Instituto de Educação Superior de Brasília

Gilliard Reis Fernandes

Título: Pipeline de dados referente aos gastos dos deputados no ano de 2022.

Brasília

2025

Gilliard Reis Fernandes

Título: Pipeline de dados referente aos gastos dos deputados no ano de 2022.

Projeto Integrador de Big Data II apresentado ao Instituto de Educação Superior de Brasília, como parte dos requisitos para a obtenção do título de Tecnólogo em Banco de Dados e Armazenamento de Big Data.

Orientador: Francisco Lopes de Caldas Filho

Brasília

2025

RESUMO

O presente estudo tem como objetivo desenvolver um pipeline de dados eficiente para coletar, transformar e analisar as despesas públicas dos deputados federais no ano de 2022. A metodologia adotada segue o modelo ETL (Extract, Transform, Load), no qual os dados são coletados, limpos e armazenados em um banco de dados estruturados (SQLite3) para posterior análise. Como resultado, foi possível identificar padrões de gastos, comparações entre parlamentares e/ou partidos, e despesas mais recorrentes. Concluímos que o uso de pipelines de dados pode facilitar a análise/gestão de grandes volumes de informações e possibilitando insights sobre a alocação de verbas públicas no legislativo brasileiro.

Palavras-chave: Pipeline de dados; Despesas públicas; ETL; Padrões de gastos; Insights.

SUMÁRIO

1. INTRODUÇÃO

Este estudo tem como objetivo desenvolver e implementar um pipeline de dados voltado para a coleta, a transformação e a análise dos gasstos dos deputados federais no ano de 2022. Utilizando o modelo ETL (Extract, Transform, Load), este trabalho pretende automatizar a obtenção de informações a partir de fontes públicas - API, aplicar técnicas de limpeza e estruturação dos dados e disponibilizar visualizações interativas que permitam uma melhor compreensão dos padrões de despesas parlamentares.

A justificativa para este estudo se baseia na crescente demanda por soluções tecnológicas que facilitem a análise transparente dos dados governamentais, promovendo maior controle social e acesso à informação de maneira simplificada. A implementação de pipelines de dados na área de gestão pública pode contribuir significativamente para auditorias automatizadas, otimização de processos de fiscalização e identificação de inconsistências nos gastos legislativos.

Com isso, espera-se que esta pesquisa possa incentivar o desenvolvimento de novas soluções baseadas em automação e análise de dados, auxiliando na fiscalização do uso de recursos públicos pelos parlamentares.

2. DESENVOLVIMENTO

2.1 MODELAGEM DO BANCO DE DADOS

A modelagem do banco de dados foi estruturada em quatro tabelas principais:

deputados_56_detalhes - Armazena detalhes pessoais de cada deputado.

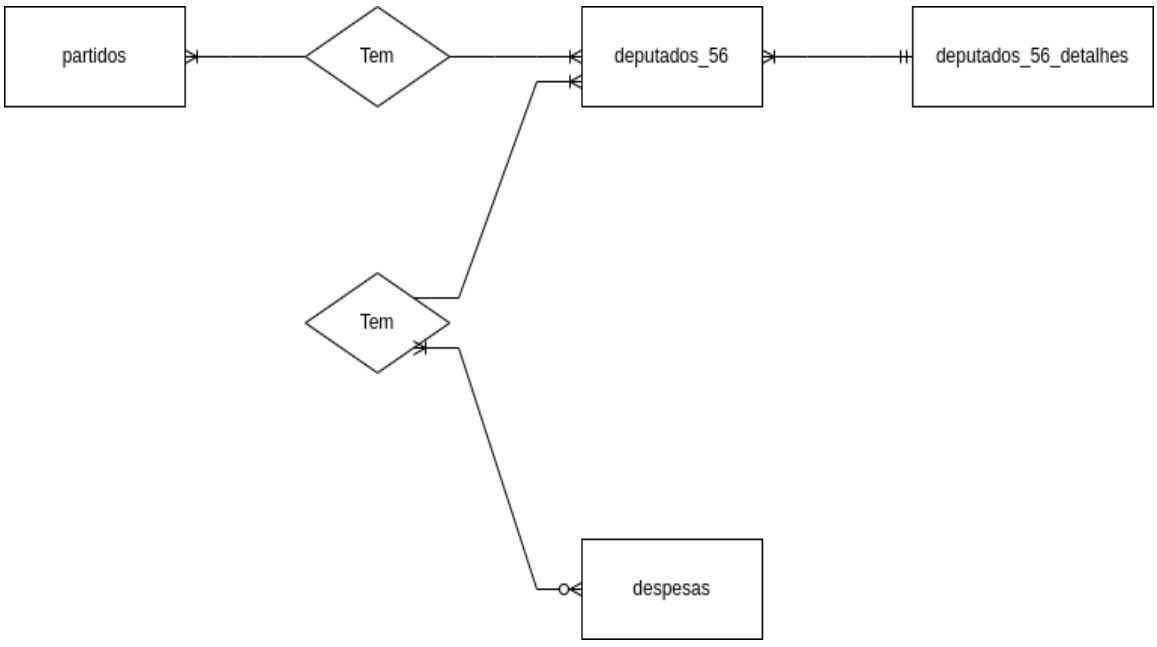
partidos - Contém informações sobre os partidos políticos.

