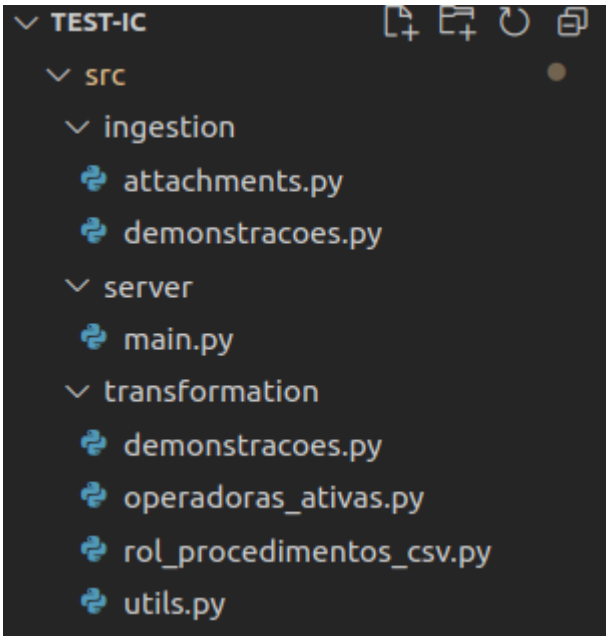
Comentários sobre o teste - IntuitiveCare

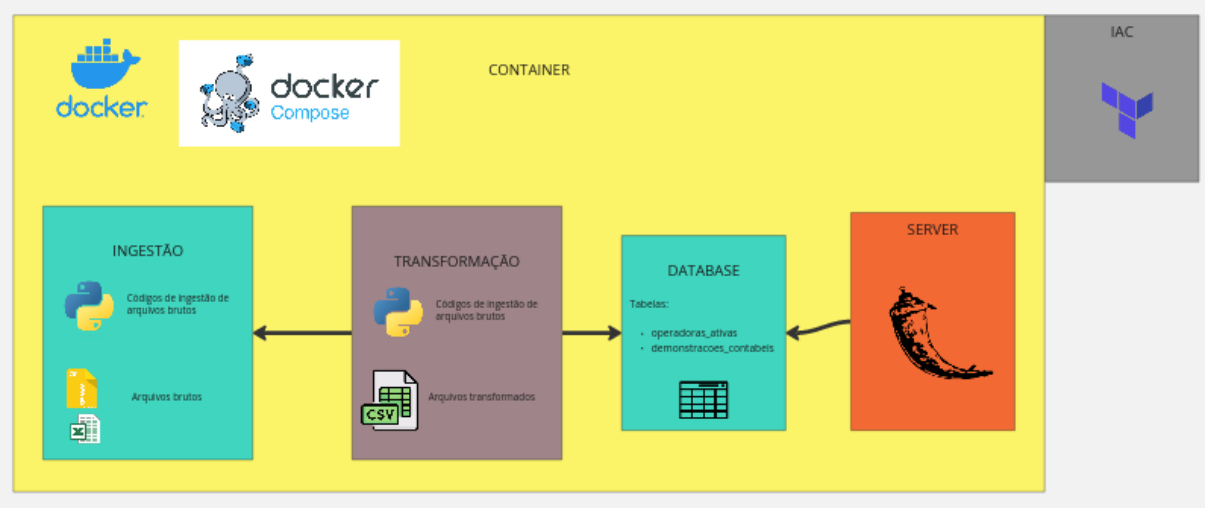
Luiz Vieira - 22/01/2023

Descrição Geral

Nesse documento descrevo os motivos das escolhas que fiz no teste. Procuro esclarecer e explicar as ideias que tive e fazer comentários que ajudem a entender o código.

Em primeiro lugar, escolhi não organizar o código a partir de uma divisão por testes (ex: *teste1/,teste2.py,* etc). Ao invés disso, tentei organizar em camadas que, em geral, fazem sentido em projetos da vida real, já que os 4 testes se conectam como se fosse um único projeto. Por isso criei 3 pastas principais, cada uma se refere a uma camada lógica do processo: *ingestion/*, *transformation/* e *server/*. Os demais arquivos, como p. ex. de conteinerização e IAC, ficaram avulsos na pasta *src/*.





Para ajudar a identificar quais códigos são para quais testes, segue a indicação:

Etapas do teste:

Teste 1 (WebScraping):

- src/intestion/attachments.py

Teste 2 (Transformação de dados):

- src/transformation/rol_procedimentos_csv.py

Teste 3 (Banco de dados):

- src/intestion/demonstracoes.py
- src/transformation/demonstracoes.py
- src/transformation/operadoras_ativas.py
- src/schema.sql
- src/top_10_ano.sql
- src/top_10_trimestre.sql

Teste 4 (API):

- src/server/main.py

A divisão dos arquivos foi escolhida seguindo a lógica de ‘artefatos’, em que cada arquivo representa uma tabela, um arquivo ou um (conjunto de) objeto(s) qualquer. Essa separação facilita a identificação do *data lineage,* a manutenção e a orquestração dos pipelines.

