Desafio CB LAB - Engenharia de Dados
Matheus Rodrigues Santana

Git do projeto: https://github.com/Maathws/CB-LAB

Desafio 1 -

1. **Estrutura e explicação dos dados no ERP.json:**

O arquivo JSON contém informações sobre um pedido (guestCheckId) de um restaurante, incluindo detalhes como itens, impostos aplicados, e informações gerais da transação.

O esquema do arquivo com seus respectivos tipos de dados para inserção no Banco de Dados:

{"columns":[{"curUTC":"DATETIME","locRef":"VARCHAR(255)","guestChecks":"ARRAY"},{"guestCheckId":"BIGINT","chkNum":"INT","opnBusDt":"DATE","opnUTC":"DATETIME","opnLcl":"DATETIME","clsdBusDt":"DATE","clsdUTC":"DATETIME","clsdLcl":"DATETIME","lastTransUTC":"DATETIME","lastTransLcl":"DATETIME","lastUpdatedUTC":"DATETIME","lastUpdatedLcl":"DATETIME","clsdFlag":"BOOLEAN","gstCnt":"INT","subTtl":"DECIMAL(10, 2)","nonTxblSlsTtl":"DECIMAL(10, 2)","chkTtl":"DECIMAL(10, 2)","dscTtl":"DECIMAL(10, 2)","payTtl":"DECIMAL(10, 2)","balDueTtl":"DECIMAL(10, 2)","rvcNum":"INT","otNum":"INT","ocNum":"INT","tblNum":"INT","tblName":"VARCHAR(50)","empNum":"INT","numSrvcRd":"INT","numChkPrntd":"INT","taxes":"ARRAY","detailLines":"ARRAY"},{"taxNum":"INT","txblSlsTtl":"DECIMAL(10, 2)","taxCollTtl":"DECIMAL(10, 2)","taxRate":"DECIMAL(5, 2)","type":"INT"},{"guestCheckLineItemId":"BIGINT","rvcNum":"INT","dtlOtNum":"INT","dtlOcNum":"INT","lineNum":"INT","dtlId":"INT","detailUTC":"DATETIME","detailLcl":"DATETIME","lastUpdateUTC":"DATETIME","lastUpdateLcl":"DATETIME","busDt":"DATE","wsNum":"INT","dspTtl":"DECIMAL(10, 2)","dspQty":"INT","aggTtl":"DECIMAL(10, 2)","aggQty":"INT","chkEmpId":"INT","chkEmpNum":"INT","svcRndNum":"INT","seatNum":"INT","menuItem":"OBJECT"},{"miNum":"INT","modFlag":"BOOLEAN","inclTax":"DECIMAL(10, 2)","activeTaxes":"VARCHAR(255)","prcLvl":"INT"}]}

2. **Transrição do arquivo ERP.json para tabelas SQL em um Bano de Dados:**

A transcrição do JSON para tabelas SQL foi representada de forma prática pelos arquivos disponíveis no repositório Git do projeto. A estrutura do código foi dividida em dois arquivos principais: retornaDicionarioErp.py e dataBase.py, cada um com responsabilidades específicas.

2.1 retornaDicionarioErp.py: Este arquivo contém duas funções que lidam com a leitura e manipulação do arquivo JSON.

A função retornaDicionarioErp() lê o arquivo JSON localmente, o que está sendo utilizado como modelo para este teste prático.
A função retornaDicionarioErpUsandoRequests() faz uma requisição HTTP utilizando a biblioteca requests do Python para obter os dados de um endpoint de API, permitindo uma integração mais dinâmica com o sistema.

2.2 dataBase.py: Este arquivo contém toda a lógica relacionada à estrutura do banco de dados. Ele é responsável por:

A criação do banco de dados, das tabelas SQL, como guest_checks, menu_item, taxes e detail_lines assim como a inserção dos dados do JSON nas tabelas correspondentes, garantindo que a estrutura de dados seja mapeada corretamente.