deputados_56 - Relaciona os deputados com suas siglas partidárias e estados.

despesas - Registra as despesas parlamentares de cada deputado.

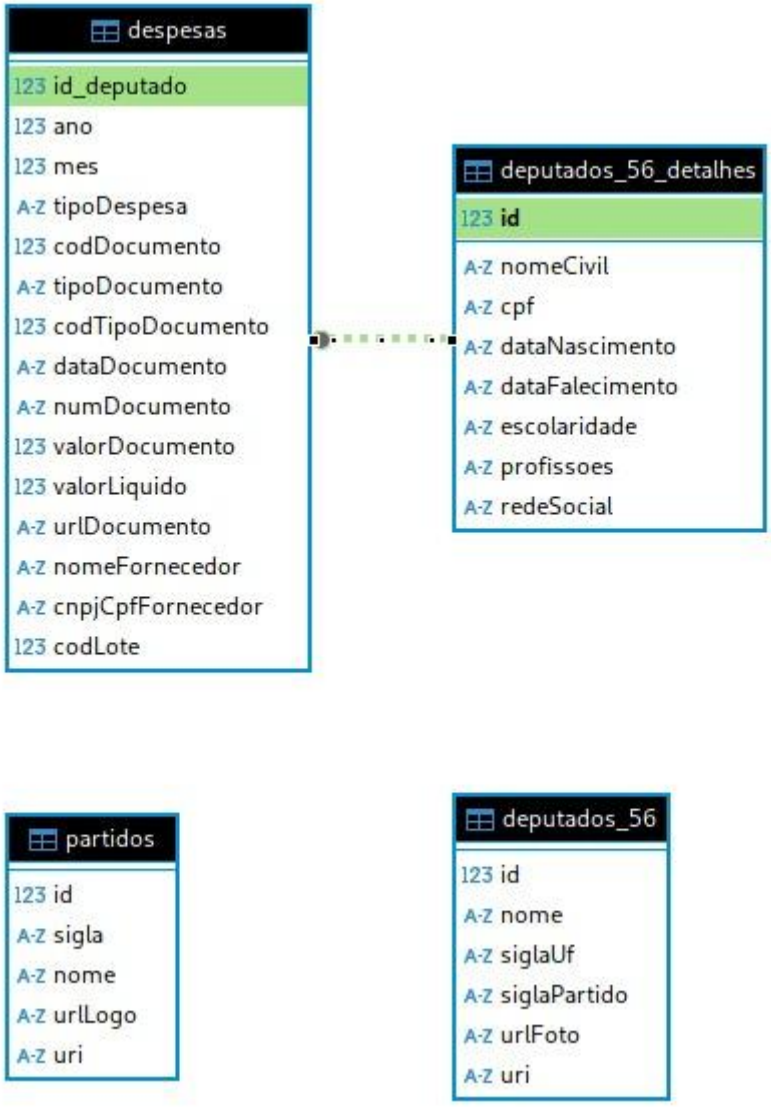
2.2 MODELO CONCEITUAL

O modelo conceitual do banco de dados representa a estrutura de informações relacionadas a partidos políticos, deputados da 56ª legislatura e suas respectivas despesas, organizadas em entidades conectadas por relações. Partidos tem uma relação N:N deputados56 por um ponto de decisão ("Tem"), indicando que um partido pode ter vários deputados. Outra decisão similar conecta deputados56 a deputados56detalhes, porém posso ter vários deputados em deputados_56 devido mudança de partido, que armazena informações detalhadas sobre cada deputado em deputados_56_detalhes (não se repete), e a despesas, que registra seus gastos onde vários deputados pode ter nenhuma ou várias despesas. O modelo sugere um fluxo hierárquico de dados, onde partidos agregam deputados, e estes, por sua vez, possuem tanto detalhes quanto registros financeiros.



2.3 MODELO LOGÍCO

As tabelas partidos e deputados_56 não possuem relações diretas com outras tabelas, pois são utilizadas no processo inicial de coleta de dados via requisições GET da API. Após essa etapa, os dados dos deputados são armazenados detalhadamente na tabela deputados_56_detalhes, que se torna a principal referência para vinculação das despesas. A tabela despesas possui uma chave estrangeira (id_deputado) referenciando deputados_56_detalhes, configurada com ON DELETE CASCADE, garantindo que todas as despesas associadas a um deputado sejam automaticamente removidas caso ele seja excluído. Assim, o modelo lógico mantém os dados organizados e estruturados, permitindo um armazenamento eficiente e seguro das informações coletadas pela API.



2.4 MODELO FÍSICO

