Pipeline de Dados (Esteira de Dados) Brasil.IO

Objetivo do Projeto

O objetivo deste script é automatizar a extração, o armazenamento e a transformação de dados de "Gastos Diretos" da API pública do Brasil.IO. O processo foi desenhado para ser robusto, lidando com grandes volumes de dados (mais de 1000 páginas) e interrupções esperadas da API, como limites de requisição.

O fluxo de trabalho segue um padrão de Data Lakehouse, movendo os dados pelas camadas:

1. **API** (Origem)
2. **RAW / Bruta** (JSON)
3. **BRONZE** (Parquet Particionado)

Como o Código Funciona (Passo a Passo)

O script é dividido em três funções principais que são chamadas em ordem pela função main().

1. A Preparação (ensure_dirs)

Antes de qualquer coisa, o script verifica se a estrutura de pastas (dataset/raw, dataset/bronze, etc.) existe. Se não existir, ele as cria. É um passo simples para garantir que o script não falhe mais tarde ao tentar salvar arquivos em um local inexistente.

2. A Coleta Inteligente de Dados (fetch_and_store_data)

Essa função é responsável por baixar os dados da API (camada RAW).

Lógica de Checkpoint (Continuação):

1. Ela primeiro verifica se o arquivo govbr_all_pages.json já existe na pasta raw.
2. Se existir, ela lê o arquivo, conta quantos registros já foram salvos e calcula em qual página o download parou (dividindo o total de registros por 1000, que é o total de itens por página que a API retorna).

3. Isso garante que, se o script for interrompido (seja pelo erro 429 ou se você o fechar), ao executá-lo novamente, ele não baixará as 57 páginas (ou mais) que já temos, economizando horas de trabalho.

Lógica de Paginação e "Auto-Retry" (O Loop while):

- **Pausa para requisição:** A cada página baixada com sucesso (Status 200), o script pausa por 1 segundo (PAUSE_SECONDS). Isso sinaliza à API que não somos um robô mal-intencionado, o que *ajuda* a evitar o erro 429.
- **Lidando com o Erro 429 (Limite de Requisições):** Quando o script inevitavelmente recebe o erro 429 (Limite de Requisições), ele não aborta. Em vez disso:
    - Ele salva imediatamente todo o progresso (todos os registros na memória) no arquivo govbr_all_pages.json.
    - Inicia uma pausa longa de 60 segundos.
    - Após a pausa, o loop tenta baixar a mesma página que falhou novamente.
- Isso permite que você deixe o script rodando sozinho; ele vai pausar e tentar novamente (quantas vezes for necessário) até conseguir baixar todas as páginas setadas.

3. A Transformação e Organização (transform_to_parquet_and_partition)

Depois que a etapa RAW está completa (ou parcialmente completa), esta função entra em ação para criar a camada BRONZE.

1. **Leitura:** Ela lê o único e grande arquivo govbr_all_pages.json da camada RAW.
2. **Correção de Conflito:** O dataset original já possui colunas chamadas ano e mes. Como queremos usar esses nomes para o particionamento (que cria as pastas), a função primeiro remove essas colunas originais para evitar erros no Pandas.
3. **A procura:** Este é o passo crucial que resolveu o erro KeyError: 'date'.
    a. A função procura por uma lista de nomes de colunas de data prováveis (ex: 'data_pagamento', 'data_pagamento_original').
    b. Ao encontrar 'data_pagamento', ela usa essa coluna para criar as novas colunas ano e mes que serão usadas para o particionamento.
4. **Escrita (Camada Bronze):** Por fim, o script salva o DataFrame inteiro na pasta bronze. O Pandas, usando o comando partition_cols=['ano', 'mes'], não salva um arquivo gigante, mas sim o quebra em vários arquivos Parquet menores, organizados em subpastas (ex: bronze/ano=2024/mes=01/dados.parquet).

4. O Maestro (main)

A função main() apenas orquestra o processo:

1. Chama ensure_dirs().
2. Chama fetch_and_store_data() para baixar tudo.
3. Se a etapa 2 funcionar, chama transform_to_parquet_and_partition() para organizar os dados.