.br/swagger/api.html#api. Antes de eu começar a utilizar a API mencionada, busquei estudar uma outra API de dados governamentais (de compras e licitações federais), porém conforme fui tentando desenvolver, fiquei confuso em relação a alguns relacionamentos e com a semântica de dados, além das irregularidades e demora em algumas solicitações. Este acontecimento me fez migrar para uma API mais simples.

O Sistema Gerenciador de Bancos de Dados utilizado foi o PostgreSQL. Como os dados abertos ao público, o funcionamento das requisições da API utilizada não necessitam do uso de uma chave individual. Para obter-se os dados, as requisições são feitas via HTTP GET, onde podemos forneces os parâmetros necessários para a busca através da URL.

O primeiro passo do desenvolvimento foi pegar os dados gerais referentes aos atuais deputados através de uma requisição. Depois de tratar e inserir esses dados básicos na tabela deputados do banco de dados, o próximo passo foi pegar as informações adicionais dos deputados (ainda terminando de preencher a tabela deputado). Isso é possível através de requisições na API utilizando como parâmetro de busca o id de cada deputado. Vale ressaltar que um novo tipo de dado foi criado dentro do atributo gabinete, para poder guardar as informações de onde é possível encontrar o deputado.

O passo seguinte foi a criação de uma nova tabela para guardar o conteúdo multivalorado do atributo das redes sociais do deputado (ver o modelo entidade relacionamento abaixo). Além disso, um deputado pode realizar discursos, que são guardados na tabela discurso. A API fornecia alguns dados adicionais, mas após observar os dados, percebi que muitos atributos acabavam sendo um pouco irrelevantes ou nem estavam preenchidos na maioria dos casos. Esses atributos não foram utilizados.

O modelo entidade relacionamento (MER) concebido foi:

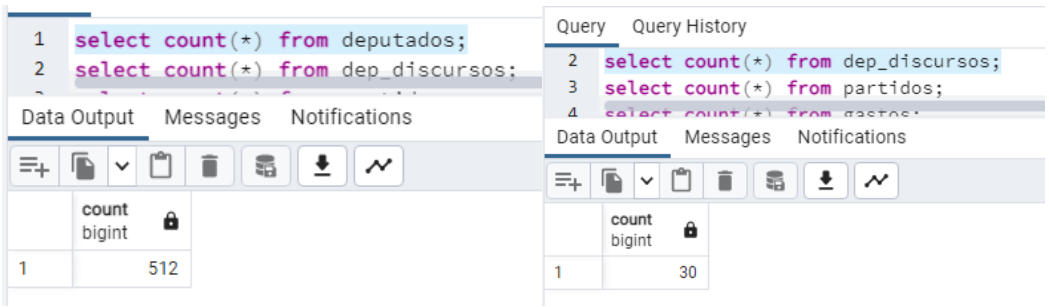


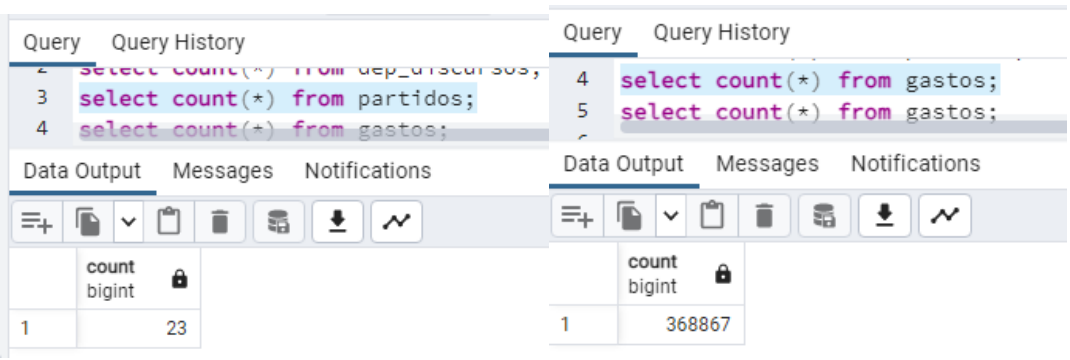
Na questão de permissões, foram criados dois tipos de usuário:

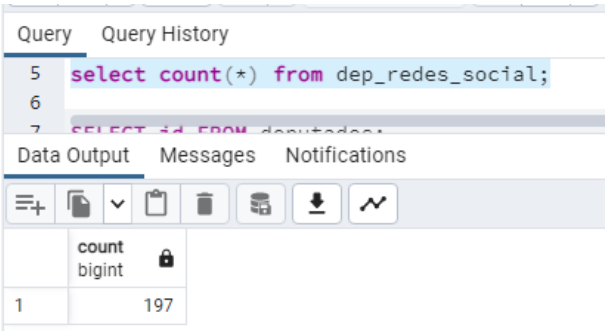
- Um usuário padrão, que tem acesso a ler todos os dados do banco
- Um usuário administrador, com permissão de inserir dados no banco (usuário e senha estão no arquivo SQL)

Criei apenas um índice na tabela deputados, no atributo nome. Pensei em criar um índice na tabela de gastos, porém percebo que essa tabela está sujeita a muitas ações DML. As consultas estão retornando em um tempo imperceptível.

