UniCeub – Centro Universitário de Brasília  
  
Disciplina: Banco de Dados II  
Professor: Deusdeth Mariano  
  
**Banco de Dados NoSQL: Flexibilidade e Desempenho em Sistemas de Armazenamento Não Relacional**Aluno: Vitor Davi Batista Paulo  
Curso: Análise e Desenvolvimento de Sistemas - 2º semestre  
  
Brasília - 2025

# 1. Introdução

Neste trabalho eu apresento uma visão geral sobre bancos de dados NoSQL, buscando explicar por que eles surgiram, como funcionam e em que situações fazem mais sentido do que bancos relacionais tradicionais. Durante as primeiras disciplinas do curso foi dito que os fundamentos dos modelos relacionais, modelagem ER e DDL, logo isso me ajudou a entender melhor as diferenças práticas entre os dois mundos. As instruções desta atividade e os tópicos obrigatórios orientaram a realizar sua estrutura.

# 2. Fundamentos de NoSQL

O termo NoSQL é frequentemente explicado como 'Not Only SQL'. Na prática, trata-se de um conjunto de sistemas de gerenciamento de banco de dados que não seguem o modelo relacional de forma rigorosa. Eles se tornaram mais conhecidos a partir da década de 2000, quando empresas passaram a lidar com volumes exorbitantes de dados e a exigir alta disponibilidade e escalabilidade horizontal. Os bancos NoSQL diferenciam-se dos relacionais principalmente por possibilitarem esquemas flexíveis, por oferecerem modelos de dados variados e por priorizarem desempenho em múltiplos **nós** (cenários distribuídos).  
  
Resumo dos principais tipos:  
• Chave‑valor: armazenam pares chave/valor e são ideais para caches e sessões (ex.: Redis).  
• Documentos: guardam documentos em JSON/BSON; permitem consultas por campos e são úteis em apps web (ex.: MongoDB).  
• Colunar: organizam dados por colunas para acelerar agregações em grandes volumes (ex.: Cassandra).  
• Grafos: modelam relações complexas entre entidades, indicados para redes sociais e recomendações (ex.: Neo4j).  
  
Vantagens importantes incluem a escalabilidade horizontal, a flexibilidade de esquema e o desempenho em leituras e escritas massivas. Entre as limitações estão a menor uniformidade de ferramentas e padrões (comparado ao SQL), desafios em garantir transações ACID completas em todos os modelos e a necessidade de projetar a aplicação de forma que compense a falta de relacionamentos e normalização automáticos.

# 3. Relação com OLTP e OLAP

OLTP (Online Transaction Processing) e OLAP (Online Analytical Processing) são dois modos de uso dos dados que, na prática, exigem soluções diferentes. Enquanto o OLTP está relacionado ao dia a dia do sistema, registro de vendas, atualizações de estoque, cadastros e pagamentos — o OLAP é para análises que ajudam a tomar decisões, como relatórios de vendas por período, análise de comportamento de clientes e comparação de desempenho entre lojas.

No meu entendimento, o que separa os dois é a exigência de latência e garantia: aplicações OLTP precisam responder rápido e com consistência (por exemplo, não dá para vender o mesmo item duas vezes). Já o OLAP aceita demoras maiores porque trabalha com conjuntos de dados maiores e agregações — geralmente dá para usar dados um pouco defasados quando o objetivo é observar tendências mensais.

Os bancos relacionais nasceram já pensando nas tarefas do dia a dia — vendas, atualizações de estoque, cadastros — então, nas primeiras aulas, a gente viu que modelagem ER, DDL e as constraints existem para garantir que os dados sigam as regras do negócio e não virem uma bagunça. O NoSQL apareceu depois, quando as empresas precisaram escalar para várias máquinas e lidar com dados que mudam de formato o tempo todo. Não quer dizer que NoSQL não faça transações; hoje alguns bancos cuidam de operações transacionais simples e até de coisas mais complexas, só que o jeito de projetar é diferente. Por exemplo: um cache chave-valor como o Redis é ótimo para a sessão e contadores; já um banco de documentos, tipo MongoDB, facilita salvar o perfil do usuário mesmo quando os campos vão aparecendo ou sumindo conforme o app evolui.

Para OLAP, bancos colunares ou soluções específicas de data warehouse (como Snowflake, BigQuery, ClickHouse) costumam ser preferidos, porque são otimizados para leitura e agregação massiva. Em empresas menores, é comum ver um ETL simples que copia os dados do sistema transacional para uma base analítica; em empresas maiores usa-se pipelines com streaming (Kafka, por exemplo) e arquiteturas Lambda/Kappa para processar eventos quase em tempo real e alimentar dashboards e relatórios.

Na prática, o que eu tenho visto (e que faz sentido com o que estudamos) é uma mistura de tecnologias: o SGBD relacional cuida das transações críticas — pagamentos, cadastros, integridade do negócio — enquanto um banco NoSQL armazena logs, eventos do usuário, sessões e até alguns dados para análise. Isso isola cargas: o OLTP fica rápido e consistente; o analítico pode escalar, fazer agregações pesadas e nem sempre precisa das mesmas garantias de consistência imediata. Um padrão bem comum é usar CDC (Change Data Capture) — com ferramentas tipo Debezium — para espelhar mudanças do banco relacional em um broker ou em uma base analítica, garantindo que relatórios e dashboards recebam os dados mais recentes sem sobrecarregar o OLTP.

