UNIVERSIDADE ESTADUAL DO MARANHÃO
CENTRO DE CIÊNCIAS TECNOLÓGICAS
CURSO DE ENGENHARIA DE COMPUTAÇÃO

GUSTAVO SILVEIRA PONTES

APLICABILIDADE E DESEMPENHO DE BANCO DE DADOS NOSQL EM APLICAÇÕES MODERNAS: UM ESTUDO DE CASO EM MONGODB, CASSANDRA, NEO4J E REDIS

SÃO LUÍS - MA
2025

UNIVERSIDADE ESTADUAL DO MARANHÃO
CENTRO DE CIÊNCIAS TECNOLÓGICAS
CURSO DE ENGENHARIA DA COMPUTAÇÃO

GUSTAVO SILVEIRA PONTES

APLICABILIDADE E DESEMPENHO DE BANCO DE DADOS NOSQL EM APLICAÇÕES MODERNAS: UM ESTUDO DE CASO EM MONGODB, CASSANDRA, NEO4J E REDIS

Trabalho de Conclusão de Curso apresentado ao curso de Graduação em Engenharia de Computação na Universidade Estadual do Maranhão como requisito parcial para obtenção do título de Bacharel em Engenharia de Computação, sob orientação do Prof. Dr. Luís Carlos Costa Fonseca.

SÃO LUÍS - MA
2025

Pontes, Gustavo Silveira
Aplicabilidade e desempenho de banco de dados NoSQL em aplicações modernas: um estudo de caso em MongoDB, Cassandra, Neo4j e redis / Gustavo Silveira Pontes. – São Luis, MA, 2025.
40 f

TCC (Graduação em Engenharia de Computação) - Universidade Estadual do Maranhão, 2025.

Orientador: Prof. Dr. Luís Carlos Costa Fonseca.

1.NoSQL. 2.Desempenho. 3.MongoDB. 4.Cassandra. 5.Neo4j - 6.Redis. I.Titulo.

CDU: 004.65

Elaborado por Cássia Diniz - CRB 13/910

GUSTAVO SILVEIRA PONTES

APLICABILIDADE E DESEMPENHO DE BANCO DE DADOS NOSQL EM APLICAÇÕES MODERNAS: UM ESTUDO DE CASO EM MONGODB, CASSANDRA, NEO4J E REDIS

SÃO LUÍS, dia 31 de julho de 2025:

Prof. Dr. Luís Carlos Costa Fonseca

Orientador - UEMA

Prof. Pedro Brandão Neto

Examinador Interno - UEMA

Documento assinado digitalmente
gov.br REINALDO DE JESUS DA SILVA
Data: 23/08/2025 21:35:46-0300
Verifique em https://validar.iti.gov.br

Prof. Reinaldo de Jesus da Silva

Examinador Interno - UEMA

SÃO LUÍS - MA

2025

AGRADECIMENTOS

Agradeço à Universidade Estadual do Maranhão (UEMA) e ao corpo docente do curso de Engenharia de Computação, por todo o conhecimento transmitido ao longo destes anos de graduação, que formaram a base para o desenvolvimento desta pesquisa. Um agradecimento especial ao professor Carlos Henrique Rodrigues de Oliveira, por sua excelência profissional e pela forma inspiradora com que conduziu suas disciplinas, que foram fundamentais para a minha formação.

À minha família, em especial aos meus pais, por todo o amor, apoio incondicional, incentivo e, principalmente, pela paciência durante as longas horas de estudo e dedicação que este trabalho exigiu. Sem vocês, esta conquista não seria possível.

A um amigo, pelo companheirismo, mesmo depois de todos os problemas que enfrentamos ao longo da jornada pude contar com seus conselhos e ideias. A jornada foi melhor com a sua ajuda.

RESUMO

O crescimento exponencial do volume e da complexidade dos dados na era do Big Data impulsionou a adoção de bancos de dados NoSQL, que oferecem alta escalabilidade, flexibilidade e performance. Contudo, a diversidade de modelos – Documento, Colunar, Grafo e Chave-Valor – apresenta um desafio na escolha da tecnologia adequada para cada aplicação. Este trabalho apresenta um estudo de caso comparativo sobre a aplicabilidade e o desempenho de quatro proeminentes bancos de dados NoSQL: MongoDB, Cassandra, Neo4j e Redis. Para a análise, foi desenvolvida uma aplicação em Python com uma *API* (*FastAPI*) e uma interface de visualização (Streamlit), executada em um ambiente dockerizado. Foram realizados testes de performance em quatro cenários distintos: escrita em lote, leitura com filtros complexos, agregação de dados e travessia de relacionamentos. Os resultados demonstram que não há uma solução universalmente superior, e que a performance de cada banco está intrinsecamente ligada à sua arquitetura e à carga de trabalho específica, fornecendo assim insights práticos para o desenvolvimento de aplicações modernas.

Palavras-chave: NoSQL; Desempenho; MongoDB; Cassandra; Neo4j; Redis; Estudo de Caso.

ABSTRACT

The exponential growth in data volume and complexity in the Big Data era has driven the adoption of NoSQL databases, which offer high scalability, flexibility, and performance. However, the diversity of models—such as Document, Columnar, Graph, and Key-Value—presents a challenge in choosing the appropriate technology for each type of application. This work presents a comparative case study on the applicability and performance of four prominent NoSQL databases: MongoDB, Cassandra, Neo4j, and Redis. For this analysis, an application was developed in Python, using the FastAPI *framework* for the API and Streamlit for the visualization interface, running in a dockerized environment. Performance tests were conducted across four distinct scenarios: batch writing, complex filtered reads, data aggregation, and relationship traversal. The results demonstrate that there is no one-size-fits-all solution, and the performance of each database is intrinsically linked to its architecture and the specific workload, thus providing practical insights for the development of modern applications.

Keywords: NoSQL; Performance; MongoDB; Cassandra; Neo4j; Redis; Case Study.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API - *Application Programming Interface* (Interface de Programação de Aplicações)

BSON - *Binary JavaScript Object Notation*

CPU - *Central Processing Unit* (Unidade Central de Processamento)

CRUD - *Create, Read, Update, Delete* (Criar, Ler, Atualizar, Deletar)

I/O - *Input/Output* (Entrada/Saída)

ID - Identificador

IMDb - *Internet Movie Database*

JIT - *Just-In-Time* (Compilação em Tempo de Execução)

JSON - *JavaScript Object Notation*

LPG - *Labeled Property Graph* (Grafo de Propriedades Rotulado)

LSM - *Log-Structured Merge-Tree*

NoSQL - *Not Only SQL*

RAM - *Random Access Memory* (Memória de Acesso Aleatório)

s - Segundos

SSD - *Solid-State Drive* (Unidade de Estado Sólido)

TCC - Trabalho de Conclusão de Curso

UEMA - Universidade Estadual do Maranhão

WSL2 - *Windows Subsystem for Linux 2*

SUMÁRIO

1. INTRODUÇÃO

A era do Big Data, marcada pelo crescente volume e complexidade das informações, rapidamente expôs as limitações dos bancos de dados relacionais tradicionais. Para responder a essa nova demanda por alta performance, flexibilidade e escalabilidade, uma nova geração de tecnologias, conhecidas como bancos de dados *NoSQL*, tornou-se peça fundamental no ecossistema de aplicações modernas. Este capítulo introdutório, portanto, estabelece as bases desta pesquisa, apresentando a contextualização do problema, a justificativa de sua relevância e os objetivos que guiaram o desenvolvimento do presente trabalho.

1.1. Contextualização

Após a década de 1990, o cenário tecnológico testemunhou um crescimento exponencial no volume, variedade e velocidade dos dados, um fenômeno impulsionado pela ascensão da Web 2.0 e pela popularização de aplicações interativas e redes sociais. Esse novo paradigma, conhecido como Big Data, expôs as limitações dos sistemas de gerenciamento de bancos de dados relacionais (SGBDRs), que eram tradicionalmente utilizados para dados estruturados e consistência forte. A demanda por maior escalabilidade, flexibilidade no esquema de dados e alta performance para lidar com dados não estruturados e semiestruturados impulsionou o desenvolvimento e a adoção massiva de uma nova categoria de bancos de dados: os *NoSQL*.