Além disso, para garantir a segurança no manuseio de informações sensíveis, como credenciais de banco de dados, foi utilizado o arquivo .env para armazenar variáveis de ambiente.

Embora o método .env seja uma prática comum, uma alternativa mais segura seria utilizar um gerenciador de variáveis sensíveis, como o AWS Secrets Manager, que oferece uma forma mais robusta e segura de gerenciar e acessar informações sensíveis em ambientes de produção, como por exemplo a rotação automática e periódica de valores de dados.

3. Abordagem escolhida:

A modelagem do banco de dados foi realizada com uma estrutura normalizada e relacional para garantir integridade, facilitar manutenção e operaçoes de busca. As informações foram separadas em tabelas como guest_checks, menu_item, taxes e detail_lines, evitando redundâncias e assegurando que cada tabela contenha dados específicos.

A escolha por esse modelo relacional se baseia em três premissas principais:

- **Manutenção e Escalabilidade**: O modelo facilita a adição de novos campos ou tabelas à medida que o sistema evolui, sem impactar a estrutura existente.
- **Integridade Referencial**: O uso de chaves estrangeiras garante consistência entre as tabelas, como a relação entre detail_lines e menu_item.
- **Eficiência em Consultas**: A normalização otimiza consultas, o que é crucial para sistemas de alto volume, como os de restaurante.

Campos como discount, serviceCharge, tenderMedia e errorCode foram definidos como nulos em sua criação para evitar desperdício de espaço quando não aplicáveis, sendo adicionado valores apenas quando há o dado no arquivo.
O modelo também permite fácil expansão, com a capacidade de adicionar novos tipos de transações ou impostos sem comprometer a operação do sistema.

4. **Processo de testes:**

Os testes foram realizados em um banco de dados em nuvem via Amazon RDS, garantindo que a solução funcione de forma eficiente e segura. Foram realizados testes de integração para validar a leitura e inserção dos dados do JSON nas tabelas SQL, incluindo testes de performance para verificar a escalabilidade com grandes volumes de dados simulados. A conectividade com o banco foi estável e as consultas SQL apresentaram bom desempenho.

O projeto também foi testado com o arquivo adicional ERP2.json, que contém múltiplos pedidos de um restaurante em um único arquivo, simulando um cenário real com vários registros.

5. **Preocupações:**

Uma das principais preocupações foi a identificação dos dados obrigatórios e opcionais no arquivo ERP, já que dados podem estar ausentes ou em formatos inesperados. Para lidar com isso, uma função de mapeamento seria implementada para garantir que todos os dados obrigatórios estejam presentes e no formato correto. Caso haja falhas, o sistema deve gerar alertas ou tratar os erros adequadamente.

Exemplo de função de validação de dados utilizando a biblioteca logging::

```
def validar_dados(grupamentoDados):
        camposObrigatorios = [ "guestCheckId", "chkNum", "opnBusDt"...]
        for campo in camposObrigatorios:
                if campo not in grupamentoDados:
                        logging.error(f"Campo obrigatorio ausente: {campo}")
                        return False
                if grupamentoDados[campo] is None:
                        logging.error(f"Campo obrigatório '{campo}' é None")
                        return False
…
```

Assim como utilizar a função nativa de python: isinstance() para verificar o tipo do dado no arquivo .json antes de inserir no banco de dados.

Outra preocupação é a possibilidade de mudanças na estrutura do arquivo ERP, o que exige que o sistema seja flexível e capaz de se adaptar a novas versões do arquivo, mantendo a integridade dos dados. A solução também precisa ser capaz de lidar com grandes volumes de dados, sendo fundamental realizar testes de carga e otimizar o banco de dados para garantir boa performance, mesmo com grandes transações.

Desafio 2 -

1. **Por que armazenar as respostas das APIs?**

Armazenar as respostas das APIs é essencial para garantir a persistência dos dados e possibilitar análises detalhadas no futuro. Embora as APIs forneçam informações em tempo real, nem sempre conseguimos processar de imediato ou acessar com a frequência desejada. Por isso, manter essas respostas armazenadas traz uma série de benefícios importantes, como:

- **Análises históric**as: Com os dados armazenados, é possível realizar comparações de desempenho, identificar padrões no comportamento dos clientes e analisar a evolução da receita ao longo do tempo, fornecendo insights valiosos para a tomada de decisão.

