UNIVERSIDADE FEDERAL DO CEARÁ

INSTITUTO UNIVERSIDADE VIRTUAL

CURSO DE GRADUAÇÃO EM SISTEMAS E MÍDIAS DIGITAIS

AGATHA LINDEMBERG ARAÚJO DOS SANTOS

ANÁLISE DE DESEMPENHO DAS TECNOLOGIAS DE CARREGAMENTO DE DADOS EM BANCOS DE DADOS PELA LINGUAGEM JAVA

FORTALEZA

2026

AGATHA LINDEMBERG ARAÚJO DOS SANTOS

ANÁLISE DE DESEMPENHO DAS TECNOLOGIAS DE CARREGAMENTO DE DADOS EM BANCOS DE DADOS PELA LINGUAGEM JAVA

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Sistemas e Mídias Digitais do Instituto Universidade Virtual da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Sistemas e Mídias Digitais.

Orientador: Prof. Dr. Leonardo Oliveira Moreira

FORTALEZA

2026

AGATHA LINDEMBERG ARAÚJO DOS SANTOS

ANÁLISE DE DESEMPENHO DAS TECNOLOGIAS DE CARREGAMENTO DE DADOS EM BANCOS DE DADOS PELA LINGUAGEM JAVA

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Sistemas e Mídias Digitais do Instituto Universidade Virtual da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Sistemas e Mídias Digitais.

Aprovada em:

BANCA EXAMINADORA

Prof. Dr. Leonardo Oliveira Moreira (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. Windson Viana de Carvalho
Universidade Federal do Ceará (UFC)

Prof. Dr. José Gilvan Rodrigues Maia
Universidade Federal do Ceará (UFC)

À minha família, que sempre caminhou ao meu lado e fez da minha educação uma prioridade. À minha mãe, cujo cuidado foi o abraço que me sustentou quando pensei em desistir. Ao meu pai, por me oferecer a certeza de que eu nunca enfrentaria nada sozinha. Ao meu irmão, que enche meus dias de leveza, alegria e risadas.

AGRADECIMENTOS

À minha mãe e ao meu pai, por estarem ao meu lado em cada etapa — nas leves e, sobretudo, nas mais desafiadoras. Por serem a base que sustentou meus passos ao longo dos anos e por me darem o impulso necessário sempre que o desânimo aparecia. Obrigada por sempre colocarem minha educação em primeiro lugar e por serem, juntos, o maior exemplo de dedicação aos estudos que levo comigo.

Ao meu irmão, Wagner Júnior (Wagberg), obrigada por apoiar cada escolha que fiz, por ser meu consolo nos momentos turbulentos e por estar presente mesmo quando eu não sabia pedir ajuda. Sua presença trouxe conforto e força quando mais precisei.

Ao meu professor e orientador, Leo, por toda a atenção, paciência e cuidado ao me conduzir nesta reta final da graduação. E por cada disciplina que ministrou, nas quais despertou ainda mais meu amor por programação. Aos professores que compõem esta banca, Gilvan e Windson (Windows), agradeço por fazerem parte da minha formação e aceitarem estar comigo neste momento tão importante.

Aos meus amigos — Arthur Branco, Isadora Bruno Leite, Italo Freire, Jasmine Mendes, Luana de Sousa, Marcos Kaio, Mattheus Del, Maxwell Martins, Oliver Perina e Paulie Medeiros — meu carinho e gratidão profundos. Por cada trabalho em grupo, pelas conversas e risadas na cantina da Química, pelas caronas, pelos RPGs, pelos jogos de tabuleiro, e por sempre me tratarem com tanto carinho. Obrigada por não terem desistido de mim.

“Então, não se preocupe. O futuro não dá medo. Você vai conhecer muitas pessoas que vai amar e vai conhecer muitas pessoas que também vão te amar. A noite pode parecer sem fim agora, mas, um dia, a manhã vai chegar. Você vai crescer sendo banhada por essa luz.”

(Suzume)