Este código em Python utiliza SQLite3 para criar um banco de dados que armazena informações sobre deputados, partidos e despesas. Ele ativa as chaves estrangeiras para garantir consistência referencial, cria tabelas para detalhes dos deputados, partidos, deputados ativos e suas despesas, e estabelece relações entre elas com "ON DELETE CASCADE" para manter a integridade dos dados. Após executar o script SQL para criar as tabelas, o código salva as alterações e fecha a conexão com o banco de dados.

```
conn.execute("PRAGMA foreign_keys = ON;")
cursor.executescript(
  """
  CREATE TABLE IF NOT EXISTS deputados_56_detalhes (
    id INTEGER NOT NULL PRIMARY KEY,
    nomeCivil TEXT NOT NULL,
    cpf TEXT NOT NULL,
    dataNascimento TEXT,
    dataFalecimento TEXT,
    escolaridade TEXT,
    profissoes TEXT,
    redeSocial TEXT -- Salvar lista como uma str separada por ;
  );
  CREATE INDEX IF NOT EXISTS idx_deputados_56_detalhes_nomeCivil ON deputados_56_detalhes(id);

  CREATE TABLE IF NOT EXISTS partidos (
    id INTEGER NOT NULL PRIMARY KEY,
    sigla TEXT NOT NULL UNIQUE,
    nome TEXT NOT NULL,
    urlLogo TEXT,
    uri TEXT NOT NULL
  );
```

```
    CREATE TABLE IF NOT EXISTS deputados_56 (
        id INTEGER NOT NULL PRIMARY KEY,
        nome TEXT NOT NULL,
        siglaUf TEXT NOT NULL,
        siglaPartido TEXT NOT NULL,
        urlFoto TEXT NOT NULL,
        uri TEXT NOT NULL,
        FOREIGN KEY (id) REFERENCES deputados_56_detalhes(id) ON DELETE
CASCADE
    );
    CREATE TABLE IF NOT EXISTS despesas (
        id_deputado INTEGER NOT NULL,
        ano INTEGER NOT NULL,
        mes INTEGER NOT NULL,
        tipoDespesa TEXT NOT NULL,
        codDocumento INTEGER NOT NULL,
        tipoDocumento TEXT NOT NULL,
        codTipoDocumento INTEGER NOT NULL,
        dataDocumento TEXT NOT NULL,
        numDocumento TEXT NOT NULL,
        valorDocumento REAL NOT NULL,
        valorLiquido REAL NOT NULL,
        urlDocumento TEXT,
        nomeFornecedor TEXT NOT NULL,
        cnpjCpfFornecedor TEXT NOT NULL,
        codLote INTEGER NOT NULL,
        FOREIGN KEY (id_deputado) REFERENCES deputados_56_detalhes(id) ON DELETE
CASCADE
    );
    CREATE INDEX IF NOT EXISTS idx_despesas_deputados ON despesas(numDocumento);
    """)
conn.commit()
conn.close()
```

