SQL vs NoSQL: Escolhendo a Tecnologia Certa

Estudo de Caso de Migração e Integração Híbrida

A escolha entre SQL e NoSQL não é binária. O mundo real usa ambos em uma Arquitetura Híbrida (Polyglot Persistence).

Abertura: SQL vs NoSQL - Revisão e Contexto Híbrido

SQL (Relacional)

ESTRUTURA

Tabelas com schemas rígidos. Relações entre tabelas via Foreign Keys.

CONSISTÊNCIA

ACID: Atomicidade, Consistência, Isolamento, Durabilidade. Dados sempre corretos.

ESCALABILIDADE

Escalabilidade Vertical (mais poder no servidor). Difícil escalar horizontalmente.

IDEAL PARA

Dados Transacionais: Faturamento, Credenciais, Integridade crítica.

EXEMPLO

PostgreSQL, MySQL, Oracle

NoSQL (Não-Relacional)

ESTRUTURA

Documentos, Chave-Valor, Grafos. Schemas flexíveis (Schema-on-Read).

CONSISTÊNCIA

BASE: Básicamente Disponível, Estado Soft, Eventual Consistency. Dados eventualmente corretos.

ESCALABILIDADE

Escalabilidade Horizontal (mais servidores). Distribuído por natureza.

IDEAL PARA

Dados de Alto Tráfego: Sessões, Históricos, Recomendações, Catálogos.

EXEMPLO

MongoDB, DynamoDB, Redis, Cassandra

Arquitetura Híbrida (Polyglot Persistence)

O mundo real não escolhe entre SQL ou NoSQL. Escolhe **ambos**. Cada banco de dados é otimizado para um problema específico. Exemplo: Netflix usa PostgreSQL para transações e MongoDB para recomendações. Amazon usa DynamoDB para sessões e RDS para faturamento. Essa é a **Arquitetura Híbrida**.

Gatilhos de Migração: Por que Migrar?

1 Volume de Dados (Escala)

DESCRIÇÃO

Crescimento exponencial de dados: logs, IoT, eventos, cliques. SQL não escala horizontalmente bem.

IMPACTO

Servidor SQL fica lento. Custos de hardware crescem. Queries demoram minutos.

EXEMPLO

Netflix: 1 bilhão de eventos/dia. PostgreSQL não aguenta. Migrou para NoSQL.

2 Performance (Lentidão de JOINs)

DESCRIÇÃO

JOINs complexos (5-6 tabelas) em escala. Cada JOIN adiciona latência. Página inicial demora a carregar.

IMPACTO

Latência de 500-800ms. Usuários abandonam. Receita cai. SLA violado.

EXEMPLO

Feed de recomendações: 6 JOINs = 800ms. MongoDB com documento denormalizado = 50ms.

3 Flexibilidade (Schema Dinâmico)

DESCRIÇÃO

Necessidade de alterar schema rapidamente. Novos campos aparecem frequentemente (reviews, tags).

IMPACTO

ALTER TABLE em produção é arriscado. Downtime. Rollback complexo. Velocidade de desenvolvimento cai.

EXEMPLO

Perfil de usuário: adicionar campo "preferências" em MongoDB = instantâneo. Em SQL = downtime.

Arquitetura Híbrida: SQL + NoSQL

Quando Usar SQL

PRIORIDADE

ACID: Atomicidade, Consistência, Isolamento, Durabilidade. Dados SEMPRE corretos.

CARACTERÍSTICAS

- Dados críticos para negócio
- Transações complexas
- Integridade referencial
- Relatórios com JOINs

EXEMPLOS DE DADOS

- Faturamento e Pagamentos
- Credenciais de Login
- Dados Financeiros
- Pedidos Finalizados
- Saldo Bancário

EXEMPLO DE SGBD

PostgreSQL, MySQL, Oracle, SQL Server

Quando Usar NoSQL

PRIORIDADE

BASE: Disponibilidade, Tolerância à Partição. Dados eventualmente consistentes.