Diferente dos bancos relacionais, os sistemas NoSQL foram projetados com o intuito de oferecer lidar com grandes volumes de dados não estruturados e semiestruturados, operando sem a necessidade de junções complexas e esquemas rígidos, o que os torna ideais para aplicações modernas que demandam alta performance (McCREARY e KELLY, 2013, p. 4-5).

1.2. Problema de Pesquisa

A proliferação de bancos de dados NoSQL resultou em um ecossistema diversificado, com cada tecnologia possuindo uma arquitetura e um modelo de dados distintos – como Documento (MongoDB), Família de Colunas (Cassandra), Grafo (Neo4j) e Chave-Valor (Redis). Essa variedade, embora poderosa, apresenta um desafio significativo para arquitetos de software e desenvolvedores: a escolha da ferramenta correta para uma determinada carga de

trabalho. A performance de uma aplicação pode ser drasticamente afetada pela seleção de um banco de dados cuja arquitetura não é otimizada para os padrões de acesso e manipulação de dados predominantes no sistema. Diante disso, emerge a seguinte questão de pesquisa: **Como o modelo de dados e a arquitetura de diferentes bancos NoSQL impactam seu desempenho e aplicabilidade em cenários de uso comuns, como escrita em lote, buscas complexas e agregações analíticas?**

1.3. Justificativa

Apesar da vasta literatura teórica sobre as características de cada banco de dados NoSQL, há uma carência de estudos de caso práticos que comparem essas tecnologias de forma empírica e controlada, utilizando uma mesma aplicação e base de dados. Este trabalho se justifica pela necessidade de fornecer uma análise de desempenho concreta e aplicada, que sirva como um guia prático para estudantes e profissionais da área de tecnologia. Ao construir uma aplicação funcional e submeter os bancos de dados a testes de performance idênticos, este projeto gera métricas e insights valiosos que podem auxiliar na tomada de decisões técnicas em projetos reais, reduzindo incertezas e otimizando a arquitetura de novas aplicações.

1.4. Objetivos

Diante do cenário e do problema exposto, este trabalho tem como meta principal avaliar e comparar a performance e a aplicabilidade de quatro bancos de dados NoSQL líderes de mercado.

1.4.1. Objetivo Geral

O objetivo geral deste trabalho é realizar um estudo de caso comparativo sobre a aplicabilidade e o desempenho de MongoDB, Cassandra, Neo4j e Redis em diferentes cenários de carga de trabalho, emulando operações de uma aplicação moderna.

1.4.2. Objetivos Específicos

Para alcançar o objetivo geral, os seguintes objetivos específicos foram traçados:

- Desenvolver uma aplicação de teste com *backend* em *FastAPI* e *frontend* em *Streamlit*, em um ambiente dockerizado[1], para interagir com os bancos de dados;
- Modelar e realizar a carga de uma base de dados do IMDb [2], adaptando a estrutura dos dados para o modelo ideal de cada um dos quatro bancos;
- Executar um conjunto de testes de desempenho controlados, medindo o tempo de resposta para operações de escrita em lote, leitura com filtros complexos, agregação e busca por *relacionamento*;
- Analisar e discutir os resultados de performance obtidos, correlacionando-os com a arquitetura fundamental de cada banco de dados para extrair conclusões sobre sua adequação a cada cenário.

[1] O termo significa que um aplicativo ou serviço foi configurado para funcionar dentro de um contêiner Docker.
[2] Lançado online em 1990 e uma subsidiária da Amazon.com desde 1998, o IMDb *(Internet Movie Database*) é a fonte mais popular e confiável do mundo para conteúdo de filmes, TV e celebridades, projetado para ajudar os fãs a explorar o mundo dos filmes e programas e decidir o que assistir. Disponível em: https://help.imdb.com/article/imdb/general-information/what-is-imdb/. Acesso em: 09 jun. 2025.

2. FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta a fundamentação teórica que serve como alicerce para o presente estudo. Inicialmente, são abordados os conceitos centrais dos bancos de dados NoSQL, incluindo seus diferentes paradigmas e características principais. Em seguida, são descritas as arquiteturas e particularidades de cada um dos quatro bancos de dados selecionados para a análise comparativa — MongoDB, Cassandra, Neo4j e Redis —, bem como as tecnologias utilizadas no desenvolvimento da aplicação de testes.

2.1. Bancos de Dados NoSQL

Os sistemas NoSQL foram projetados com o intuito de lidar com grandes volumes de dados não estruturados e semiestruturados, operando sem a necessidade de junções complexas e esquemas rígidos, o que os torna ideais para aplicações modernas que demandam alta performance (McCREARY e KELLY, 2013, p. 4-5).

Nesse contexto, a arquitetura dos bancos de dados NoSQL permite a distribuição e replicação de dados entre múltiplos *nós*, garantindo uma alta disponibilidade e menos chances de falhas. Essa capacidade de escalabilidade horizontal é um dos principais motivos pelos quais grandes empresas de e-commerce e redes sociais optam por esse tipo de banco de dados, uma vez que essas aplicações requerem gerenciamento eficiente de dados de usuários e transações em larga escala. (McCREARY e KELLY, 2013, pp. 62-95; pp. 127-153).

Esses bancos oferecem uma certa flexibilidade na *modelagem de dados*, permitindo que os esquemas sejam ajustados dinamicamente sem paralisação. Outrossim, são altamente escaláveis horizontalmente, podendo expandir a capacidade de armazenamento e processamento através da adição de mais servidores na rede. Logo, essa natureza distribuída possibilita operações de leitura e escrita rápidas, importantes para aplicações que exigem respostas em tempo real (McCREARY e KELLY, 2013, pp. 62-95).

De acordo com McCreary e Kelly (2013, p.5):

> Os sistemas NoSQL armazenam e recuperam dados em diversos formatos: lojas de chave-valor, bancos de dados de grafos, armazenamentos de família de colunas (como *Bigtable*), bancos de dados de documentos e até mesmo linhas em tabelas, diferenciando se entre si conforme os requisitos da aplicação a ser utilizada.

Os bancos de dados NoSQL podem ser categorizados em quatro tipos principais, de acordo com sua arquitetura e modelo de armazenamento:

- Bancos de dados de documentos: Armazenam informações em documentos no formato *JSON* ou *BSON*, permitindo flexibilidade no esquema. São ideais para aplicações que exigem armazenamento dinâmico, como gerenciamento de conteúdo e *Internet of Things* (IoT[3]), onde diferentes tipos de dados podem ser armazenados sem comprometer a performance (AMAZON WEB SERVICES, 2024). Exemplos de uso do MongoDB estão presentes em empresas como a Square Enix, Eletronic Arts e Adobe, devido à sua capacidade de escalar horizontalmente e gerenciar grandes volumes de dados (MONGODB, 2024).
- Bancos de dados de colunas: Os bancos de dados orientados por coluna, ou colunas largas, armazenam dados organizados como um conjunto de colunas, e suas colunas podem variar entre as linhas dentro de uma única tabela (GOOGLE CLOUD, 2024). Empresas como a Activision, Apple, eBay e Discord utilizam o Apache Cassandra por necessitarem de um sistema robusto para gerenciar grandes volumes de dados com alta disponibilidade (APACHE CASSANDRA, 2024).
- Bancos de dados de grafos: Projetados para representar dados como uma rede de entidades e suas conexões, usando a teoria dos grafos. Oferecem maior flexibilidade e eficiência para modelar relações complexas no mundo real. (AMAZON WEB SERVICES, 2024). Empresas como a NASA, BMW e Intel utilizam o Neo4j para analisar relacionamentos complexos em seus dados (NEO4J, 2024).
- Bancos de dados chave-valor: Armazenam dados como pares de chave e valor, onde a chave é um identificador único. Destacam-se por serem altamente escaláveis e distribuídos, facilitando o armazenamento e a recuperação eficiente de dados (AMAZON WEB SERVICES, 2024). Empresas como a IBM, Uber, Unity e Waze utilizam o Redis para cenários que exigem acesso de baixíssima latência (REDIS, 2024).

[3] O termo IoT, ou Internet das Coisas, refere-se à rede coletiva de dispositivos conectados e à tecnologia que facilita a comunicação entre os dispositivos e a nuvem, bem como entre os próprios dispositivos.