Também é importante entender trade-offs teóricos e práticos, como o teorema CAP: em sistemas distribuídos há escolhas entre consistência, disponibilidade e tolerância a partições. Muitas arquiteturas NoSQL priorizam disponibilidade e particionamento para suportar falhas e alta carga, o que pode levar a consistência eventual — aceitável em alguns cenários (ex.: feeds de redes sociais), mas inaceitável em transações financeiras. Por isso, projetar para o tipo de aplicação é essencial: se a operação exige ACID, o relacional ainda é a escolha mais segura; se o requisito é escalar bilhões de eventos por dia, NoSQL costuma ser mais prático.

Algumas práticas que eu acho importante citar, porque ajudam a construir sistemas híbridos mais robustos:  
 • usar réplicas de leitura para escalar consultas de relatório sem afetar gravações críticas;  
 • aplicar índices e *materialized views* no lado analítico para acelerar agregações;  
 • checar o suporte a transações no NoSQL escolhido (por exemplo, versões recentes do MongoDB suportam transações multi-documento);  
 • planejar processos de sincronização (ETL/ELT, CDC) para que os dados analíticos sejam consistentes o suficiente para tomada de decisão;  
 • modelar os dados pensando nas consultas: NoSQL geralmente exige denormalização proposital para evitar muitas junções.

# 4. Comparação técnica entre PostgreSQL e MongoDB

Para comparar tecnicamente, escolhi o PostgreSQL como modelo relacional e o MongoDB como modelo NoSQL, pois ambos são populares e bem documentados.  
  
Modelo de dados: PostgreSQL usa tabelas normalizadas, chaves e relações; isso facilita integridade referencial e modelos ER (tema dos nossos slides). MongoDB armazena documentos JSON, o que facilita representar objetos aninhados sem várias junções.  
  
Linguagem de consulta: PostgreSQL usa SQL, uma linguagem padronizada que conhecemos no começo da matéria de banco de dados II. MongoDB oferece uma API de consultas em JSON e agregação que pode ser mais espontâneo para dados semi‑estruturados.  
  
Escalabilidade: PostgreSQL escala verticalmente (mais memória/CPU) e possuí opções de replicação e sharding, mas geralmente demanda mais planejamento. MongoDB foi desenhado na escalabilidade horizontal, facilitando distribuir dados entre nós.  
  
Esquema e flexibilidade: PostgreSQL exige definição de esquema (DDL, constraints), o que oferece a previsibilidade. MongoDB permite documentos com campos diferentes na mesma coleção, o que agiliza iterações rápidas.  
  
Transações e consistência: PostgreSQL implementa ACID forte por padrão. MongoDB oferece transações multi‑documento, mas arquiteturalmente muitos deployments priorizam disponibilidade, o que pode facilitar garantias de consistência (considere o modelo CAP ao projetar sistemas distribuídos).  
  
Desempenho: em leituras simples e massivas, MongoDB pode oferecer melhor throughput; para consultas complexas, relacionamentos e joins, PostgreSQL costuma ser mais eficiente e previsível.  
  
Conclusão técnica: a escolha depende do caso de uso. Se a aplicação precisa de integridade rígida e consultas relacionais, PostgreSQL é melhor. Se o desenvolvimento exige agilidade no esquema, alta escalabilidade e trabalho com documentos, MongoDB é vantajoso.

# 5. Conclusão

Ao finalizar, reafirmo que NoSQL e bancos relacionais têm papéis complementares. Aprendi nos slides da disciplina que o modelo relacional fornece garantias e ferramentas (modelagem ER, DDL e integridade) que são essenciais para muitos sistemas; por outro lado, NoSQL surge como resposta a problemas práticos de escala.  
  
Na prática, uma decisão técnica deve considerar requisitos não só técnicos (consistência, latência, escalabilidade), mas também organização, custo e experiência da equipe. Para mim, que estou no 2º semestre de ADS, entender essas diferenças e saber indicar quando usar cada tecnologia é interessante para atuar em projetos reais.  
  
Sugiro que o leitor (e eu, ao aplicar esse conhecimento) avalie cenários reais antes de optar por um SGBD, e que documente as decisões arquiteturais para melhorar as manutenções futuras.

# Referências

SADALAGE, P. J.; FOWLER, M. NoSQL Distilled. Addison‑Wesley, 2012.  
MONIRUZZAMAN, A. B. M.; HOSSAN, S. NoSQL Database: New Era of Databases for Big Data Analytics. Int. Journal of Database Theory and Application, 2013.  
MONGODB, Inc. MongoDB Manual. Disponível em: https://www.mongodb.com/docs/. Acesso em: 23 set. 2025.  
POSTGRESQL Global Development Group. PostgreSQL Documentation. Disponível em: https://www.postgresql.org/docs/. Acesso em: 23 set. 2025.  
Slides e material do professor Deusdeth Mariano: Introdução a BD e roteiro da atividade (FATECS).