2.5 IMPORTAÇÃO DAS BIBLIOTECAS NECESSÁRIAS

Bibliotecas essenciais para o processamento de dados e interação com o banco de dados SQLite: pandas é usado para manipulação de dados, enquanto requests permite realizar requisições Web, como obtenção de informações da API. A biblioteca sqlite3 possibilita a criação e gerenciamento do banco de dados SQLite, e a bibliteca os fornece funções para manipulação de arquivos e diretórios, e time permite controle do tempo, como inserção de pausas para evitar sobrecarga em requisições. Essas bibliotecas juntas facilitam a coleta, armazenamento e análise de dados de forma automatizada.

```
import pandas as pd
import request
import sqlite3
import os
import time
```

2.6 CONECTANDO COM O BANCO DE DADOS

Código responsável por estabelece uma conexão com o banco de dados SQLite localizado na pasta datasets e caso não exista ele cria o arquivo .db, garantindo que o caminho seja absoluto para evitar problemas de diretório. Primeiro, ele obtém o caminho completo do arquivo data.db usando os.path.abspath(), depois abre ou cria o banco de dados com sqlite3.connect(db_path). Em seguida, cria um cursor (conn.cursor()), que será usado para executar comandos SQL, como criação de tabelas, inserção e consulta de dados.

```
db_path = os.path.abspath("../datasets/data.db")
conn = sqlite3.connect(db_path)
cursor = conn.cursor()
```

2.7 PIPELINE DE CONSULTA E ARMAZENAMENTO DE DADOS

A seguir temos como será o funcionamento do armazenamento de dados para cada tabela que foi criada.

2.7.1 Tabela deputados_56

Este código em Python realiza uma interação com a API da Câmara dos Deputados para obter informações detalhadas sobre os deputados ativos no período de 01/01/2022 a 31/12/2022. Inicialmente, define-se a URL base da API e os parâmetros da requisição, que incluem a data de início e fim, a ordem ascendente dos resultados e o critério de ordenação por sigla do estado (siglaUF).

A requisição é feita utilizando a biblioteca requests, que envia uma solicitação GET à API com os parâmetros definidos. Caso a resposta da API seja bem-sucedida (response.raise_for_status()), os dados retornados em formato JSON são convertidos em um DataFrame do Pandas. Este DataFrame é então filtrado para incluir apenas as colunas de interesse: id, nome, sigla do estado (siglaUf), sigla do partido (siglaPartido), URL da foto (urlFoto) e URI.

O próximo passo envolve a persistência desses dados em um banco de dados SQLite. Utilizando um gerenciador de contexto (with sqlite3.connect(db_path) as conn), o DataFrame é salvo na tabela "deputados_56", substituindo quaisquer dados previamente existentes (if_exists="replace"). Após a inserção, a conexão é feito o commit para garantir que as alterações sejam salvas permanentemente. O código também imprime uma mensagem confirmando a operação e inclui um atraso de meio segundo (time.sleep(0.5)) para evitar sobrecarga na API.