2.2. MongoDB

O MongoDB é um banco de dados NoSQL orientado a documentos projetado para oferecer flexibilidade na modelagem de dados e escalabilidade (BRADSHAW; BRAZIL; CHODOROW, 2019). Diferente dos bancos de dados relacionais, ele não exige um esquema rígido, permitindo que documentos dentro de uma mesma coleção tenham estruturas diferentes, fornecendo adaptabilidade a mudanças nos requisitos da aplicação (BRADSHAW; BRAZIL; CHODOROW, 2019).

Adicionalmente, sua arquitetura foi desenvolvida com foco em desempenho, oferecendo recursos robustos para otimizar operações de leitura e escrita. Destacam-se o seu poderoso sistema de *índices secundários*, que permite a recuperação eficiente de dados com base em qualquer campo do documento, e o *Aggregation Framework*, um conjunto de ferramentas que possibilita a execução de análises de dados complexas diretamente no servidor (BRADSHAW; BRAZIL; CHODOROW, 2019).

2.3. Cassandra

O Apache Cassandra é um banco de dados NoSQL distribuído, descentralizado e altamente escalável, projetado para lidar com grandes volumes de dados em aplicações modernas (CARPENTER e HEWITT, 2022, p. 17). Ele combina o modelo de distribuição do *Dynamo*, da Amazon com o modelo de dados do Bigtable, do Google, permitindo que os dados sejam armazenados e acessados, com uma ótima performance, em diversos nós da rede (Carpenter e Hewitt, 2022, p. 18).

Dessa forma, o Cassandra foi projetado para alta disponibilidade e tolerância a falhas, garantindo que a aplicação continue funcionando mesmo em caso de falhas em alguns nós do *cluster* (CARPENTER e HEWITT, 2022, p. 19). Assim, possui escalabilidade elástica, que possibilita a adição de novos nós ao *cluster* sem comprometer o desempenho, tornando-o adequado para aplicações que precisam crescer rapidamente (CARPENTER e HEWITT, 2022, p. 19). Além disso, o Cassandra oferece consistência ajustável, permitindo que desenvolvedores escolham entre maior disponibilidade ou maior consistência, dependendo da necessidade da aplicação (CARPENTER e HEWITT, 2022, p. 20).

2.4. Neo4j

O Neo4j é um banco de dados de grafos que excela na modelagem de dados complexos e interconectados, tornando-se essencial para análises que dependem da compreensão profunda de relações entre entidades (EIFREM, ROBINSON e WEBBER, 2015, pp. 1-9). Sua arquitetura é projetada para uma eficaz execução de consultas relacionadas a grafos, utilizando a indexação de *propriedades* e um armazenamento otimizado para relações, possibilitando que grandes conjuntos de dados interconectados sejam navegados e analisados com um alto desempenho. Além disso, emprega mecanismos de replicação e consistência para garantir a integridade dos dados em ambientes distribuídos. (CHAUDHRY e YOUSAF, 2020, p. 15).

2.5. Redis

Por fim, temos o Redis, um armazenamento de dados em memória conhecido por sua alta performance e versatilidade em suportar diferentes tipos de estruturas de dados, ideal para aplicações que necessitam de acesso rápido aos dados, como armazenamento em cache e gerenciamento de sessões de usuários (CARLSON, 2013, pp. 3-5). Assim, devido à sua arquitetura baseada em memória, que oferece tempos de resposta ágeis, o torna adequado a cargas de trabalho que exigem baixa latência. Além disso, implementa técnicas como replicação assíncrona e persistência em disco opcional, permitindo que os dados sejam obtidos e modificados de forma célere, ao passo que mantém uma alta confiabilidade (KHAN et al., 2023, p. 12).

2.6. Tecnologias da Aplicação

Para a implementação do projeto, foram selecionadas as tecnologias Python, *Docker*, FastAPI e Streamlit. A linguagem Python foi escolhida por sua simplicidade e pelo seu robusto ecossistema para manipulação de dados. Conforme destaca Sweigart (2020), a linguagem é particularmente poderosa para a automação de tarefas, permitindo ler, processar e transformar grandes conjuntos de informações de forma programática, características que foram essenciais na etapa de pré-processamento dos dados deste trabalho. Já o Docker é uma plataforma de *conteinerização* que encapsula o processo de criação, distribuição e execução de aplicações em

contêineres, garantindo consistência entre diferentes ambientes (KANE; MATTHIAS, 2023). Por fim, o Streamlit é a *biblioteca* de *código aberto* utilizada para a criação da interface web interativa do projeto (STREAMLIT DOCUMENTATION, 2025).

3. METODOLOGIA

Este capítulo detalha os procedimentos metodológicos adotados para a realização deste estudo comparativo. Serão apresentados a arquitetura da aplicação desenvolvida para os testes, a estratégia de modelagem de dados para cada um dos quatro bancos NoSQL, os cenários de desempenho que foram executados e, por fim, os métodos utilizados para a coleta e análise das métricas de performance.

3.1. Arquitetura da Aplicação de Testes

Para conduzir uma análise comparativa de desempenho entre os quatro bancos de dados NoSQL selecionados, foi desenvolvida uma aplicação de testes em um ambiente controlado e padronizado. A linguagem Python foi escolhida como base para o desenvolvimento, devido ao seu robusto ecossistema de bibliotecas e sua plena compatibilidade com os *frameworks* FastAPI e Streamlit, adotados para a construção do backend e do frontend, respectivamente. Adicionalmente, a arquitetura foi inteiramente conteinerizada com o uso de Docker e *Docker Compose*, garantindo o isolamento dos serviços e a completa reprodutibilidade do ambiente de testes, um requisito fundamental para a validade dos resultados (HUMBLE; FARLEY, 2011, p.87).

A camada de lógica da aplicação (backend) foi implementada com o *framework* FastAPI, que atuou como o núcleo centralizador das operações de teste. Sua principal responsabilidade foi orquestrar as requisições recebidas do frontend, direcionando cada chamada para a função de *CRUD* correspondente no módulo do banco de dados apropriado. Para tal, foi estruturada uma *API* com *endpoints* específicos para cada cenário de teste, organizados através de um sistema de rotas (*routers*). A integridade e a consistência dos dados foram asseguradas pela *biblioteca Pydantic*, com a qual foram definidos *schemas* para as entidades **Filme, Ator** e **Elenco**. Essa validação garantiu que toda informação processada pela API seguisse um modelo estrutural rigoroso, fator essencial para a validade das comparações de desempenho.

A interface com o usuário (frontend) da aplicação foi desenvolvida com o *framework* Streamlit, escolhido por sua agilidade na criação de aplicações web interativas em Python. Este painel de controle desempenhou duas funções metodológicas centrais: primeiramente, viabilizou a execução manual de operações de CRUD para a verificação funcional da integração

com cada banco de dados; em segundo lugar, serviu como a plataforma para a execução controlada e padronizada dos scripts de análise de desempenho. Ressalta-se que o frontend não se comunicava diretamente com os bancos, mas sim realizava requisições HTTP para os *endpoints* da API do FastAPI, conforme descrito no parágrafo anterior, que era a responsável por processar as operações.

A fundação da arquitetura de testes reside na sua conteinerização. Utilizando um *Dockerfile* como o projeto para a imagem da aplicação e o arquivo do Docker Compose como o orquestrador, o ambiente foi segmentado em múltiplos contêineres independentes, porém conectados: um para a API, um para o frontend e um contêiner dedicado para cada um dos quatro bancos de dados. Todos os serviços foram executados em uma rede virtual customizada (*app_network*), permitindo a comunicação entre eles por meio de seus nomes de serviço. Esta abordagem garante não apenas o isolamento, mas também um ambiente de execução idêntico em qualquer máquina, tornando as medições de desempenho diretamente comparáveis e cientificamente reprodutíveis (HUMBLE; FARLEY, 2011, p. 87).

Para consolidar a arquitetura, o ciclo de vida de uma requisição de teste segue um fluxo bem definido entre as camadas da aplicação. Uma ação do usuário na interface Streamlit é processada pela camada de serviço do frontend[4], que por sua vez monta e envia uma requisição HTTP para a API. No backend, o FastAPI recebe essa requisição e a delega para a camada de serviço de negócio, responsável por orquestrar a lógica da consulta[5]. Finalmente, o serviço aciona a implementação CRUD específica do banco de dados em teste para executar a operação[6]. Toda essa estrutura modular foi organizada utilizando o padrão *src-layout*, que separa as responsabilidades do código e facilita a manutenção do projeto.