RESUMO

Este trabalho apresenta uma análise comparativa das principais tecnologias de persistência de dados utilizadas em aplicações Java: JDBC, Hibernate, JPA e Spring Data JPA. Considerando o crescimento do volume de dados e a necessidade de sistemas cada vez mais eficientes, investigou-se como cada tecnologia influencia o desempenho, o consumo de recursos e a complexidade de desenvolvimento. A pesquisa foi guiada pela seguinte questão: quais são as vantagens e desvantagens de cada abordagem de manipulação de dados no contexto de aplicações Java modernas? Para responder a essa questão, o estudo realizou uma revisão teórica sobre as diferentes estratégias de acesso e persistência de dados, destacando suas características técnicas, modelos de abstração, produtividade e impacto no desempenho. Foram analisadas tanto abordagens de baixo nível, como o JDBC, quanto soluções mais abstratas, como Hibernate, JPA e Spring Data JPA, baseadas no paradigma de ORM, que realiza o mapeamento entre objetos da linguagem Java e estruturas relacionais do banco de dados. Os resultados apontam que o JDBC oferece maior controle e desempenho bruto, porém exige maior esforço de implementação e manutenção. Tecnologias ORM, como Hibernate e JPA, proporcionam maior produtividade e facilitam o desenvolvimento, embora possam introduzir estouro de memória e problemas de escalabilidade em cenários de alta carga. Já o Spring Data JPA se destaca pela simplicidade e velocidade de desenvolvimento, mas pode sofrer limitações quando são necessárias configurações altamente personalizadas ou operações intensivas. A análise evidencia que a escolha da tecnologia mais adequada depende diretamente do contexto da aplicação, envolvendo fatores como complexidade da lógica de negócio, requisitos de desempenho, volume de dados e perfil da equipe de desenvolvimento. Conclui-se que nenhuma tecnologia é universalmente superior, reforçando a importância de decisões arquiteturais bem fundamentadas. O estudo contribui para orientar profissionais na seleção da abordagem de persistência mais apropriada e para fomentar discussões sobre desempenho, escalabilidade e produtividade no ecossistema Java.

Palavras-chave: Java; Persistência de Dados; Análise de Desempenho; Benchmarking.

ABSTRACT

This work presents a comparative analysis of the main data persistence technologies used in Java applications: JDBC, Hibernate, JPA, and Spring Data JPA. Considering the growth in data volume and the increasing demand for highly efficient systems, this study investigates how each technology influences performance, resource consumption, and development complexity. The research was guided by the following question: what are the advantages and disadvantages of each data manipulation approach in the context of modern Java applications? To address this question, the study conducted a theoretical review of different data access and persistence strategies, highlighting their technical characteristics, abstraction models, productivity, and performance impact. Both low-level approaches, such as JDBC, and more abstract solutions, such as Hibernate, JPA, and Spring Data JPA, based on the ORM paradigm, which maps Java language objects to relational database structures, were analyzed. The results indicate that JDBC provides greater control and raw performance but requires higher implementation and maintenance effort. ORM technologies, such as Hibernate and JPA, offer increased productivity and simplify development, although they may introduce memory overhead and scalability issues under high-load scenarios. Spring Data JPA stands out for its simplicity and rapid development capabilities but may face limitations when highly customized configurations or intensive operations are required. The analysis demonstrates that the choice of the most appropriate technology depends directly on the application context, involving factors such as business logic complexity, performance requirements, data volume, and the development team profile. It is concluded that no single technology is universally superior, reinforcing the importance of well-founded architectural decisions. This study contributes to guiding professionals in selecting the most suitable persistence approach and fostering discussions on performance, scalability, and productivity within the Java ecosystem.

Keywords: Java; Data Persistence; Performance Analysis; Benchmarking.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE CÓDIGOS-FONTE

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
CPU	*Central Processing Unit*
CRUD	*Create, Read, Update, Delete*
DAO	*Data Access Object*
DBMS	Database Management System
EE	*Enterprise Edition*
EJB	*Enterprise JavaBeans*
G1GC	*Garbage First Garbage Collector*
GC	*Garbage Collector*
IDE	*Integrated Development Environment*
JDBC	*Java DataBase Connectivity*
JDK	*Java Development Kit*
JIT	*Just-In-Time*
jOOQ	*Java Object Oriented Querying*
JPA	*Java Persistence API*
JPQL	*Java Persistence Query Language*
JVM	*Java Virtual Machine*
NoSQL	*Not Only SQL*
ORM	Object-Relational Mapping
R2DBC	*Reactive Relational Database Connectivity*
RAM	*Random Access Memory*
SQL	*Structured Query Language*
ZGC	*Z Garbage Collector*

SUMÁRIO

1 INTRODUÇÃO

No contexto do desenvolvimento de sistemas modernos, a construção de aplicações *back-end* para a web tem sido amplamente realizada com a linguagem Java, que ocupa uma posição de destaque entre as tecnologias de servidor. Além da adoção em novos projetos, a relevância do Java também se explica pela expressiva quantidade de sistemas legados corporativos que ainda operam em muitas organizações, os quais demandam manutenção contínua e evolução tecnológica para atender às necessidades de desempenho e escalabilidade modernas. Segundo a pesquisa "The State of Developer Ecosystem 2024" da JetBrains (2024), Java é a terceira linguagem de programação mais popular, logo após JavaScript e Python. Essa predominância é reforçada por levantamentos posteriores, que indicam o Java como a quarta linguagem de programação mais pesquisada no ranking de maio de 2025 de acordo com Tiobe (2025), considerando a frequência de buscas em plataformas como Google, Yahoo!, MSN e Wikipedia.

