Coco Bambu - Desafio de Engenharia de Dados

Autor: Igor Costa Fernadez

Data de entrega: Julho de 2025

Descrição do Esquema JSON

O arquivo JSON representa dados operacionais de um restaurante, estruturados de forma hierárquica e altamente relacional. O nível raiz contém informações gerais da loja, como o código da unidade (locRef) e a data/hora atual (curUTC). O campo principal dessa estrutura é guestChecks, uma lista de comandas realizadas no restaurante.

Estrutura das Comandas (guestChecks)

Cada objeto da lista guestChecks representa uma comanda individual, com informações detalhadas como número da comanda, data e hora de abertura e fechamento (tanto em UTC quanto no horário local), número de convidados, totais financeiros (subtotal, total, descontos, pagamentos), número da mesa, e identificação do funcionário. Esses dados fornecem uma visão completa do ciclo de vida de uma comanda no restaurante.

Impostos (taxes)

Dentro de cada comanda, há uma lista taxes, contendo os impostos aplicados. Cada imposto possui campos como taxNum (ID do imposto), txblSlsTtl (valor tributável), taxRate (percentual de imposto) e taxCollTtl (valor efetivamente cobrado). Essa estrutura permite análises fiscais detalhadas por comanda.

Itens da Comanda (detailLines)

O campo detailLines traz uma lista com os itens ou ações vinculadas à comanda. Cada item possui um identificador único (guestCheckLineItemId), informações de tempo, quantidade, valor, estação de trabalho, funcionário responsável, entre outros.

Importante: cada detailLine pode conter diferentes tipos de objetos aninhados, representando operações específicas:

- menuItem: representa o produto do cardápio vendido, com dados como código do item (miNum), valor com imposto (inclTax), e modificações.
- discount *(opcional)*: descontos aplicados no item.
- serviceCharge *(opcional)*: taxas de serviço adicionais.
- tenderMedia *(opcional)*: forma de pagamento vinculada.
- errorCode *(opcional)*: mensagens de erro ou inconsistências.

Considerações Técnicas

Esse esquema é ideal para modelagem relacional em bancos de dados SQL. A separação clara entre comandas, itens, impostos e cardápio permite criar tabelas com relacionamentos normalizados. Como os campos discount, serviceCharge, tenderMedia e errorCode são opcionais e aparecem apenas em determinadas situações, o mais indicado é modelá-los em tabelas separadas ligadas pelo guestCheckLineItemId.

Abordagem Adotada – Desafio 1

A solução adotada para transcrever o JSON de comandas do restaurante para um modelo relacional foi baseada em uma análise minuciosa da estrutura dos dados e na aplicação de boas práticas de modelagem em bancos de dados.

1. Análise Estrutural do JSON

O primeiro passo foi entender o esquema do arquivo JSON fornecido, que representa dados operacionais de um restaurante, como comandas de venda (guestChecks), itens pedidos (detailLines), impostos (taxes) e dados adicionais como descontos e formas de pagamento. Observou-se que o JSON apresenta uma estrutura hierárquica com múltiplos níveis aninhados e campos opcionais (como discount, serviceCharge, tenderMedia, errorCode), o que exigiu uma abordagem flexível para modelagem.

2. Normalização dos Dados

Em vez de converter tudo para uma tabela única (o que levaria a campos nulos e redundantes), optou-se pela normalização dos dados. As entidades principais foram separadas em tabelas distintas — por exemplo, comandas, itens, impostos e itens de menu. Isso permitiu reduzir a redundância e manter a integridade dos dados. Para campos opcionais, como

discount, foram criadas tabelas auxiliares, permitindo uma estrutura limpa, consistente e extensível.

3. Modelo Relacional Escalável

A modelagem foi feita de forma a suportar grandes volumes de dados e ser reutilizável em contextos reais de restaurantes com múltiplas lojas. A chave locRef foi usada como indicador da loja, possibilitando segmentação. As tabelas relacionais permitem análises como ticket médio, itens mais vendidos, impostos cobrados por comanda, entre outras.

4. Simulação com SQLite e Pandas

Toda a transformação foi feita com bibliotecas Python (pandas, json, sqlalchemy) e executada em ambiente notebook (Google Colab), utilizando SQLite em memória para simular um banco real. Isso permitiu validar a estrutura gerada, testar queries SQL e demonstrar como os dados seriam utilizados em cenários analíticos ou operacionais.

Justificativa da Escolha

A escolha por um modelo relacional foi motivada por vários fatores:

- A estrutura dos dados possui chaves bem definidas e relações fortes, o que se encaixa melhor em bancos relacionais do que em NoSQL.
- Consultas SQL são mais apropriadas para relatórios, análises gerenciais e sistemas de BI.
- Separar os dados em tabelas específicas melhora a escalabilidade e evita problemas de performance em grandes volumes.
- O modelo pode ser migrado facilmente para MySQL, PostgreSQL ou Data Warehouses modernos.
- É compatível com ferramentas de ETL, orquestração (como Airflow) e pipelines de dados.