[4] A implementação do serviço de API do frontend pode ser consultada no repositório do projeto. Disponível em: https://github.com/Ozen-ok/tcc-nosql-comparativo/blob/b64bc39a01bd2e8bfbaef445f3ef07c35bb7d2f6/src/streamlit_app/services/api_service.py#L120. Acesso em: 10 jun. 2025.
[5] A lógica de orquestração do serviço de negócio pode ser consultada no repositório do projeto. Disponível em: https://github.com/Ozen-ok/tcc-nosql-comparativo/blob/b64bc39a01bd2e8bfbaef445f3ef07c35bb7d2f6/src/services/query_service.py#L563. Acesso em: 10 jun. 2025.
[6] Um exemplo de implementação de CRUD (para Neo4j) pode ser consultado no repositório do projeto. Disponível em: https://github.com/Ozen-ok/tcc-nosql-comparativo/blob/b64bc39a01bd2e8bfbaef445f3ef07c35bb7d2f6/src/databases/neo4j/crud.py#L412. Acesso em: 10 jun. 2025.

3.2. Modelagem dos Dados

A matéria-prima para a análise comparativa foi um conjunto de dados públicos disponibilizados pelo IMDb [7]. Após uma etapa de pré-processamento para limpeza e consolidação, realizada com a biblioteca *Pandas* do Python, a base de dados foi organizada em três entidades principais: **Filmes**, **Atores** e **Elenco**. O desafio metodológico central, detalhado nas subseções seguintes, consistiu em adaptar e modelar essas mesmas entidades para os paradigmas específicos de cada um dos quatro bancos de dados NoSQL avaliados.

3.2.1. MongoDB

Para a modelagem no MongoDB, optou-se por uma abordagem normalizada utilizando referências, uma estratégia que espelha a estrutura de um banco de dados relacional. Foram criadas três coleções distintas: filmes, atores e elenco. A coleção elenco atua como uma tabela de ligação, associando documentos da coleção atores com os da coleção filmes. Embora esta abordagem reduza a duplicação de dados, ela exige que a aplicação realize múltiplas consultas, ou *joins* em nível de aplicação, para recompor a informação completa de um filme e seu elenco (BRADSHAW; BRAZIL; CHODOROW, 2019). A criação de índices em campos frequentemente consultados foi implementada para otimizar essas operações de busca.

3.2.2. Cassandra

Para o Cassandra, optou-se por manter a estrutura de dados normalizada que foi utilizada no MongoDB, com o objetivo de estabelecer uma base de comparação estrutural direta entre os bancos. Desta forma, foram criadas três tabelas distintas: filmes, atores e elenco, representando cada entidade de forma separada. É crucial ressaltar que esta abordagem se distancia intencionalmente da prática idiomática do Cassandra, que preconiza o design orientado a consultas e a *desnormalização* dos dados em tabelas específicas para cada padrão de leitura (CARPENTER; HEWITT, 2022, p. 45). A consequência desta escolha metodológica é que

[7] Datasets não-comerciais do IMDb. Disponível em: https://datasets.imdbws.com/. Acesso em: 10 jun. 2025.

consultas que requerem a junção de dados necessitam ser resolvidas pela aplicação, que realiza múltiplas buscas sequenciais.

3.2.3. Neo4j

Para o Neo4j, os dados foram estruturados utilizando o modelo nativo de *Grafo de Propriedades Rotulado (LPG).* Nesta abordagem, as entidades Filme e Ator foram representadas como nós com seus respectivos *rótulos* (:Filme, :Ator), e seus atributos foram armazenados como propriedades em cada nó. A conexão entre um ator e um filme foi modelada como um relacionamento direcionado e *tipado*[8] - (Ator)-[:ACTED_IN]->(Filme) - que, por sua vez, continha propriedades próprias, como o nome do personagem interpretado. Este modelo é projetado para otimizar a *travessia de relacionamentos*, tornando consultas que exploram conexões entre dados, como buscar os filmes de um ator, extremamente eficientes (EIFREM; ROBINSON; WEBBER, 2015).

3.2.4. Redis

A modelagem de dados no Redis foi realizada através do uso estratégico de suas estruturas de dados nativas, indo além do simples paradigma chave-valor. As entidades Filme e Ator foram armazenadas como *hashes*, onde a chave do Redis identificava o objeto e os campos do *hash* continham seus respectivos atributos. Para representar os relacionamentos e otimizar as consultas, foram criados índices secundários manuais utilizando a estrutura de *set*. Por exemplo, para cada ator, um *set* foi criado contendo os Identificadores (Ids) [9] de todos os filmes em que atuou, permitindo uma recuperação de dados relacionais com altíssima performance. Essa abordagem de modelagem, que utiliza as estruturas de dados do Redis para simular modelos mais complexos, é um padrão documentado para se obter o máximo de desempenho em casos de uso específicos (CARLSON, 2013).

[8] Refere-se ao conceito de linguagens que utilizam variáveis com tipos específicos.
[9] São valores que identificam elementos específicos dentro de um sistema, como uma página web, um banco de dados ou uma aplicação.

3.3. Cenários de Teste

Para uma avaliação de desempenho multifacetada, foram projetados quatro cenários de teste distintos, cada um focado em uma carga de trabalho específica: escrita em lote, leitura complexa, agregação de dados e travessia de relacionamentos. Com o objetivo de obter medições precisas do desempenho dos bancos, os testes foram executados por scripts automatizados que interagiam diretamente com a API, isolando assim a performance do backend de qualquer latência introduzida pela interface do usuário.

3.3.1. Cenário 1: Inserção em lote

O primeiro cenário avaliou a performance de escrita em lote (*bulk write* [10]). Este teste consistiu na carga inicial de todo o conjunto de dados[11] em cada um dos quatro bancos, partindo de um estado vazio. A métrica aferida foi o tempo total de execução, em segundos, para que cada sistema completasse a inserção de todos os registros.

3.3.2. Cenário 2: Busca Avançada

O segundo cenário de teste foi desenhado para avaliar a performance de leitura complexa. Para isso, foi executada uma consulta que buscava filmes (tipo = 'filme') lançados a partir do ano de 2010, que continham simultaneamente os gêneros "Ação" e "Aventura" e que possuíam uma nota média igual ou superior a 7.0. Os critérios de filtro foram escolhidos para simular uma busca multifacetada realista, similar às encontradas em plataformas de catalogação de mídia como o IMDb. O objetivo deste teste foi medir a eficiência dos mecanismos de consulta e dos índices secundários de cada banco de dados para satisfazer uma requisição de alta seletividade. A métrica aferida foi o tempo médio de resposta para esta consulta.

3.3.3. Cenário 3: Agregação

[10] Refere-se à execução de várias operações de escrita (inserções, atualizações, exclusões) em uma única chamada ao banco de dados, ao invés de realizar cada operação individualmente.
[11] Os arquivos de dados originais estão disponíveis no repositório do projeto. Disponível em: https://github.com/Ozen-ok/tcc-nosql-comparativo/tree/b64bc39a01bd2e8bfbaef445f3ef07c35bb7d2f6/data. Acesso em: 11 jun. 2025.

O terceiro cenário de teste foi elaborado para mensurar a capacidade analítica de cada banco de dados através de uma consulta de agregação de alta complexidade: o cálculo da nota média de cada gênero de filme. Este teste exigia que o banco de dados tratasse a lista de gêneros de cada registro antes de realizar o agrupamento e o cálculo da média. O objetivo foi comparar a eficiência de cada sistema para executar tal operação, contrastando as soluções que possuem arcabouços de agregação robustos no servidor, como o *Aggregation Framework* do MongoDB (MONGODB, 2024) e as funções nativas do Cypher no Neo4j (NEO4J, 2024)**,** com aquelas que dependem primariamente de processamento no lado do cliente, como é o caso do Redis (REDIS, 2024) e, em menor grau, do Cassandra (APACHE CASSANDRA, 2024).

3.3.4. Cenário 4: Leitura de Relacionamento

O quarto e último cenário de teste foi projetado para avaliar a performance de cada banco na travessia de relacionamentos. O teste consistia em uma consulta para recuperar todos os filmes associados a um ator específico, selecionado a partir de uma lista pré-definida [12]. O objetivo foi medir a eficiência com que cada modelo de dados implementado permitia resolver uma consulta de relacionamento um-para-muitos, uma operação fundamental em diversas aplicações. Assim como nos cenários anteriores, a métrica aferida foi o tempo médio de resposta para a consulta.