CARACTERÍSTICAS

- Alto tráfego de leitura
- Dados que mudam frequentemente
- Escalabilidade horizontal
- Latência muito baixa

EXEMPLOS DE DADOS

- Sessões de Usuário
- Histórico de Visualização
- Recomendações
- Carrinho de Compras
- Cache de Catálogo

EXEMPLO DE SGBD

MongoDB, DynamoDB, Redis, Cassandra

Integração na Aplicação

Estudo de Caso: Plataforma de Streaming

Cenário: Uma plataforma de streaming (Netflix-like) com milhões de usuários. Problema: página inicial demora 500-800ms para carregar porque precisa de 5-6 JOINs complexos para montar o Feed de Recomendação. Solução: migrar dados de alto tráfego para NoSQL.

Antes: SQL (PostgreSQL)

Problema de Performance

ESTRUTURA

6 tabelas relacionadas: users, watch_history, videos, recommendations, categories, ratings

QUERY

```
SELECT v.title, wh.watched_at
FROM users u
JOIN watch_history wh
JOIN videos v
JOIN recommendations r
...
```

LATÊNCIA

500-800ms

PROBLEMA

Cada JOIN adiciona latência. Escala de milhões de usuários torna insuportável.

Depois: NoSQL (MongoDB)

Solução de Performance

ESTRUTURA

1 documento denormalizado por usuário com histórico e recomendações embutidas

QUERY

```
db.users.findOne({
  "_id": "user_123"
})
// Retorna tudo em 1 leitura
```

LATÊNCIA

10-50ms

VANTAGEM

Leitura única. Sem JOINs. 50x mais rápido. Escalável horizontalmente.

GANHOS DA MIGRAÇÃO

- Latência reduzida: 500-800ms → 10-50ms (50x)
- Escalabilidade horizontal: mais servidores
- Disponibilidade: tolerância a falhas
- Flexibilidade: adicionar campos sem downtime

DESAFIOS DA MIGRAÇÃO

- Redundância: título do vídeo em múltiplos documentos
- Eventual Consistency: dados podem ficar desatualizados
- Sincronização: garantir que mudanças se propagam
- Complexidade de código: aplicação gerencia redundância

Atividade 1: Análise em Grupo do Estudo de Caso

Cenário: Plataforma de Streaming (Netflix-like) migrou dados de usuários, histórico e recomendações de PostgreSQL para MongoDB. Vocês devem analisar: quais foram os ganhos de performance? Quais as limitações? Quais os desafios de sincronização?

Passos da Atividade (20-25 minutos)

1. **Entender o Antes (SQL)**
   Revisar a query SQL original: 6 JOINs para carregar o feed. Latência: 500-800ms. Problema: escala de milhões de usuários.

2. **Entender o Depois (NoSQL)**
   Revisar o documento MongoDB: dados denormalizados em um único documento. Latência: 10-50ms. Ganho: 50x mais rápido.

3. **Analisar Ganhos, Limitações e Desafios**
   Preencher a tabela: Ganhos (latência, escalabilidade), Limitações (redundância, custo), Desafios (eventual consistency, sincronização).

4. **Propor Solução para Sincronização**
   Como garantir que o nome do usuário (em SQL) e o ID (em MongoDB) permaneçam sincronizados? Usar Event-Driven? Batch? Real-time?

Análise: Ganhos vs Limitações vs Desafios

GANHOS
- Latência: 800ms → 50ms
- Escalabilidade horizontal
- Flexibilidade de schema
- Melhor UX

LIMITAÇÕES
- Redundância de dados
- Maior consumo de storage
- Complexidade de código
- Custo de sincronização

DESAFIOS
- Eventual Consistency
- Sincronização de dados
- Perda de Foreign Keys
- Validação em aplicação

Cenários Adicionais: Varejo e Clínica

E-commerce (Varejo)

CONTEXTO

Plataforma de vendas online com milhões de usuários. Carrinho de compras é acessado 100x mais que checkout.

PROBLEMA SQL

Query: SELECT p.name, p.price, sc.quantity FROM shopping_cart sc JOIN products p... (3 JOINs). Latência: 150-200ms. Escala: 1M de carrinhos simultâneos = servidor sobrecarregado.

SOLUÇÃO NOSQL

Carrinho como documento MongoDB ou chave-valor Redis com TTL. Uma leitura. Dados denormalizados: product_id, name, price, quantity. Latência: 5-20ms.