Ex: o arquivo ingestion/demonstracoes.py é responsável somente por baixar os arquivos das demonstrações. O arquivo transformation/demonstracoes.py é responsável somente por ler os arquivos de demonstrações que foram ingeridos e criar a tabela a tabela `demonstracoes_contabeis`.

Foi feita a conteinerização usando docker e docker-compose, para que todo o código possa ser executado independentemente do ambiente. Basta ter o docker instalado no ambiente para executar todos os processos.

Basta executar:

```
Unset
sudo docker-compose build
```

Aguardar a finalização do build que poderá demorar alguns minutos. Se não houver erros, executar:

```
Unset
sudo docker-compose up
```

Essa etapa vai executar o código de ingestão, transformação e finalmente subir o servidor da API (tudo a partir do arquivo *init.sh*). Se tudo estiver funcionando, basta acessar localhost:5000 em um browser para ver a API em funcionamento.

A pasta /terraform contêm um **esboço** de IAC para a cloud GCP.
NOTA: os códigos terraform não puderam ser testados devido à falta de acesso a um projeto na GCP no momento do teste. Eles servem apenas para mostrar uma possível implementação usando IAC.

Sobre o teste 1:

Usei a biblioteca BeautifulSoup do pacote bs4 para fazer o scraper. Essa biblioteca é suficiente para a tarefa, já que a url está definida e a página é aparentemente estática. Para scrapers mais complexos pode-se usar o Selenium, mas é preciso considerar a sua complexidade e o seu grande consumo de memória.

Sempre que possível usei comandos de pacotes do próprio linux (versão debian do container) devido a facilidade e baixo consumo de recursos desnecessários. Nesse caso, baixei os arquivos com *wget* e zipei com *zip*.

Para o nome do arquivo zip, usei a data atual para manter um registro de ingestões por dia (prática mínima de armazenamento em casos reais). Se a ingestão for executada mais de uma vez por dia, o processo vai substituir o zip do dia em questão.

Sobre o teste 2:

Para as transformações, usei a biblioteca pandas, devido a sua versatilidade de ferramentas e a alta documentação e comunidade. Como os dados são pequenos (ainda não precisam de uma stack propriamente de big data), essa biblioteca é suficiente para os processamentos. Em caso de altos volumes, o pandas não é recomendado, nesses casos provavelmente escolheria usar spark em algum ambiente de cluster para processamento distribuído, sendo uma implementação mais complexa.

Os códigos fazem a comparação de arquivos entre a pasta de csv processados e a de anexos vindos da ingestão. A ideia é processar todos os arquivos de ingestão que não possuam um csv correspondente.

Sobre o teste 3:

Novamente uso o pandas, adicionando o conector *pymysql* para conexão com o banco.

- Na transformação da operadoras_ativas:

Renomeei as colunas para eliminar os espaços e ficar mais “sql friendly”.

- Na transformação da demonstracoes_contabeis:

Faço primeiro a verificação do encode, pois os arquivos possuem diferentes codificações. Renomeei as colunas e ajustei os tipos, especialmente os valores numéricos precisam estar com tipo double/float para poderem ser usados em cálculos numéricos.
O arquivo *schema.sql* tem o *create* das tabelas para serem executados antes dos códigos. O *to_sql* executa o create caso as tabelas não existam, mas o create pelo pandas não gera tipagem e indexação adequada, por isso decidi separar o create para ter mais controle sobre a estrutura das tabelas..

As consultas analíticas estão nos arquivos top_10…
Para a consulta de trimestre, coloquei o filtro para o **segundo último trimestre**, pois não havia o arquivo 4T2022.zip disponível no momento do teste.

Sobre o teste 4:

Usei o framework flask para a API, para manter o teste inteiro em linguagem python.
A API pega o conteúdo dos arquivos *.sql*, a depender do endpoint chamado pelo usuário.
Na busca textual, comparei os campos nome_fantasia e razao_social com LIKE, também usei a função *SOUNDEX* para pegar similaridade de fonemas.

localhost:5000	consulta de ano
localhost:5000/ano	consulta de ano
localhost:5000/trimestre	consulta de trimestre.
localhost:5000/busca/<string>	busca textual

Qualquer outra rota irá resultar em erro do handler.

Demais considerações:

- O arquivo .dockerignore foi usado para não levar arquivos desnecessários ao *context* do docker.

- A imagem do container utilizada foi *python-slim,* baseada em debian e ideal para uso com pandas. Costumo usar alpine para coisas mais simples, mas o build no alpine pode ficar muito longo, pois ele não possui os compiladores gcc e g++ (usados pelo *pandas*, *numpy* etc), além de não lidar bem com wheel.

- Uma melhoria possível é separar os processos de ingestão, tranformação e servidor em 3 containers diferentes. Em casos específicos a conteinerização pode ser feita ao nível de artefato/tabela.

- Outra melhoria possível é fazer a conteinerização em cloud no código terraform (ex. kubernetes na gcp).