1. Por que armazenar as respostas das APIs?

Armazenar as respostas das APIs no data lake traz várias vantagens estratégicas e operacionais para a análise de dados e o processo de tomada de decisão na empresa. Aqui estão algumas razões principais:

- Centralização e Acessibilidade: O armazenamento centralizado das respostas facilita o acesso e a análise dos dados em um único repositório. Isso também facilita a integridade dos dados, pois todas as informações da loja estão em um formato acessível e organizado.
- Análises Históricas e Preditivas: Ao armazenar as respostas das APIs, é possível realizar análises históricas para comparar o desempenho das lojas ao longo do tempo e realizar previsões de vendas, identificar padrões de comportamento do cliente e otimizar o gerenciamento de estoque.
- Flexibilidade e Escalabilidade: O data lake pode armazenar grandes volumes de dados estruturados, semi-estruturados e não estruturados. A flexibilidade permite que você armazene diferentes tipos de dados (como JSON, CSV, logs, imagens) e as respostas de diferentes APIs, sem a necessidade de transformação imediata, otimizando o tempo de processamento.
- Facilidade de Integração e ETL: Uma vez que os dados são armazenados no data lake, eles podem ser facilmente processados usando pipelines de ETL (Extração, Transformação e Carga) para serem carregados em sistemas analíticos ou em data warehouses para análise em tempo real.

2. Como você armazenaria os dados? Crie uma estrutura de pastas capaz de armazenar as respostas da API. Ela deve permitir manipulações, verificações, buscas e pesquisas rápidas.

A estrutura de pastas para armazenar as respostas das APIs deve ser organizada de maneira lógica e escalável, considerando a data das transações, a loja e o tipo de dado da API. A hierarquia pode ser algo como o exemplo abaixo:

```
/data_lake/
    ├── /raw/
    │   ├── /fiscal_invoices/
    │   │   ├── store_001/
    │   │   │   ├── 2024-01-01_fiscal_invoice.json
    │   │   │   ├── 2024-01-02_fiscal_invoice.json
    │   │   ├── store_002/
    │   │   │   ├── 2024-01-01_fiscal_invoice.json
    │   ├── /guest_checks/
    │   │   ├── store_001/
    │   │   │   ├── 2024-01-01_guest_check.json
    │   │   │   ├── 2024-01-02_guest_check.json
    │   │   ├── store_002/
    │   │   │   ├── 2024-01-01_guest_check.json
    │   ├── /chargebacks/
    │   │   ├── store_001/
    │   │   │   ├── 2024-01-01_chargeback.json
    │   ├── /transactions/
    │   │   ├── store_001/
    │   │   │   ├── 2024-01-01_transaction.json
    │   ├── /cash_management_details/
    │   │   ├── store_001/
    │   │   │   ├── 2024-01-01_cash_management.json
    │
    ├── /processed/
    │   ├── /guest_checks/
    │   ├── /fiscal_invoices/
    │   ├── /chargebacks/
    │   ├── /transactions/
    │   ├── /cash_management_details/
    │
    ├── /logs/
    │   ├── api_logs/
    │   │   ├── 2024-01-01_api_call_log.json
    │   │   ├── 2024-01-02_api_call_log.json
```

Explicação da Estrutura:

1. /raw/: A pasta "raw" contém os dados brutos (sem processamento) diretamente das respostas das APIs. Eles são armazenados em subpastas por tipo de API (`/fiscal_invoices/`, `/guest_checks/`, etc.), e por loja (`store_001`, `store_002`), utilizando a data (`busDt`) como parte do nome do arquivo para facilitar a consulta por data.
2. /processed/: Após os dados serem processados e transformados, como a limpeza, normalização ou junção de informações, os dados processados são armazenados nesta pasta. Isso permite que as análises sejam feitas a partir de dados já preparados e consistentes.
3. /logs/: Registros de chamadas da API podem ser armazenados aqui para auditoria e rastreamento. Esses logs ajudam na depuração e podem ser úteis para verificar erros e otimizar o desempenho das consultas.
4. Subpastas por loja: Organizar os dados por loja (`store_001`, `store_002`) permite consultas rápidas e segregação dos dados de forma eficiente.
5. Arquivos por data: Armazenar os arquivos com o formato `YYYY-MM-DD` permite que os dados sejam indexados cronologicamente, facilitando a busca e análise de dados históricos.

---

3. Considerando que a resposta do endpoint `getGuestChecks` foi alterada, por exemplo, o campo `guestChecks.taxes` foi renomeado para `guestChecks.taxation`. O que isso implicaria?

Essa alteração de nome de campo (renomeação de `taxes` para `taxation`) tem implicações no processo de extração, transformação e carga (ETL) e na manipulação de dados:

- Mudança no Pipeline de ETL: Qualquer script ou processo de ETL que dependa desse campo deverá ser atualizado para refletir a nova nomenclatura. Isso pode significar modificações no código que trata a ingestão de dados, especialmente se o pipeline estiver hardcoded (com referências diretas ao campo anterior).
- Transformação dos Dados: Durante a etapa de transformação, onde os dados podem ser ajustados, limpos ou mapeados, o campo antigo (`taxes`) precisa ser

substituído pela nova chave (`taxation`). Isso deve ser feito de forma transparente para que o processamento continue sem falhas.

- Impacto nas Consultas e Dashboards: Se houver consultas SQL ou dashboards analíticos configurados para utilizar o campo `taxes`, essas também precisarão ser alteradas para garantir que os relatórios e análises funcionem corretamente. O sistema precisará ser adaptado rapidamente para lidar com a nova estrutura do dado.
- Documentação e Notificação: Para garantir que todos os envolvidos no processo de manipulação dos dados estejam cientes da mudança, é importante atualizar a documentação do sistema de dados e informar as equipes técnicas que dependem dessa API para ajustar seus fluxos de trabalho.

Portanto, uma mudança como essa exige que o processo de ingestão e transformação dos dados seja revisado, que scripts e consultas sejam atualizados, e que a documentação da API seja mantida em dia para garantir a continuidade das operações de análise de dados.