Justificativa para armazenamento e estruturação de respostas de APIs no Data Lake

Armazenar as respostas de APIs em um Data Lake bem estruturado traz benefícios significativos tanto para confiabilidade dos sistemas quanto para performance de análise de dados. Abaixo, apresento os principais motivos e como a estrutura de diretórios proposta contribui para atingir esses objetivos.

Benefícios estratégicos do armazenamento

O armazenamento de dados traz diversos benefícios estratégicos para sistemas e processos analíticos. Primeiro, permite **persistência histórica**, uma vez que APIs

normalmente não mantêm histórico. Com os dados armazenados, é possível realizar auditorias, reprocessamentos e comparações ao longo do tempo.

Além disso, há maior **resiliência a falhas**: em situações de instabilidade ou alteração nas APIs, os dados já gravados garantem a continuidade dos processos sem interrupções. Outro ponto importante é a **performance analítica**, já que evita múltiplas chamadas às APIs e favorece análises locais mais rápidas e eficientes.

O armazenamento também contribui para a **padronização e centralização**, criando uma fonte única de dados, o que reduz inconsistências e facilita a governança. Por fim, há ganhos significativos em **segurança e compliance**, pois os dados podem ser armazenados com controle de acesso e versionamento — algo essencial para atender a exigências legais e regulatórias.

Justificativa da estrutura de diretórios proposta

A estrutura é eficaz, escalável e pronta para integração com pipelines automatizados e sistemas distribuídos:

```
/data_lake/
  └── receita/
      ├── endpoint=getFiscalInvoice/
      │   └── loja=store_001/
      │       └── ano=2025/
      │           └── mes=07/
      │               └── dia=24/
      │                   └── fiscal_invoice.json
      ├── endpoint=getGuestChecks/
      │   └── loja=store_001/
      │       └── ano=2025/
      │           └── mes=07/
      │               └── dia=24/
      │                   └── guest_checks.json
      ├── endpoint=getChargeBack/
      ├── endpoint=getTransactions/
      └── endpoint=getCashManagementDetails/
```

A organização da estrutura de armazenamento traz vantagens técnicas e funcionais importantes. O uso de componentes como endpoint=... permite **agrupar dados por tipo**, como fiscal ou transações, o que facilita a criação de agregações específicas para análise. Já a

inclusão de loja=store_001 promove a **isolação por unidade ou franquia**, sendo extremamente útil para **filtragens regionais e estudos localizados**.

A estrutura baseada em ano/mes/dia oferece **suporte nativo a partições temporais**, o que é essencial para **processamento incremental** e otimização de rotinas analíticas. Por fim, manter uma **nomenclatura consistente** permite a **manipulação automatizada** da estrutura através de scripts e ferramentas de orquestração como o Apache Airflow, tornando o fluxo de dados mais robusto e escalável.

Impacto na mudança da resposta do endpoint

Quando o campo guestChecks.taxes é renomeado para guestChecks.taxation, isso implica em diversas consequências técnicas e operacionais. Embora possa parecer uma simples alteração nominal, seu impacto pode ser significativo em todo o pipeline de dados, conforme descrito abaixo:

- **Quebra de compatibilidade**: Scripts, consultas SQL, ETLs ou transformações que referenciam diretamente guestChecks.taxes deixarão de funcionar, gerando falhas ou campos nulos. Dashboards e relatórios também podem apresentar erros ou dados ausentes.
- **Atualização obrigatória em sistemas e pipelines**: É necessário revisar e adaptar todos os componentes que consomem esse dado, desde a ingestão até a camada de visualização. Isso inclui ajustes em schemas, mapeamentos, documentação e modelos de machine learning que usem esse atributo.
- **Necessidade de versionamento de schema**: Mudanças na estrutura da API exigem controle de versão para garantir rastreabilidade. O schema anterior deve ser documentado, e a nova versão, registrada com suas alterações, para permitir reprocessamentos e comparações históricas.
- **Validação semântica**: Apesar de parecer um simples rename, é importante confirmar se o campo taxation tem a mesma semântica e formato de taxes. Caso contrário, pode haver distorções analíticas ou interpretações erradas.
- **Governança e comunicação**: Alterações devem ser comunicadas aos times técnicos e de negócio. A documentação da API (como Swagger/OpenAPI) precisa ser atualizada, assim como o catálogo de dados e ferramentas de data lineage.

- **Riscos de regressão**: Se não for feita uma validação completa nos ambientes de teste, há risco de regressão silenciosa nos dados — o pipeline pode rodar, mas os resultados podem ser inconsistentes ou incompletos.

Para garantir a compatibilidade após a alteração do campo, é essencial atualizar todas as referências ao campo antigo taxes, substituindo-as por taxation. Além disso, para lidar com diferentes versões de dados e evitar falhas no processamento, é recomendável aplicar um fallback condicional no código que busque primeiro por taxation e, caso não exista, utilize taxes. Dessa forma, o sistema continuará funcionando corretamente, independentemente da versão do schema.

imposto = guest.get("taxation") or guest.get("taxes")