3.4. Ambiente de Teste

Todos os testes de desempenho foram executados em um único e consistente ambiente de hardware e software. O hardware utilizado consistiu em um computador equipado com um processador AMD Ryzen 7 5700X3D, 16 GB de memória RAM DDR4 e um SSD SATA Samsung 870 EVO de 512 GB. O sistema operacional base foi o Windows 11 Pro (versão 22H2), com o ambiente de testes sendo executado sobre o *Windows Subsystem for Linux 2 (WSL2)* rodando a distribuição Ubuntu 24.04.2 LTS. A conteinerização foi gerenciada pelo Docker Desktop v4.41.2, que orquestrou os contêineres dos bancos de dados nas seguintes versões: MongoDB v8.0, Cassandra v5.0, Neo4j v5.18 e Redis v8.0. A *imagem Docker* da

[12] A lista de atores utilizada como entrada para os testes está disponível no repositório do projeto. Disponível em: https://github.com/Ozen-ok/tcc-nosql-comparativo/blob/b64bc39a01bd2e8bfbaef445f3ef07c35bb7d2f6/data/atores.tsv. Acesso em: 11 jun. 2025.

aplicação foi baseada em Python 3.10, enquanto os scripts de teste foram executados externamente aos contêineres, a partir do ambiente WSL2, utilizando Python 3.12.3.

3.5. Coleta de Métricas

A métrica primária para a avaliação de desempenho foi o tempo de execução de cada operação, aferido em segundos com o uso da biblioteca time do Python. Para garantir a consistência dos resultados, cada cenário de teste [13] foi executado dez vezes consecutivas. A primeira medição de cada série foi descartada para mitigar os efeitos de *warm-up* do sistema, como o aquecimento de caches. Portanto, o valor de performance final utilizado para a análise comparativa corresponde à média aritmética das nove execuções válidas subsequentes. Os tempos brutos de todas as dez execuções foram persistidos em arquivos no formato JSON para fins de registro e transparência.

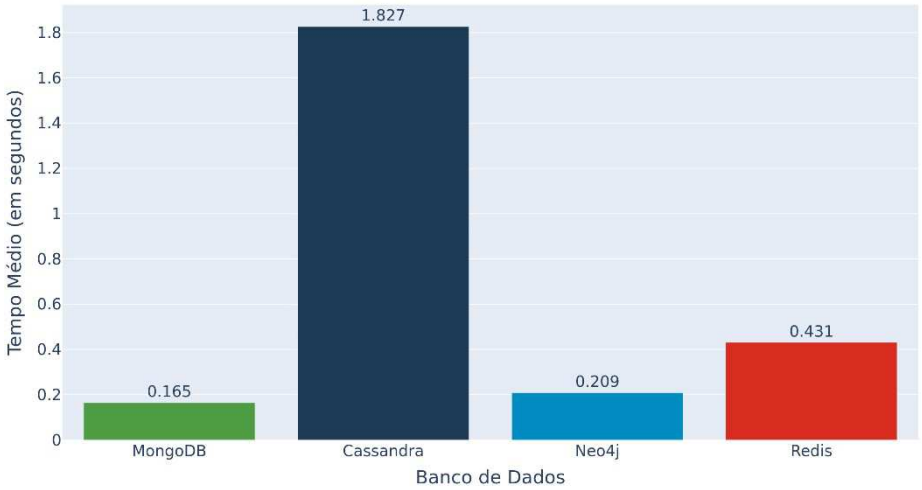
[13] Os scripts utilizados para a execução automatizada dos testes estão disponíveis em: https://github.com/Ozen-ok/tcc-nosql-comparativo/tree/b64bc39a01bd2e8bfbaef445f3ef07c35bb7d2f6/testes. Acesso em: 11 jun. 2025.

4. RESULTADOS

Este capítulo apresenta os resultados quantitativos obtidos a partir da execução dos cenários de teste de desempenho descritos na Metodologia. Os dados para cada um dos quatro cenários — inserção em lote, busca avançada, agregação e leitura de relacionamento — serão expostos em subseções individuais, utilizando gráficos para a visualização comparativa e tabelas para a apresentação dos valores precisos de tempo médio.

4.1. Resultados do teste de inserção em lote

Figura 1 - Gráfico de tempo médio para o teste de inserção em lote



Fonte: autoria própria (2025).

Tabela 1 - Valores de tempo médio para o teste de inserção em lote

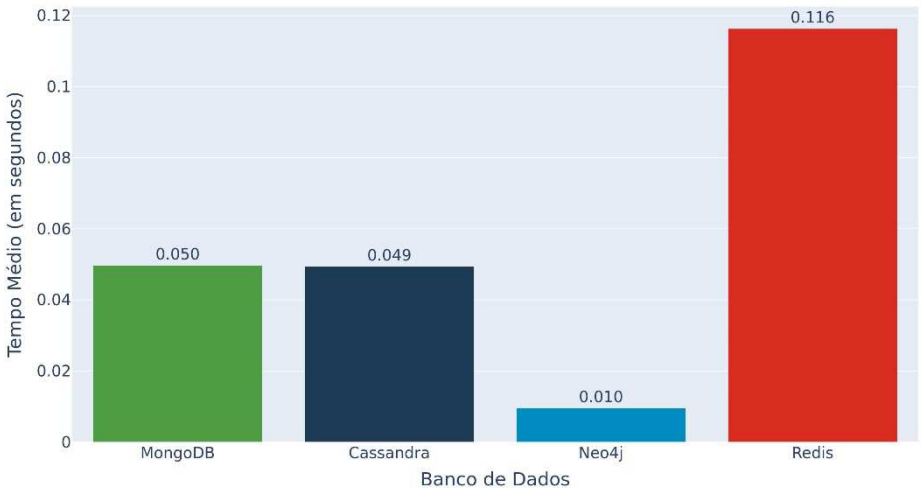
Banco de Dados	Tempo Médio (s)
MongoDB	0.1643
Cassandra	1.8722
Neo4j	0.2079
Redis	0.4302

Fonte: autoria própria (2025).

A Figura 1 e a Tabela 1 apresentam os resultados referentes ao cenário de inserção em lote. Observa-se que o MongoDB foi o banco de dados com o menor tempo médio de execução, completando a carga massiva de dados em aproximadamente 0.16 segundos. O Neo4j apresentou um desempenho similar, com um tempo de 0.21 segundos, seguido pelo Redis, com 0.43 segundos. Em contrapartida, o Cassandra demonstrou ser significativamente mais lento neste cenário, registrando o maior tempo médio, com 1.87 segundos.

4.2. Resultados do teste de busca avançada

Figura 2 - Gráfico de tempo médio para o teste de busca avançada



Fonte: autoria própria (2025).

Tabela 2 - Valores de tempo médio para o teste de busca avançada

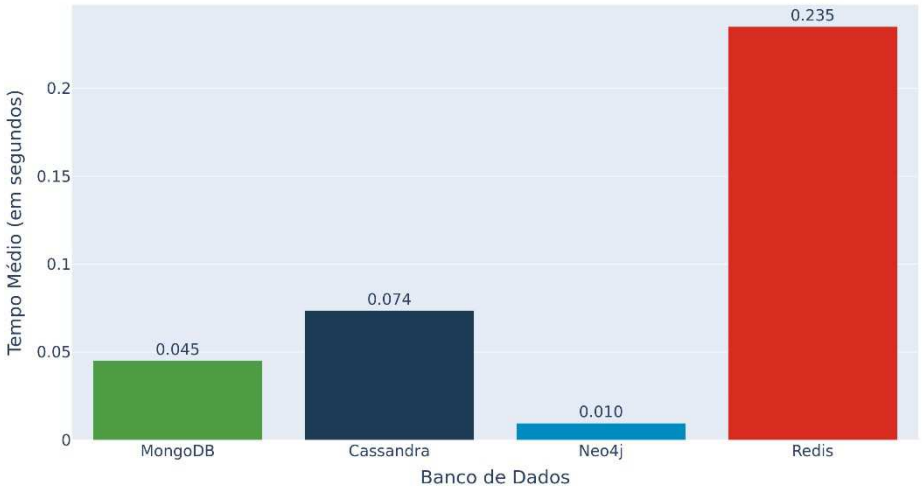
Banco de Dados	Tempo Médio (s)
MongoDB	0.050
Cassandra	0.049
Neo4j	0.010
Redis	0.116

Fonte: autoria própria (2025).