Além disso, o código verifica se a resposta da API contém um link para a próxima página de resultados. Se um link com a relação "next" for encontrado, sua URL é armazenada para possíveis requisições subsequentes, permitindo a paginação dos dados. Em caso de qualquer exceção durante a requisição, uma mensagem de erro específica é capturada e exibida, garantindo que problemas de conectividade ou outras falhas sejam reportados de forma clara.

```
base_url = https://dadosabertos.camara.leg.br/api/v2/deputados
params = {
   "dataInicio": "2022-01-01",
   "dataFim": "2022-12-31",
   "ordem": "ASC",
   "ordenarPor": "siglaUF"}
```

```
try:
    response = requests.get(base_url, params=params)
    response.raise_for_status()
    data = response.json()

    if "dados" in data and data["dados"]:
        deputados = pd.DataFrame(data["dados"])
        deputados = deputados[["id", "nome", "siglaUf", "siglaPartido", "urlFoto", "uri"]]

        with sqlite3.connect(db_path) as conn:
            deputados.to_sql("deputados_56", conn, if_exists="replace", index=False)
            conn.commit()

            print(f"Salvos dados do deputado {deputados['nome']}")
            time.sleep(0.5)

    else:
        print("Nenhum dado encontrado.")

    url = None
    if "links" in data:
        for link in data["links"]:
            if link.get("rel") == "next":
                url = link.get("href")
                break

except requests.RequestException as e:
    print(f"Erro na requisição: {e}")
```

2.7.2 Tabela deputados_56_detalhes

O código em python a seguir consulta detalhes sobre cada deputado e obtendo informações sobre a profissão a partir da URI armazenada na tabela. Inicialmente, define-se uma função convert_rede_social que converte listas de redes sociais em uma string separada por ponto e vírgula, facilitando o armazenamento no banco de dados.

A execução principal do código ocorre dentro de um bloco try, onde uma conexão com o banco de dados SQLite é estabelecida (with sqlite3.connect(db_path) as conn). Utilizando um cursor, o código executa uma consulta para obter os IDs e URIs dos deputados armazenados na tabela deputados_56. Esses URIs são utilizados para fazer requisições GET à API da Câmara dos Deputados, buscando informações detalhadas de cada deputado.

Para cada URI, se a resposta da API for bem-sucedida (response.raise_for_status()), os dados JSON retornados são processados. O código verifica se a resposta contém a chave "dados" e, caso positivo, extrai as informações detalhadas do deputado (nome civil, CPF, data de nascimento, data de falecimento, escolaridade, profissão e redes sociais). A função convert_rede_social é utilizada para garantir que as redes sociais sejam armazenadas corretamente como uma string.

Antes de inserir os dados no banco de dados, o código verifica se o deputado já está presente na tabela deputados_56_detalhes (cursor.execute("SELECT id FROM deputados_56_detalhes WHERE id = ?", (id,))). Se o deputado não estiver presente (if not result), os detalhes são inseridos na tabela utilizando uma instrução SQL INSERT.

Após cada inserção, é feito o commit (conn.commit()) para garantir que as alterações sejam salvas permanentemente, e um atraso de meio segundo (time.sleep(0.5)) é introduzido para evitar sobrecarga na API. Uma mensagem de confirmação é impressa para cada deputado inserido com sucesso.

O código também inclui tratamento de exceções para capturar e exibir erros de requisição (requests.RequestException) e erros relacionados ao banco de dados (sqlite3.Error).

```
try:
    with sqlite3.connect(db_path) as conn:
        cursor = conn.cursor()
        cursor.execute("SELECT id, uri FROM deputados_56")
        uris = cursor.fetchall()

        for id, uri in uris:
            response = requests.get(uri)
            response.raise_for_status()
            data = response.json()

            if "dados" in data and data["dados"]:
                dados = data["dados"]
                detalhes = {
                    "id": id,
                    "nomeCivil": dados.get("nomeCivil", ""),
                    "cpf": dados.get("cpf", ""),
                    "dataNascimento": dados.get("dataNascimento", ""),
                    "dataFalecimento": dados.get("dataFalecimento", ""),
                    "escolaridade": dados.get("escolaridade", ""),
                    "profissoes": dados.get("profissoes", ""),
                    "redeSocial": convert_rede_social(dados.get("redeSocial", ""))
                }

                # Verificando se o deputado já está na tabela
                cursor.execute("SELECT id FROM deputados_56_detalhes WHERE id = ?", (id,))
                result = cursor.fetchone()

                if not result:
                    cursor.execute(
                        """
                        INSERT INTO deputados_56_detalhes
                        (id, nomeCivil, cpf, dataNascimento, dataFalecimento, escolaridade, profissoes,
redeSocial)
```

```
                VALUES (?, ?, ?, ?, ?, ?, ?, ?)
                """,
                (
                    detalhes["id"],
                    detalhes["nomeCivil"],
                    detalhes["cpf"],
                    detalhes["dataNascimento"],
                    detalhes["dataFalecimento"],
                    detalhes["escolaridade"],
                    detalhes["profissoes"],
                    detalhes["redeSocial"]
                )
            )
            cursor.execute("CREATE INDEX IF NOT EXISTS idx_deputados_56_detalhes_id ON deputados_56_detalhes(id)")
            conn.commit()

        time.sleep(0.5)
        print(f"Descrição do Deputado {detalhes['id']} inserido com sucesso")

    else:
        print("Nenhum dado encontrado para a URI:", uri)
except requests.RequestException as e:
    print(f"Erro na requisição: {e}")

except sqlite3.Error as e:
    print(f"Erro no banco de dados: {e}")
```