GANHOS E DESAFIOS

✓ GANHOS

Latência 30x menor. Checkout mais rápido. Menos abandono.

⚠ DESAFIOS

Preço muda em SQL, carrinho fica desatualizado. Sincronização necessária.

Clínica (Saúde)

CONTEXTO

Sistema de prontuário eletrônico. Médico precisa acessar histórico completo do paciente rapidamente.

PROBLEMA SQL

Query: SELECT * FROM patients p JOIN appointments a JOIN medical_records mr... (5 JOINs). Latência: 250-400ms. Médico espera 400ms para ver histórico = ruim.

SOLUÇÃO NOSQL

Prontuário como documento MongoDB com histórico completo embutido. Consultas, prescrições, exames em um lugar. Uma leitura. Latência: 30-80ms.

GANHOS E DESAFIOS

✓ GANHOS

Latência 5x menor. Médico acessa histórico em 50ms. LGPD: CPF mascarado.

⚠ DESAFIOS

Dados do médico redundantes. Se nome muda, atualizar em todos os prontuários. Complexo.

Desafios e Vantagens do NoSQL no Projeto Integrador

Vantagens Potenciais

Carrinho de Compras Rápido

Carrinho como chave-valor (Redis) ou documento (MongoDB). Uma leitura em vez de 3-4 JOINs. Latência: 50ms vs 200ms.

Projeto: E-commerce. Benefício: Checkout 4x mais rápido.

Histórico de Paciente Completo

Prontuário como documento único. Todas as consultas, prescrições e exames em um lugar. Leitura única em vez de 5 JOINs.

Projeto: Clínica. Benefício: Médico acessa histórico em 50ms vs 300ms.

Catálogo Dinâmico

Adicionar campos (tags, reviews, recomendações) sem ALTER TABLE. Schema flexível. Deploy instantâneo.

Projeto: Biblioteca. Benefício: Novos campos sem downtime.

Desafios Práticos

Sincronização de Dados

Dados redundantes em SQL e NoSQL. Se preço muda em SQL, precisa atualizar em NoSQL. Eventual Consistency.

Desafio: Garantir que carrinho no Redis tem preço correto do SQL. Código extra necessário.

Validações na Aplicação

SQL tem Foreign Keys automáticas. NoSQL não. Integridade deve ser gerenciada pela aplicação. Mais código, mais bugs.

Desafio: Garantir que customer_id no carrinho existe no SQL. Validação manual necessária.

Redundância de Dados

Documentos denormalizados = dados duplicados. Nome do produto em múltiplos documentos. Manutenção complexa.

Desafio: Se nome do produto muda, atualizar em todos os carrinhos. Operação custosa.

Atividade 2: Reflexão Individual sobre Projeto Integrador

Contexto: Vocês implementaram o projeto integrador da UC (Clínica, E-commerce ou Biblioteca) usando SQL (modelo relacional). Agora, reflitam: como um banco NoSQL (MongoDB, DynamoDB) poderia ter melhorado o projeto? Quais dados se beneficiariam de NoSQL? Quais desafios enfrentariam?

Instruções para Escrever o Parágrafo (10-15 minutos)

1. **Escolher um Recurso:** Selecione um recurso do seu projeto que foi implementado em SQL (ex: carrinho de compras, histórico de consultas, catálogo de livros).
2. **Analisar o Modelo SQL:** Quantas tabelas? Quantos JOINs? Qual a latência esperada? Qual o problema enfrentado?
3. **Propor Solução NoSQL:** Como esse recurso ficaria em MongoDB ou DynamoDB? Qual a estrutura do documento? Qual a latência esperada?
4. **Contrastar Vantagens e Desafios:** Qual seria o ganho de performance? Qual o desafio de sincronização ou redundância?

ESTRUTURA SUGERIDA DO PARÁGRAFO

"No projeto X, implementamos SQL para garantir a integridade do [Dado Transacional]. Contudo, para o recurso [Dado de Alto Volume/Variável], um banco NoSQL (como MongoDB/DynamoDB) traria a vantagem de [Ganho de Velocidade], mas exigiria que a aplicação gerenciasse o desafio da [Garantia da Consistência de Dados]."