A Figura 2 e a Tabela 2 expõem os resultados do cenário de busca avançada. Neste teste, o Neo4j demonstrou a melhor performance, com um tempo médio de 0.046 segundos. O MongoDB e o Cassandra apresentaram resultados muito próximos e igualmente rápidos, com 0.049 e 0.053 segundos, respectivamente. O Redis, por sua vez, foi o menos eficiente para este tipo de consulta complexa, registrando um tempo médio de 0.112 segundos.

4.3. Resultados do teste de agregação

Figura 3 - Gráfico de tempo médio para o teste de agregação



Fonte: autoria própria (2025).

Tabela 3 - Valores de tempo médio para o teste de agregação

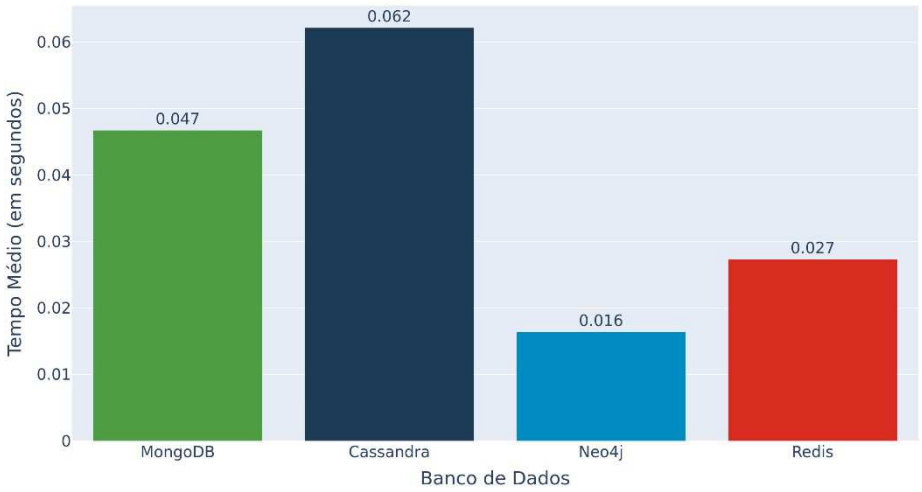
Banco de Dados	Tempo Médio (s)
MongoDB	0.045
Cassandra	0.074
Neo4j	0.010
Redis	0.235

Fonte: autoria própria (2025).

Os dados referentes ao teste de agregação, que calculou a nota média por gênero, estão apresentados na Figura 3 e na Tabela 3. O Neo4j novamente obteve o melhor desempenho, com um tempo de execução de apenas 0.013 segundos. O MongoDB também se mostrou muito eficiente para a tarefa, com uma média de 0.045 segundos. O Cassandra e o Redis registraram tempos consideravelmente maiores, com 0.073 e 0.231 segundos, respectivamente, indicando menor aptidão para este tipo de carga de trabalho analítica.

4.4. Resultados do teste de leitura de relacionamento

Figura 4 - Gráfico de tempo médio para o teste de leitura de relacionamento



Fonte: autoria própria (2025).

Tabela 4 - Valores de tempo médio para o teste de leitura de relacionamento

Banco de Dados	Tempo Médio (s)
MongoDB	0.047
Cassandra	0.062
Neo4j	0.016
Redis	0.027

Fonte: autoria própria (2025).

A Figura 4 e a Tabela 4 detalham a performance de cada banco no teste de travessia de relacionamentos. Como esperado, o Neo4j foi o mais rápido, executando a busca pelos filmes de um ator em 0.018 segundos. Notavelmente, o Redis obteve a segunda melhor performance, com 0.028 segundos, superando o MongoDB (0.047 segundos) e o Cassandra (0.064 segundos) neste cenário específico.

5. DISCUSSÃO

Esta seção se dedica à análise interpretativa dos resultados de desempenho apresentados no capítulo anterior. O objetivo é correlacionar os tempos de execução observados com as estratégias de modelagem de dados e as características arquitetônicas de cada banco de dados. Através de uma abordagem comparativa, serão discutidos os pontos fortes e fracos de cada tecnologia, explicando os motivos por trás da performance em cada cenário de teste.

Conforme os resultados, o Neo4j demonstrou uma clara superioridade nos cenários de leitura (busca avançada, agregação e leitura de relacionamento). Tal desempenho se deve ao fato de sua arquitetura nativa de grafos armazenar relacionamentos não como valores a serem indexados, mas como ponteiros físicos diretos entre os nós. Esse conceito, conhecido como adjacência livre de índice (*index-free adjacency*), torna a travessia de conexões uma operação extremamente rápida e de custo computacional constante, independentemente do volume total de dados no grafo (EIFREM; ROBINSON; WEBBER, 2015).

O MongoDB destacou-se com a melhor performance no cenário de inserção em lote e manteve um desempenho consistentemente alto nos testes de leitura, afirmando seu perfil como um banco de dados de propósito geral robusto. Sua eficiência na escrita massiva pode ser atribuída ao formato de armazenamento binário BSON e a um motor otimizado para a inserção rápida de documentos, que não possui a sobrecarga de gerenciar relacionamentos transacionais complexos. Notavelmente, mesmo com um modelo de dados normalizado que exigia a execução de *joins* em nível de aplicação, sua performance de leitura se manteve competitiva. Tal resultado é justificado pelo seu poderoso sistema de índices secundários, que otimiza a filtragem, e pela capacidade do seu ***Aggregation Framework*** de executar operações analíticas complexas diretamente no servidor (BRADSHAW; BRAZIL; CHODOROW, 2019).

O Redis, por sua vez, demonstrou um perfil de desempenho altamente especializado. Sua excelente performance no teste de leitura de relacionamento não derivou de uma capacidade relacional nativa, mas de uma modelagem de dados que utilizou a estrutura de Set como um índice secundário em memória, permitindo buscas de baixíssima latência (CARLSON, 2013). Em contrapartida, sua dificuldade no cenário de agregação evidenciou uma arquitetura que delega o processamento analítico para a aplicação cliente, reforçando seu papel como uma ferramenta especialista em velocidade para operações específicas.

De forma contrastante, a especialização do Cassandra se manifestou em seu desempenho de escrita. O banco registrou o maior tempo de execução no teste de inserção em lote, um comportamento esperado de sua arquitetura interna, otimizada para garantir alta taxa de

ingestão de dados e resiliência em ambientes distribuídos (CARPENTER; HEWITT, 2022). Portanto, este resultado não diminui o valor do Cassandra em seu caso de uso ideal, mas sim reforça que seu ponto forte não é a latência de escrita em um único nó, escopo deste estudo.

A análise comparativa de desempenho entre os quatro bancos de dados NoSQL valida um princípio fundamental da arquitetura de software. Conforme defendem Humble e Farley (2011), decisões arquiteturais eficazes não se baseiam na busca por uma tecnologia universalmente superior, mas sim na seleção de ferramentas e padrões que melhor se alinham aos requisitos e ao contexto específico de cada projeto. Nesse sentido, os resultados deste estudo não apontaram um vencedor absoluto, mas sim um espectro de especializações, revelando que a escolha da tecnologia mais adequada está intrinsecamente ligada à natureza da carga de trabalho e aos padrões de acesso a dados da aplicação.

6. CONCLUSÃO

Este trabalho se propôs a realizar uma análise de desempenho comparativa entre quatro bancos de dados NoSQL de paradigmas distintos: MongoDB, Cassandra, Neo4j e Redis. Por meio de uma aplicação de testes em um ambiente conteinerizado, foram executados quatro cenários de carga de trabalho para simular operações comuns em aplicações modernas. Os resultados obtidos validam um princípio fundamental da arquitetura de software: não existem tecnologias ou padrões universalmente superiores, mas sim um conjunto de ***trade-offs*** [14] que devem ser analisados à luz dos requisitos específicos de cada problema (RICHARDS; FORD, 2020).

Dentre os principais resultados, destaca-se a superioridade do Neo4j em cenários que envolvem a travessia de relacionamentos, graças ao seu motor de grafos nativo. O MongoDB provou ser um sistema de propósito geral robusto e flexível, enquanto o Redis demonstrou alta performance para leituras específicas através de uma modelagem de dados inteligente. Por fim, os resultados do Cassandra, embora inferiores em um ambiente de nó único, foram consistentes com sua arquitetura projetada para alta escalabilidade de escrita em ambientes distribuídos.