2.7.2.1 Inserindo a profissão de cada Deputado

O código em Python tem como objetivo atualizar a tabela deputados_56_detalhes no banco de dados com as profissões dos deputados obtidas pela API da Câmara dos Deputados. Percorrendo uma lista de identificadores dos deputados, faz requisições HTTP para obter os dados e os insere no banco.

A execução ocorre dentro de um bloco with, garantindo o fechamento correto da conexão com o banco. Para cada deputado, a requisição é feita a uma URL formatada dinamicamente e se a resposta for bem-sucedida, o código verifica se há informações na chave "dados". As profissões são extraídas e organizadas pela função convert_profissoes(), transformadas em uma string única separada por ponto e vírgula e inseridas na coluna profissoes através do comando SQL UPDATE.

Para evitar sobrecarga na API, há um intervalo de 0,5 segundos entre as requisições. Além disso, mensagens de confirmação informam quais dados foram atualizados. O código também inclui um tratamento de exceções para capturar erros de conexão e garantir que falhas nas requisições não interrompam a execução do programa.

```
def convert_profissoes(profissoes):
    if isinstance(profissoes, list):
        # Garante que só strings válidas sejam utilizadas e ignora valores `None`
        return "; ".join(
            prof["titulo"] for prof in profissoes if "titulo" in prof and isinstance(prof["titulo"], str)
        )
    return ""

with sqlite3.connect(db_path) as conn:
    cursor = conn.cursor()

    for id, uri in uris:
        try:
            response = requests.get(
                f"https://dadosabertos.camara.leg.br/api/v2/deputados/{id}/profissoes"
            )
            response.raise_for_status()
            data = response.json()

            if "dados" in data and data["dados"]:
                lista_profissoes = data["dados"]
                profissao = convert_profissoes(lista_profissoes)
```

```
            cursor.execute(
                "UPDATE deputados_56_detalhes SET profissoes = ? WHERE id = ?",
                (profissao, id)
            )
            conn.commit()

            time.sleep(0.5)
            print(f"Profissões inseridas com sucesso para ID {id}: {profissao}")
        else:
            print(f"Nenhuma profissão encontrada para ID {id}")

    except requests.RequestException as e:
        print(f"Erro na requisição para ID {id}: {e}")
```

2.7.3 Tabela partidos

O código Python realiza a coleta de informações sobre partidos políticos ativos no período de 01/01/2022 a 31/12/2022 utilizando a API da Câmara dos Deputados e armazena esses dados em um banco de dados SQLite. Inicialmente, a URL da API é definida para buscar partidos dentro do período especificado, organizados em ordem crescente pela sigla. Um loop while garante a paginação dos dados seguido pelo loop for padrão, o motivo seria que a url não precisa de parâmetros adicionais pois já recebe a string para realizar todas as consultas limitada ao número de partidos referente ao ano de 2022. A cada iteração, os dados dos partidos são extraídos no formato JSON e armazenados em uma lista, incluindo informações como id, sigla, nome, URL do logotipo e URI.