O print com o count das tabelas pode ser observado abaixo:







O arquivo SQL para a criação do banco se encontra na pasta zipada, junto podem ser encontrados um backup do banco de dados, além de uma pasta com dois arquivos python do código do programa.

O vídeo do programa funcionando pode ser visto no link: [Trabalho Banco de Dados 2 - consumindo dados da API da câmara]

O github do trabalho pode ser encontrado aqui: [https://github.com/flavio055063/Api-dados-deputados-app-bd.git]

Infelizmente, ainda é possível que ocorra algumas falhas no código python na hora de inserir os dados consultados da API (mas creio que já resolveram o problema). Não sei o que aconteceu, mas, até alguns dias atrás, a URL de requisição de detalhes sobre os deputados parou de funcionar, por causa de um erro interno no servidor da API.

Tentando inserir os valores dos gastos de cada deputado, meu computador i5 de 4° geração com 16 GB de RAM DDR3 já levou mais de 80 minutos, conseguindo inserir apenas 53813. Isso dá uma média de 673 tuplas por minuto de funcionamento.

Para que as inserções ocorressem mais rápido, tive que adaptar a parte do código que insere os gastos na tabela, abrindo apenas uma conexão, inserindo tudo e por fim encerrando a conexão. Nesse método, todas as tuplas foram inseridas em menos de 5 minutos.

Para fazer o código funcionar, execute o *main.py* dentro da pasta Modelo MVC.

Testes de performance:

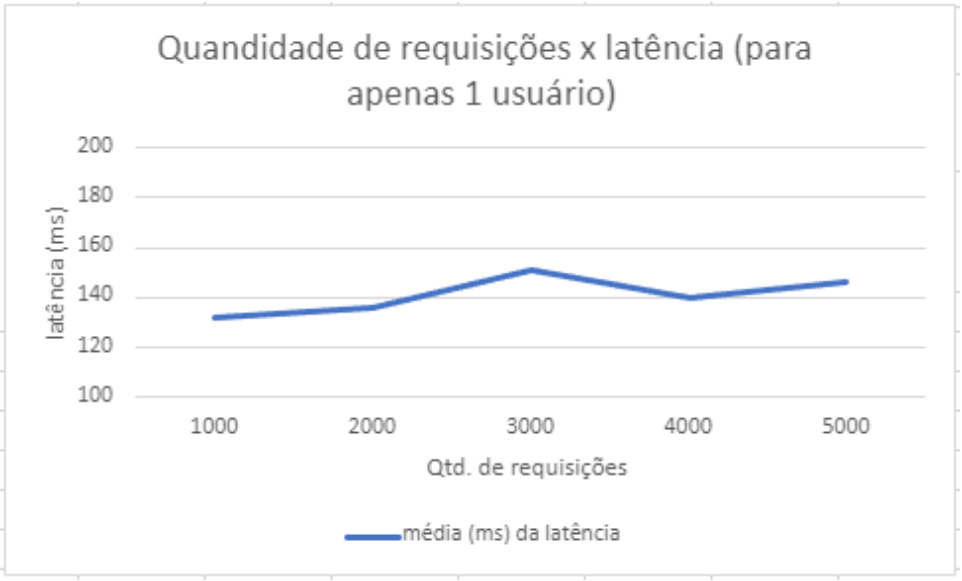
Para fazer o teste, o número máximo de conexões do PostgreSQL foi definido em 6000. Como previsto no documento, há dois principais testes que são realizados.

	max_connections text
1	6000

1- **Mantendo a quantidade de usuários (threads) como 1 e aumentando a quantidade de requisições até o teste retornar erro:**

Após definir apenas um usuário no JMeter É perceptível que alterar a quantidade de requisições para o número fixo de apenas 1 usuário não provoca aumento de latência significativo (permanece quase constante).

Quandidade de requisições x latência (para apenas 1 usuário)	
Qtd de requisições	Média (ms) da latência
1000	132
2000	136
3000	151
4000	140
5000	146



2- Definindo uma quantidade de requisições fixa e aumentando a quantidade de usuários até o teste retornar erro:

Quandidade de usuários x latência (para apenas 10 requisições por usuário)	
Qtd de usuários	Média (ms) da latência
25	629
50	1188
75	1651
100	2283
125	2676
150	3045
175	3323
200	3705
225	4418
250	5291
300	6068
400	7493
500	9062
600	10575

Com 600 usuários algumas tentativas retornaram erro, então esse número foi considerado como o limite de conexões estáveis. Percebi também que no JMeter, a criação de threads (usuários) estava ficando um pouco lenta e demorava para chegar no máximo definido. Enquanto isso acontecia, algumas das iterações das threads criadas já realizavam a consulta no banco de dados.

O computador utilizado para os testes foi:

- Processador: Processador Intel® Core™ i5-4460, cache de 6M, até 3,40 GHz, 4° geração (Haswell), com 4 núcleos e 4 threads.
- Sistema operacional: Windows 10 (64 bits)
- Memória (RAM): 16GB (DDR3)
- GPU: NVIDIA GeForce GTX 970 4GB