É importante, contudo, reconhecer as limitações deste estudo para a correta contextualização dos resultados. A principal limitação foi a execução dos testes em um ambiente de nó único, que não permite avaliar o comportamento de bancos como o Cassandra em um *cluster* distribuído, seu principal caso de uso. Adicionalmente, o volume de dados utilizado foi contido e a análise se restringiu à métrica de tempo, não abrangendo o consumo de recursos como CPU e memória.

Com base nas limitações constatadas, percebem-se diversas oportunidades para a continuação e aprofundamento desta pesquisa. Sugere-se a replicação dos testes em um ambiente de *cluster* com múltiplos nós, a utilização de um conjunto de dados em maior escala (na ordem de milhões ou bilhões de registros) e a inclusão de outras categorias de bancos NoSQL. Tais estudos futuros permitiriam uma compreensão ainda mais abrangente do ecossistema de bancos de dados não relacionais.

[14] Refere-se a uma situação onde é necessário escolher entre duas ou mais opções, e ao escolher uma, inevitavelmente abre-se mão de algo em outra área.

REFERÊNCIAS

AMAZON WEB SERVICES. **O que é um banco de dados de chave-valor?** 2024. Disponível em: https://aws.amazon.com/pt/nosql/key-value/. Acesso em: 17 jun. 2025.

AMAZON WEB SERVICES. **O que é um banco de dados de documentos?** 2024. Disponível em: https://aws.amazon.com/pt/nosql/document/. Acesso em: 17 jun. 2025.

AMAZON WEB SERVICES. **O que é um banco de dados de grafos?** 2024. Disponível em: https://aws.amazon.com/pt/nosql/graph/. Acesso em: 17 jun. 2025.

APACHE CASSANDRA. ***Aggregate functions (cqlsh)***. 2024. Disponível em: https://cassandra.apache.org/doc/4.0/cassandra/cql/functions.html#aggregate-functions. Acesso em: 11 jun. 2025.

APACHE CASSANDRA. ***Cassandra Use Cases***. 2024. Disponível em: https://cassandra.apache.org/_/case-studies.html. Acesso em: 17 jun. 2025.

BRADSHAW, Shannon; BRAZIL, Eoin; CHODOROW, Kristina. ***MongoDB: The Definitive Guide***. 3. ed. Sebastopol, CA: O'Reilly Media, 2019.

CARLSON, Josiah L. ***Redis in Action***. Shelter Island, NY: Manning, 2013.

CARPENTER, Jeff; HEWITT, Eben. ***Cassandra: The Definitive Guide***. 3. ed. rev. Sebastopol, CA: O'Reilly Media, 2022.

CHAUDHRY, Shoaib; YOUSAF, Muhammad. ***Graph-Powered Analytics and Machine Learning with Neo4j***. Birmingham: Packt Publishing, 2020.

EIFREM, Emil; ROBINSON, Ian; WEBBER, Jim. ***Graph Databases***. 2. ed. Sebastopol, CA: O'Reilly Media, 2015.

GOOGLE CLOUD. ***Bancos de dados de colunas largas***. 2024. Disponível em: https://cloud.google.com/discover/what-is-nosql?hl=pt-br. Acesso em: 17 jun. 2025.

HUMBLE, Jez; FARLEY, David. ***Continuous Delivery:*** *Reliable Software Releases through Build, Test, and Deployment Automation.* Boston: Addison-Wesley, 2011.

KANE, Sean P.; MATTHIAS, Karl. ***Docker: Up & Running:*** *Shipping Reliable Containers in Production*. 3. ed. Sebastopol, CA: O'Reilly Media, 2023.

KHAN, Tasleem; et al. ***Redis Certified Developer:*** *A practical guide to building high-performance applications with Redis.* Birmingham: Packt Publishing, 2023.

McCREARY, Dan; KELLY, Ann. ***Making Sense of NoSQL:*** *A guide for managers and the rest of us.* Shelter Island, NY: Manning, 2013.

MONGODB. ***Aggregation Pipeline***. 2024. Disponível em: https://www.mongodb.com/docs/manual/core/aggregation-pipeline/. Acesso em: 11 jun. 2025.

MONGODB. ***MongoDB at Scale***. 2024. Disponível em: https://www.mongodb.com/products/capabilities/mongodb-scale. Acesso em: 17 jun. 2025.
NEO4J. ***Cypher Manual:*** *Aggregating Functions.* 2024. Disponível em: https://neo4j.com/docs/cypher-manual/current/functions/aggregating/. Acesso em: 11 jun. 2025.

NEO4J. ***Neo4j Graph Database & Analytics***. 2024. Disponível em: https://neo4j.com/. Acesso em: 17 jun. 2025.

REDIS. ***Data types***. 2024. Disponível em: https://redis.io/docs/latest/develop/data-types/. Acesso em: 11 jun. 2025.

REDIS. **Redis no mundo todo**. 2024. Disponível em: https://redis.io/pt/. Acesso em: 17 jun. 2025.

RICHARDS, Mark; FORD, Neal. ***Fundamentals of Software Architecture:*** *An Engineering Approach.* Sebastopol, CA: O'Reilly Media, 2020.

STREAMLIT. ***Streamlit Documentation***. 2025. Disponível em: https://docs.streamlit.io. Acesso em: 17 jun. 2025.

SWEIGART, Al. ***Automate the Boring Stuff with Python:*** *Practical Programming for Total Beginners.* 2. ed. San Francisco, CA: No Starch Press, 2020.

GLOSSÁRIO

Adjacência livre de índice (*index-free adjancency*): É um conceito fundamental em bancos de dados de grafos nativos, que se refere à forma como os nós são conectados e acessados diretamente, sem a necessidade de consultas a índices.

***Aggregation Framework* (MongoDB):** Um conjunto de ferramentas nativas do MongoDB para o processamento de múltiplos documentos e o retorno de resultados computados, operando através de um *pipeline* de estágios (como $*match*, $*group*, etc.) que permite análises complexas no lado do servidor.

API (*Application Programming Interface*): Conjunto de regras e protocolos que permite a comunicação e a troca de dados entre diferentes sistemas de software.

App_network: A rede virtual privada e isolada, criada pelo Docker Compose, à qual todos os contêineres do projeto foram conectados, permitindo a comunicação entre eles por meio de seus nomes de serviço.

Backend: A parte de uma aplicação que roda no servidor e é responsável pela lógica de negócio, processamento de dados e comunicação com o banco de dados.

Biblioteca: Uma coleção de código pré-escrito (funções, classes) que pode ser reutilizada por desenvolvedores para realizar tarefas comuns sem ter que reescrever o código do zero. Exemplo: Pydantic para validação de dados.

Bigtable: O Bigtable é um armazenamento de chave-valor e famílias de colunas, ideal para o acesso rápido a dados estruturados, semiestruturados ou não estruturados.

BSON (*Binary JSON*): Um formato de serialização binária utilizado para armazenar e transmitir documentos do tipo JSON. É o formato de armazenamento de dados interno utilizado pelo MongoDB, que o estende para incluir tipos de dados adicionais, como datas e dados binários, de forma mais eficiente que o JSON textual.

***Cluster*:** é uma técnica de computação que agrupa diversos computadores ou servidores em um mesmo sistema para trabalharem em conjunto.

Código aberto: Desenvolvido de forma descentralizada e colaborativa e conta com a revisão e a produção pela comunidade. Ele costuma ser mais barato, mais flexível e mais duradouro do que as opções proprietárias, já que é desenvolvido por comunidades independentes, e não por um único autor ou empresa.

Contêiner: Um ambiente de execução leve, portátil e isolado que empacota uma aplicação e todas as suas dependências (código, bibliotecas, etc.). Garante que a aplicação rode da mesma forma independentemente da máquina.

Conteinerização: A prática de empacotar o código de uma aplicação e todas as suas dependências (bibliotecas, configurações, etc.) em uma unidade padronizada e isolada chamada *contêiner*. Este processo garante que a aplicação seja executada de forma consistente e confiável em qualquer ambiente computacional.