Caso os dados tenham sido coletados com sucesso, um DataFrame do Pandas é criado para facilitar a manipulação e análise. Em seguida, utilizando um gerenciador de contexto (with sqlite3.connect(db_path) as conn), esse DataFrame é salvo na tabela "partidos" do banco de dados SQLite, substituindo eventuais dados anteriores. Em caso de falha na requisição HTTP, um erro é impresso e a execução é interrompida. Ao final do processo, uma mensagem confirma se os dados foram armazenados com sucesso ou se a coleta falhou.

```
url = (
    "https://dadosabertos.camara.leg.br/api/v2/partidos?"
    "dataInicio=2022-01-01&"
    "dataFim=2022-12-31&"
    "ordem=ASC&"
    "ordenarPor=sigla"
)

todos_partidos = []

while url:
    response = requests.get(url)

    if response.status_code == 200:
        data = response.json()

        for partido in data["dados"]:
            partido_info = {
                "id": partido.get("id", ""),
                "sigla": partido.get("sigla", ""),
                "nome": partido.get("nome", ""),
                "urlLogo": partido.get("urlLogo", ""),
                "uri": partido.get("uri", "")
            }
            todos_partidos.append(partido_info)
        url = None
        for link in data["links"]:
            if link.get("rel") == "next":
                url = link.get("href")
                break

    else:
        print(f"Erro na requisição ({response.status_code})")
        break
```

```
if todos_partidos:
    df_final = pd.DataFrame(todos_partidos)

    with sqlite3.connect(db_path) as conn:
        df_final.to_sql("partidos", conn, if_exists="replace", index=False)

    print("✅ Dados dos partidos armazenados com sucesso no banco de dados!")
else:
    print("❌ Nenhum dado foi coletado.")
```

2.7.3.1 Inserindo da url do logo de cada partido eleitoral

O código Pythona seguir realiza a atualização das imagens de logotipo dos partidos políticos armazenados no banco de dados SQLite com base nos dados obtidos da API da Câmara dos Deputados. Inicialmente, uma conexão com o banco de dados é estabelecida e um cursor é criado para permitir a execução de comandos SQL. Em seguida, é feita uma consulta (SELECT id, uri FROM partidos) para obter uma lista com os IDs e URIs dos partidos previamente armazenados.

Para cada partido na lista, o código faz uma requisição HTTP à URI correspondente, obtendo detalhes atualizados no formato JSON. Se os dados forem encontrados, a URL do logotipo (urlLogo) é extraída e armazenada na tabela "partidos" através de um comando UPDATE. Após a atualização, a conexão é e feito commit para garantir a persistência dos dados. Caso a API não retorne informações, uma mensagem informa que nenhum dado foi encontrado para aquele partido. Durante o processo, mensagens são exibidas para indicar o progresso da atualização dos logotipos.

```
conn = sqlite3.connect(db_path)
cursor = conn.cursor()
cursor.execute("SELECT id, uri FROM partidos")
partidos = cursor.fetchall()
```

```
for id, uri in partidos:
    response = requests.get(uri)
    response.raise_for_status()
    data = response.json()

    if "dados" in data and data["dados"]:
        partido = data["dados"]
        url_logo = partido.get("urlLogo", "")

        cursor.execute("UPDATE partidos SET urlLogo = ? WHERE id = ?", (url_logo, id))
        conn.commit()

        print("URL da logo atualizada com sucesso:", id)

    else:
        print("Nenhum dado encontrado para a URI:", uri)
```

2.7.4 Tabela despesas

E por último a tabela referente a despesas, após estabelecer uma conexão com o banco de dados e recupera os IDs dos deputados armazenados na tabela "deputados_56". Em seguida, percorre cada ID e constrói a URL da API para buscar as despesas correspondentes ao deputado.

