# Documento Arquitetural de Dados - Projeto Amazonas (V2)

## 1) Descrição do Sistema

A loja fictícia **Amazonas** nasceu como uma empresa de comércio físico e agora está expandindo para o digital.

O objetivo é construir um e-commerce robusto, capaz de atender milhões de clientes e processar pedidos com alta disponibilidade e performance. Para isso, optamos pelo uso do **MongoDB** como banco **NoSQL**, explorando sua flexibilidade, escalabilidade horizontal e suporte a estruturas desnormalizadas.

Nosso foco foi projetar o modelo de dados, reforçando a ideia de desnormalização para reduzir dependência de joins e facilitar leituras rápidas, que são fundamentais em aplicações de e-commerce.

## 2) Estrutura de Dados Proposta

### Coleções Principais

Foram desenvolvidas seis coleções principais que representam os aspectos críticos do sistema:

1. **customers (clientes)** 
   * Atributos principais: `customer\_id`, `name`, `email`, `phones`, `addresses`, `created\_at`.
   * Finalidade: armazenar dados cadastrais e endereços dos clientes.
   * Decisão: mantida como dimensão, mas snapshots de informações relevantes são replicados em `orders` e `reviews`.
2. **products (produtos)** 
   * Atributos principais: `product\_id`, `title`, `description`, `category`, `brand`, `price`, `stock`, `status`.
   * Finalidade: catálogo de produtos.
   * Decisão: também tratada como dimensão, mas snapshots de preço, nome e categoria são copiados em `orders`, `carts` e `reviews` para preservar histórico e agilizar consultas.
3. **carts (carrinhos)**

* Estrutura: cada cliente possui um único carrinho ativo.
* Atributos principais: `customer\_id`, `items[]` contendo `product\_id`, `qty`, e um `product\_snapshot` (nome, categoria, preço no momento da adição).
* Decisão: desnormalizar para evitar busca no catálogo a cada exibição de carrinho.

1. **orders (pedidos)**

* Estrutura: documento com `order\_id`, `customer\_snapshot`, `items[]` com `product\_snapshot`, endereço de entrega, resumo de pagamento, total, status.
* Decisão: desnormalizado. Guardamos snapshot de cliente (nome, e ail, endereço padrão) e snapshot dos produtos (preço e status na época da compra). Isso garante consistência histórica e leitura rápida.