EXEMPLO DE PARÁGRAFO BEM ESCRITO

"No projeto de E-commerce, implementamos SQL para garantir a integridade do faturamento e pedidos finalizados. Contudo, para o carrinho de compras (que muda frequentemente), um banco NoSQL como Redis traria a vantagem de latência reduzida (50ms vs 200ms), permitindo checkout mais rápido. Porém, exigiria que a aplicação gerenciasse o desafio de sincronizar o preço do carrinho com o preço atual no SQL, evitando inconsistências."

CHECKLIST: SEU PARÁGRAFO DEVE CONTER

- ☐ Nome do projeto (Clínica/E-commerce/Biblioteca)
- ☐ Problema com SQL (latência, JOINs)
- ☐ Ganho de performance esperado
- ☐ Recurso específico analisado
- ☐ Solução NoSQL proposta
- ☐ Desafio de sincronização/redundância

Debate: Quando Complexidade SQL > Redundância NoSQL?

Em que momento a complexidade de gerenciar um schema relacional complexo (muitos JOINs, validações) se torna mais custosa do que gerenciar redundância em NoSQL?

Complexidade SQL ALTA

Quando SQL fica custoso

Muitos JOINs (5+)
Query precisa de 5-6 JOINs. Cada JOIN adiciona latência. Em escala de milhões, insuportável.

Schema Mutável
Novos campos aparecem frequentemente. ALTER TABLE em produção é arriscado. Downtime necessário.

Validações Complexas
Regras de negócio complexas. Foreign Keys não cobrem tudo. Lógica na aplicação.

Alto Tráfego de Leitura
Milhões de leituras/segundo. Servidor SQL não aguenta. Escalabilidade vertical é limite.

Redundância NoSQL ALTA

Quando NoSQL fica custoso

Dados Muito Redundantes
Mesmo dado em múltiplos documentos. Se muda, precisa atualizar em todos. Operação custosa.

Eventual Consistency Crítica
Dados precisam estar SEMPRE corretos (faturamento, transações). Eventual Consistency não é aceitável.

Sincronização Complexa
Manter SQL e NoSQL sincronizados é complexo. Event-Driven? Batch? Real-time? Código extra.

Relatórios com JOINs
Relatórios precisam de dados de múltiplas fontes. NoSQL não é bom para isso. SQL é melhor.

Questões para Debate em Grupo

❓ No seu projeto integrador (Clínica/E-commerce/Biblioteca), em que momento SQL ficaria muito complexo? Qual seria o gatilho para migrar para NoSQL?
❓ Como você mediria o custo de complexidade SQL vs custo de redundância NoSQL? Qual métrica usaria?
❓ Se você tivesse que escolher entre "muitos JOINs lentos" e "dados redundantes", qual escolheria? Por quê?
❓ A Arquitetura Híbrida (SQL + NoSQL) resolve o problema? Ou apenas adia a decisão?

Conclusão: Síntese da UC3 e Tomada de Decisão

1. **Modelagem Conceitual**
   Entidades, Relacionamentos, Normalização
2. **DDL/DML**
   CREATE, INSERT, Update, Delete, Queries
3. **Segurança e Governança**
   Permissões, Criptografia, Conformidade
4. **Data Lake e Nuvem**
   Arquitetura, Particionamento, Pipelines
5. **NoSQL e Híbrido**
   Escalabilidade, Flexibilidade, Trade-offs

A habilidade final de um Engenheiro de Dados é escolher a tecnologia certa para o problema certo

APRENDIZADOS CONSOLIDADOS

- ✓ SQL para dados transacionais e ACID
- ✓ Governança garante confiabilidade
- ✓ NoSQL para escala e velocidade
- ✓ Data Lakes centralizam dados brutos
- ✓ Arquitetura Híbrida é a realidade
- ✓ Pipelines orquestram transformações

REFLEXÃO FINAL: RESPONSABILIDADE DO ENGENHEIRO DE DADOS

*Você aprendeu a **modelar dados**, **implementar bancos de dados**, **garantir segurança**, **escalar em nuvem** e **escolher entre SQL e NoSQL**. Agora você tem as ferramentas para construir sistemas de dados que são **confiáveis, seguros, escaláveis e impactam realmente o negócio**. Use esse conhecimento com responsabilidade.*