A requisição é feita dentro de um loop for que gerencia a paginação dos dados, garantindo que todas as despesas sejam obtidas mesmo se houver várias páginas de resultados. Se a resposta da API contiver despesas, estas são armazenadas em um DataFrame, e colunas específicas são filtradas antes da inserção na tabela "despesas", garantindo a organização dos dados. Para evitar sobrecarga da API, um atraso de meio segundo (time.sleep(0.5)) é aplicado entre as requisições. Em caso de erro na requisição ou qualquer outra exceção, o ID do deputado é armazenado em uma lista de erros (despesas_nao_encontradas), que é exibida ao final para indicar quais deputados não tiveram suas despesas recuperadas.

```
with sqlite3.connect(db_path) as conn:
    deputados = pd.read_sql_query("SELECT id FROM deputados_56", conn)

deputados_ids = deputados[“id”].unique()
despesas_nao_encontradas = []

for deputado in deputados_ids:
    url_despesas = (
        f"https://dadosabertos.camara.leg.br/api/v2/deputados/{deputado}/despesas?"
        f"idLegislatura=56&ano=2022&itens=100&ordem=ASC&ordenarPor=dataDocumento"
    )

    while url_despesas:
        try:
            response = requests.get(url_despesas)
            response.raise_for_status()
            data = response.json()

            if "dados" in data and data["dados"]:
                despesas = pd.DataFrame(data["dados"])

                if not despesas.empty:
                    despesas["id_deputado"] = deputado  # Adicionar o ID do deputado
                    despesas = despesas[[
                        "id_deputado",
                        "ano",
                        "mes",
                        "tipoDespesa",
                        "codDocumento",
                        "tipoDocumento",
                        "codTipoDocumento",
                        "dataDocumento",
                        "numDocumento",
                        "valorDocumento",
```

```
                "valorLiquido",
                "urlDocumento",
                "nomeFornecedor",
                "cnpjCpfFornecedor",
                "codLote"]]
            with sqlite3.connect(db_path) as conn:
                despesas.to_sql("despesas", conn, if_exists="append", index=False)
            time.sleep(0.5)
        else:
            break

        url_despesas = None
        for link in data.get("links", []):
            if link.get("rel") == "next":
                url_despesas = link.get("href")
                break

    except requests.RequestException as e:
        print(f"Erro de requisição para deputado {deputado}: {e}")
        despesas_nao_encontradas.append(deputado)
        break
    except Exception as e:
        print(f"Erro inesperado ao processar deputado {deputado}: {e}")
        despesas_nao_encontradas.append(deputado)
        break

if despesas_nao_encontradas:
    print("Deputados com falha na requisição de despesas:", despesas_nao_encontradas)
```

3. CONSIDERAÇÕES FINAIS

Este trabalho apresentou uma solução para a coleta, processamento e armazenamento de dados sobre deputados federais e suas respectivas despesas, utilizando a API da Câmara dos Deputados e o banco de dados SQLite. A abordagem adotada integra diversas bibliotecas do ecossistema Python, tais como Pandas, Requests, SQLite3, OS e Time, cada uma desempenhando um papel crucial no fluxo de trabalho.

A coleta de dados foi realizada através de requisições HTTP à API da Câmara dos Deputados. Utilizando a biblioteca Requests, foram feitas requisições para obter informações detalhadas sobre os deputados, informações partidárias e despesas parlamentares. A implementação de mecanismos de paginação e tratamento de exceções garantiu a robustez e a completude da coleta de dados.

Os dados coletados foram processados e organizados utilizando a biblioteca Pandas, que permitiu a manipulação eficiente de estruturas tabulares. Os dados foram filtrados e transformados conforme necessário para atender aos requisitos de armazenamento e análise subsequente. A função convert_rede_social foi desenvolvida para converter listas de redes sociais em strings formatadas, facilitando o armazenamento no banco de dados.

O armazenamento dos dados foi realizado em um banco de dados SQLite, garantindo a persistência e a integridade dos dados. Foram criadas tabelas relacionais para armazenar informações detalhadas dos deputados, partidos e despesas parlamentares. A utilização de chaves estrangeiras e índices assegurou a integridade referencial e a eficiência das consultas.

A integração das bibliotecas e a automação do fluxo de trabalho permitiram a execução contínua e eficiente da coleta, processamento e armazenamento de dados. A implementação de atrasos (time.sleep) entre as requisições evitou sobrecarga na API, respeitando os limites de taxa de requisição.

REFERÊNCIAS

Pandas: https://pandas.pydata.org/pandas-docs/stable/

Requests: https://docs.python-requests.org/en/latest/

SQLite: https://www.sqlite.org/docs.html

API da Câmara dos Deputados: https://dadosabertos.camara.leg.br/swagger/api.html