- **Validação e verificação**: O armazenamento das respostas permite validar a integridade dos dados antes de utilizá-los em relatórios ou sistemas de inteligência de negócios, garantindo que as informações sejam consistentes e confiáveis.

- **Disponibilidade contínua:** Em casos de indisponibilidade temporária da API, o armazenamento local garante que os dados necessários para análises e relatórios ainda estejam acessíveis, sem interrupções no processo.

- **Facilidade em auditorias:** Manter um histórico das respostas também é importante para auditorias e rastreamento, permitindo comprovar a origem e a consistência das informações utilizadas em relatórios e decisões estratégicas.

Essa prática não só facilita a manipulação e análise dos dados, mas também garante maior segurança e eficiência em um ambiente de negócios onde a confiabilidade da informação é crítica.

2. **Arquitetura de armazenamento de dados:**

Estrutura de pastas:
/data-lake/apiName/busDt/response.json

Podendo ser alterado para datas dinâmicas, trazendo novas funcionalidades.
/data-lake/apiName/dataInicial={dia}/{mes}/{ano}&dataFinal={dia}/{mes}/{ano}/response.json

Cada resposta de API será salva como response.json dentro de diretórios que indicam o nome da API, o intervalo de datas e o storeId. Isso facilita a organização e a consulta dos dados por períodos e lojas.

Com essa estrutura, os dados podem ser facilmente acessados e consultados utilizando ferramentas como AWS Athena. A solução é escalável, permitindo o armazenamento de

grandes volumes de dados de maneira eficiente, com flexibilidade para incluir novos endpoints ou ajustes nos dados conforme necessário.

2.1. Upload do json no datalake, arquivo desafio2.py:

O arquivo desafio2.py realiza o upload de arquivos JSON filtrados para um Data Lake na AWS S3. Ele carrega variáveis de ambiente para configurar credenciais e região do S3, organiza os dados em uma estrutura de diretórios baseada no endpoint (apiName) e na data operacional (busDt), e utiliza o boto3 para conectar-se ao S3 e enviar os dados. O JSON é convertido de um dicionário Python para o formato adequado e armazenado com o tipo application/json. Após o upload, o caminho completo do arquivo é exibido, garantindo rastreabilidade e organização eficiente dos dados para inteligência de negócios.

3. Alteração de nomenclatura de dados:

Mudanças no formato dos dados, como alterações em nomes de campos ou estrutura das respostas das APIs, podem causar impacto direto na ingestão e no processamento dos dados. Para lidar com isso de forma global e eficiente, o sistema deve ser projetado para detectar, mapear e adaptar-se a mudanças de nomenclatura ou estrutura, independentemente das variáveis ou do arquivo em questão.

- **Funções de Mapeamento de Nomenclatura:** O sistema deve implementar funções automatizadas que identifiquem diferenças de nomenclatura entre versões dos arquivos JSON. Essas funções devem comparar as chaves das novas respostas com as nomenclaturas previamente conhecidas, tentando inferir relações entre elas com base em similaridades ou no contexto geral dos dados.

- **Identificação de Relações:** Se for detectada uma nova chave que não estava mapeada anteriormente, o sistema deve tentar identificar a relação dela com as nomenclaturas antigas. Por exemplo, se o campo taxation substituir taxes, a função de mapeamento deve reconhecer que ambos se referem a impostos, com base em padrões de nomenclatura ou no conteúdo associado.

- **Notificação ao Time de Desenvolvimento:** Após identificar alterações ou novas nomenclaturas, o sistema deve notificar o time de desenvolvimento, e com a possível identificação de relação com dados antigos pode trazer eficiência no tempo de resposta para correção do erro e/ou tomar decisões para um possível novo modelamento de dados.