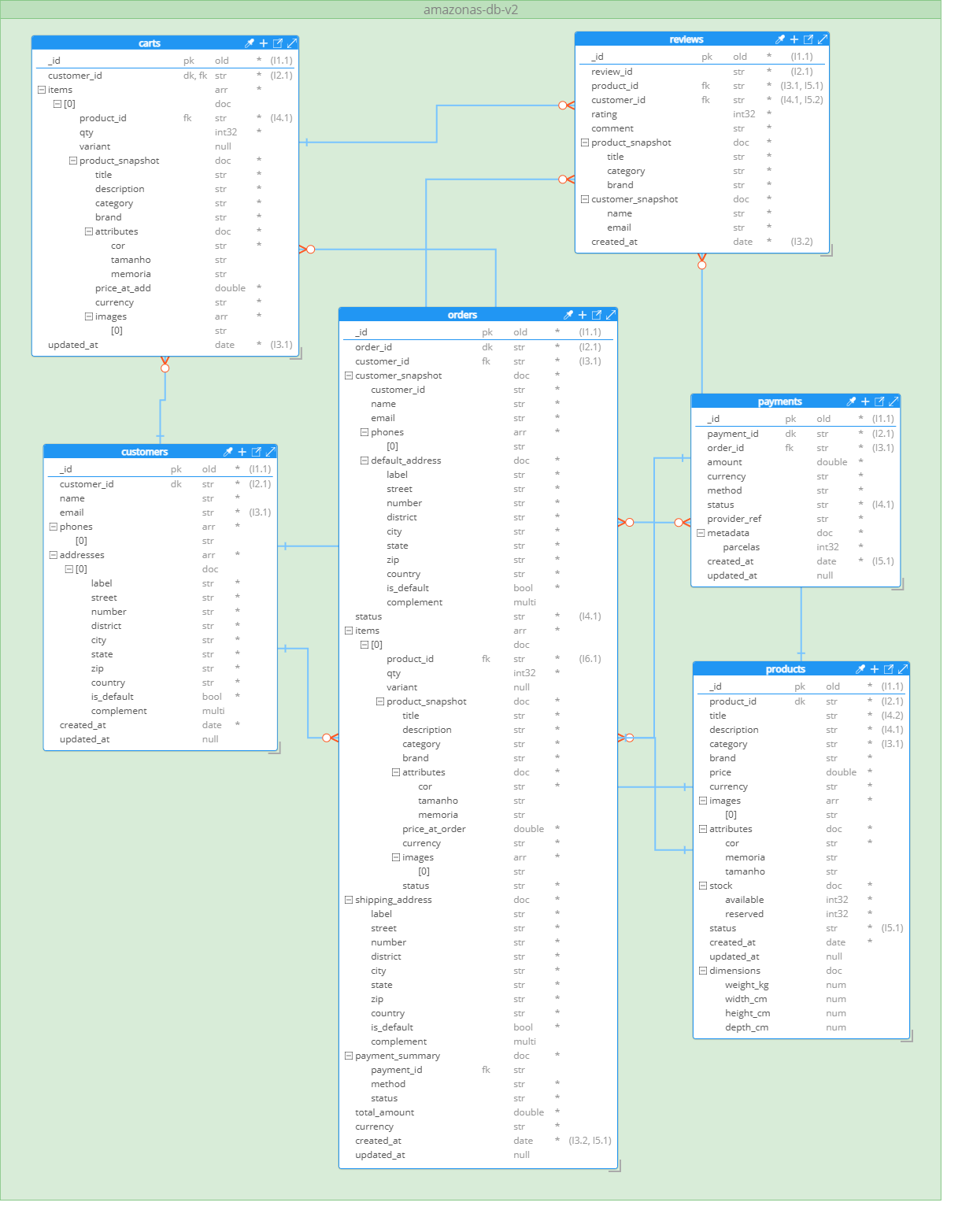
1. **reviews (avaliações)**

* Estrutura: `review\_id`, `product\_id`, `customer\_id`, `rating`, `comment`, além de `product\_snapshot` e `customer\_snapshot`.
* Decisão: snapshots evitam necessidade de joins para exibir avaliações com nome do cliente e informações básicas do produto.

1. **payments (pagamentos)**

* Estrutura: `payment\_id`, `order\_id`, `amount`, `method`, `status`, `provider\_ref`.
* Decisão: coleção separada para permitir múltiplos registros de pagamento por pedido. Como pode haver reprocessamento ou tentativas, preferimos manter separado e vincular via `order\_id`.

### Representação visual do modelo (Hackolade studio)



## Relações entre Coleções

* **customers → orders**: relação 1:N. Um cliente pode ter vários pedidos.
* **customers → carts**: relação 1:1. Cada cliente tem apenas um carrinho ativo.
* **customers → reviews**: relação 1:N.
* **products → reviews**: relação 1:N.
* **orders → payments**: relação 1:N.

Apesar das relações, optamos por evitar foreign keys e em vez disso replicar informações críticas (snapshots).

Exemplo: se o preço de um produto mudar, o `order` continua refletindo o valor pago no momento da compra.

## 3) Justificativa da Desnormalização

* **Performance de leitura**: consultas típicas de e-commerce (histórico de pedidos, carrinho, avaliações) ficam rápidas porque os documentos já contêm todos os dados necessários.
* **Histórico imutável**: snapshots preservam contexto (preço, descrição, status) no momento da compra ou avaliação.
* **Escalabilidade**: ao reduzir joins e lookups, facilitamos particionamento e replicação, pontos fundamentais para lidar com grandes volumes.
* **Trade-off**: aceitamos certa redundância (ex.: nome do cliente repetido em `orders` e `reviews`) em troca de consultas mais ágeis e consistentes.

## 4) Plano de Escalabilidade (Visão Inicial)

### Sharding (Particionamento)

* **orders**: shard key `{ customer\_id, created\_at }` para distribuir pedidos de clientes ao longo do tempo.
* **payments**: shard por `{ order\_id }`, mantendo afinidade com pedidos.
* **reviews**: shard por `{ product\_id }` para balancear conforme volume de produtos populares.
* **products**: pode ser shard por `category`, mas dado o volume mais baixo em relação a pedidos, talvez não seja necessário no início.

### Réplicas (Alta Disponibilidade)

* **customers** e **products** replicados para leitura em múltiplos nós (catálogo e perfis têm alta taxa de leitura).
* **orders** e **payments** replicados para garantir tolerância a falhas e consistência.

### Como lidar com Crescimento de Dados

* **orders** será a coleção de maior volume. Sharding horizontal distribui a carga entre diferentes nós.
* **carts** são pequenos e transitórios, não demandam particionamento.
* Estratégia de TTL (time to live) pode ser aplicada a `carts` inativos.

### Como lidar com Alta Concorrência

* MongoDB suporta alta taxa de escrita. Índices cuidadosamente planejados garantem consultas rápidas.
* Leituras intensivas podem ser direcionadas para réplicas secundárias.
* Escritas críticas (ex.: `orders`, `payments`) seguem para primário.

## 5) Implementação em Atlas ou DynamoDB

Se fosse implementado em nuvem:

* **MongoDB Atlas**: facilitaria a configuração de réplicas e sharding automático, além de monitoramento nativo.
* **DynamoDB (AWS)**: seria alternativa interessante, com chave de partição baseada em `customer\_id` ou `order\_id`, aproveitando a elasticidade nativa do serviço.
* Em ambos os casos, o modelo desnormalizado se adapta bem, já que consultas típicas de e-commerce o baseadas em chave-partição e não exigem joins complexos.

## 6) Conclusão

O modelo **amazonas-db-v2** foi projetado de forma desnormalizada, priorizando performance de leitura e consistência histórica.

A redundância intencional (snapshots de clientes e produtos em pedidos, carrinhos e avaliações) garante que o sistema responda rápido às consultas mais comuns sem depender de joins.

Com sharding e réplicas, o sistema se torna capaz de lidar com o crescimento exponencial de clientes e pedidos, mantendo-se disponível e escalável.

O código do projeto se encontra no meu perfil do [github](https://github.com/lucas-placido/xpe-amazonas-nosql-db).