CRUD: Acrônimo para as quatro operações básicas de armazenamento persistente: ***C****reate* (Criar), *Read* (Ler), *Update* (Atualizar) e *Delete* (Deletar).

Desnormalização: Processo de otimização de banco de dados que consiste em introduzir redundância e duplicar dados de forma controlada. O objetivo é reduzir a necessidade de operações de junção (*joins*), melhorando significativamente a performance de leitura em troca de um maior uso de espaço de armazenamento.

Docker: A plataforma de software utilizada para criar, gerenciar e executar contêineres. Foi a ferramenta central para garantir a reprodutibilidade e o isolamento do ambiente de testes deste projeto.

Docker Compose: Uma ferramenta para definir e gerenciar aplicações Docker multi-contêiner. Foi utilizada através do arquivo docker-compose.yml para orquestrar o levantamento da aplicação de testes e das quatro instâncias de banco de dados simultaneamente.

Dockerfile: Um arquivo de texto que contém um conjunto de instruções sequenciais para a criação de uma *imagem Docker*. Funciona como a "planta baixa" ou a receita do ambiente da aplicação, especificando a imagem base (ex: Python 3.10), as dependências a serem instaladas (via requirements.txt), os arquivos a serem copiados (como o código-fonte da pasta src e os assets) e o comando padrão para iniciar a aplicação.

Dynamo (modelo de distribuição): Refere-se a um conjunto de princípios de arquitetura para sistemas de banco de dados distribuídos, introduzidos em um artigo de 2007 pela Amazon. Suas principais características incluem descentralização, alta disponibilidade através de replicação de dados e uma política de "consistência eventual". Este modelo influenciou o design de vários bancos de dados NoSQL, incluindo o Apache Cassandra.

***Endpoint*:** Um ponto final de comunicação em uma API. É uma URL específica onde uma requisição pode ser enviada para executar uma operação particular. Por exemplo, /api/v1/mongo/filmes é um *endpoint* para buscar filmes no MongoDB.

FastAPI: Um *framework* web Python moderno e de alta performance para a construção de *APIs*. É baseado em tipos de dados Python padrão e se destaca pela sua velocidade e pela geração automática de documentação interativa. Foi o *framework* utilizado para desenvolver o *backend* da aplicação.

***Framework*:** Uma estrutura de software que oferece uma base para o desenvolvimento de aplicações, impondo uma arquitetura e fornecendo funcionalidades prontas para uso. Diferente de uma biblioteca, o *framework* dita o fluxo de controle da aplicação. FastAPI e Streamlit são os *frameworks* utilizados neste projeto.

Frontend: A parte de uma aplicação com a qual o usuário interage diretamente; a interface do usuário (UI). Neste trabalho, o frontend é um painel de controle desenvolvido com o *framework* Streamlit para executar e visualizar os testes.

Grafo de Propriedades Rotulado (LPG): O modelo de dados utilizado pelo Neo4j. É composto por *nós* (entidades), *relacionamentos* (conexões entre nós), *propriedades* (atributos em nós e relacionamentos) e *rótulos* (que agrupam os nós em categorias).

***Hash* (Estrutura de Dados Redis):** Um tipo de dado no Redis que é ideal para representar objetos. Funciona como um mapa de campos e valores, onde a chave principal do Redis aponta para este mapa. Neste projeto, foi utilizado para armazenar os atributos de cada Filme e Ator.

ID (Identificador): é um atributo que atribui um nome único a um elemento HTML ou a um objeto em um código.

Imagem Docker: Um modelo usado apenas para leitura, contendo um conjunto de instruções para a criação de um contêiner. Funciona como um conjunto de instruções para o ambiente da aplicação.

Índice Secundário Manual: Uma estrutura de dados, criada e mantida pela própria aplicação, que permite a busca eficiente de dados por um atributo que não é a chave primária. No contexto da implementação com Redis, foram utilizados *Sets* para criar índices que mapeiam, por exemplo, um gênero a um conjunto de IDs de filmes, ou um ator a um conjunto de IDs de filmes em que atuou.

***Join* em Nível de Aplicação (*Application-Level Join*):** Padrão de consulta em que a aplicação simula uma operação de JOIN relacional ao executar múltiplas buscas sequenciais em um banco de dados NoSQL. A "junção" dos dados de diferentes coleções ou tabelas é realizada no código da própria aplicação, e não pelo servidor do banco de dados.

JSON (*JavaScript Object Notation*): Um formato de arquivo e de troca de dados leve, de fácil leitura para humanos e de fácil interpretação para máquinas. É o padrão de comunicação para a maioria das APIs web modernas, incluindo a desenvolvida neste trabalho.

Modelagem de Dados: O processo de definir como os dados serão armazenados, organizados e relacionados em um banco de dados. A modelagem varia drasticamente entre os diferentes paradigmas de bancos NoSQL (documento, coluna-larga, grafo, chave-valor).

Nó: A unidade fundamental de um banco de dados de grafos, usada para representar uma entidade. Exemplos: Filme, Ator.

NoSQL: Uma classe de sistemas de gerenciamento de banco de dados que não segue o modelo relacional tradicional. São otimizados para escalabilidade, flexibilidade e desempenho em casos de uso específicos. O TCC compara quatro tipos distintos de bancos NoSQL.

Pandas: Biblioteca de software escrita para a linguagem Python, utilizada para manipulação e análise de dados. Neste trabalho, foi fundamental na etapa de pré-processamento dos dados.

Propriedade: Um par de chave-valor usado para armazenar dados dentro de *nós* e *relacionamentos*. Por exemplo, titulo: 'Inception' em um nó :Filme, ou nome_personagem: 'Cobb' em um relacionamento :ACTED_IN.

Pydantic: Uma biblioteca Python utilizada para validação e parse de dados. Neste projeto, foi usada para definir os *schemas* dos dados (Filme, Ator), garantindo a integridade e a tipagem correta das informações que transitam pela API.

Relacionamento: A conexão que define como dois *nós* estão relacionados. No Neo4j, relacionamentos são direcionados, tipados e podem ter propriedades. Exemplo: (Ator)-[:ACTED_IN]->(Filme).

Relacionamento Tipado (Grafo): No contexto de bancos de dados de grafos, refere-se a um relacionamento que possui um "tipo" ou "nome" que descreve sua natureza semântica. Exemplo: :ACTED_IN, :FRIENDS_WITH, :OWNS.

Rótulo: Usado para agrupar *nós* em categorias ou classificações. Um nó pode ter múltiplos rótulos. Neste projeto, os rótulos utilizados foram :Filme e :Ator.

***Set* (Estrutura de Dados Redis):** Um tipo de dado no Redis que armazena uma coleção de elementos únicos e não ordenados. Sua principal vantagem é a altíssima velocidade para adicionar, remover e verificar a existência de membros. Neste trabalho, foram utilizados para criar *índices secundários manuais*.

***Src-layout*:** Padrão de organização de projetos em Python onde todo o código-fonte da aplicação reside dentro de um diretório nomeado src.

Streamlit: Um *framework* Python de código aberto utilizado para criar e compartilhar aplicações web interativas e focadas em ciência de dados com poucas linhas de código. Neste trabalho, foi a ferramenta escolhida para o desenvolvimento do *frontend* da aplicação de testes.

Travessia de Relacionamentos (*Graph Traversa*l): O processo de navegar através de um banco de dados de grafos, partindo de um ou mais *nós* de início e "viajando" para outros nós ao seguir os *relacionamentos* que os conectam. É a operação fundamental para explorar conexões e descobrir padrões em dados conectados, sendo uma funcionalidade nativa e altamente otimizada em bancos de dados de grafos como o Neo4j.

***Warm-up* (em Testes de Performance):** Refere-se ao processo de realizar uma ou mais execuções iniciais de um teste, cujos resultados são descartados. O objetivo é "aquecer" o sistema, permitindo que caches sejam populados, conexões sejam estabelecidas e o compilador Just-In-Time (JIT) otimize o código, garantindo que as medições subsequentes reflitam o desempenho do sistema em um estado estável.

WSL2 (*Windows Subsystem for Linux 2*): Uma camada de compatibilidade desenvolvida pela Microsoft que permite a execução de um ambiente Linux completo, incluindo a maioria das ferramentas de linha de comando e aplicações, diretamente no Windows, sem a necessidade de uma máquina virtual tradicional. Neste projeto, foi utilizado para prover o ambiente de execução dos contêineres Docker.