A ampla adoção da linguagem também pode ser observada em grandes empresas de tecnologia que utilizam a linguagem Java em arquiteturas críticas de *back-end* e serviços escaláveis. Estudos, como de Halili *et al.* (2025), indicam que organizações como Netflix, Uber e Shopify adotam abordagens baseadas em múltiplas tecnologias, incluindo Java, em suas arquiteturas de persistência e distribuição de dados em microsserviços, ilustrando tendências reais de adoção em cenários industriais de alta demanda e complexidade . Essas evidências sugerem que, tanto para manter sistemas de larga escala quanto para implementar novas soluções distribuídas confiáveis, a plataforma Java continua sendo uma escolha estratégica para empresas que enfrentam requisitos rigorosos de disponibilidade e desempenho.

Conforme Qu (2021), com o crescimento exponencial do volume de dados e a crescente interação entre páginas web dinâmicas e bancos de dados, torna-se fundamental garantir um desempenho eficiente no acesso a dados, com ênfase na redução da latência e no aumento das taxas de transferências. De acordo com a visão de Garcia (2023), esses sistemas enfrentam restrições rigorosas de tempo e precisam lidar com cargas de trabalho dinâmicas, tornando a eficiência do Database Management System (DBMS) essencial para atender às expectativas de desempenho.

Diante desse cenário, surgem tecnologias específicas no ecossistema Java que visam otimizar a gestão e a consulta de dados. A análise das diferentes abordagens de manipulação e persistência de dados é fundamental para compreender como cada tecnologia pode impactar diretamente o desempenho, a escalabilidade e a eficiência dos sistemas. A linguagem Java

oferece múltiplas alternativas para integração com bancos de dados, cada uma com características próprias em termos de abstração, flexibilidade e desempenho. Entre as principais tecnologias estão o *Java DataBase Connectivity* (JDBC), que oferece uma interface de baixo nível e controle total sobre as operações *Structured Query Language* (SQL); o Hibernate e o *Java Persistence API* (JPA), que proporcionam Object-Relational Mapping (ORM) e abstração dos comandos SQL; além do Spring Data, que visa maximizar a produtividade, reduzindo a quantidade de código necessário para operações comuns.

Em vista disso, a pergunta de pesquisa que orienta este estudo é: Quais são as vantagens e desvantagens de cada estratégia de manipulação de dados (JDBC, Hibernate, JPA e Spring Data) no desenvolvimento de aplicações Java, considerando aspectos de desempenho, consumo de recursos e complexidade de implementação?

1.1 Objetivos

Este estudo tem como objetivo geral analisar o desempenho das tecnologias JDBC, Hibernate, JPA e Spring Data na manipulação de dados em aplicações Java, nas vantagens, limitações e nos cenários mais apropriados para cada abordagem. Para alcançar esse objetivo geral, o trabalho foi guiado por três objetivos específicos:

a) investigar as principais tecnologias de carregamento e de manipulação de dados em bancos de dados disponíveis para a linguagem Java;
b) realizar experimentos práticos de desempenho dessas tecnologias com base em métricas como tempo de execução, uso de memória e eficiência transacional; e
c) comparar os resultados obtidos de tipo de aplicação, de volume de dados e dos requisitos de desempenho nos experimentos.

1.2 Justificativa

A escolha por analisar essas tecnologias justifica-se pela ampla adoção no ecossistema Java, bem como pelas distintas abordagens que cada uma oferece no acesso e na persistência de dados. Enquanto algumas priorizam o controle fino sobre as operações e o desempenho bruto, outras enfatizam a produtividade e a redução de códigos repetitivos que, embora frequentemente necessários, não são essenciais para a lógica principal do programa. Essa diversidade torna cada tecnologia mais ou menos adequada a depender do contexto da aplicação. Conforme destaca

George (2021), a escolha da tecnologia de acesso a dados em Java deve considerar um equilíbrio entre controle de baixo nível, desempenho e produtividade no desenvolvimento.

Nesse sentido, a inclusão do JDBC neste estudo não tem como objetivo apenas compará-lo diretamente com soluções baseadas em ORM, mas também utilizá-lo como referência de desempenho e consumo de recursos, uma vez que representa a abordagem de menor nível de abstração no ecossistema Java. Essa comparação permite evidenciar o impacto introduzido pelas camadas adicionais de abstração presentes em tecnologias como Hibernate, JPA e Spring Data, especialmente no que se refere ao custo computacional e ao uso de memória. Reconhece-se, contudo, que o JDBC possui características e objetivos distintos, o que limita uma comparação direta em termos de produtividade e complexidade de desenvolvimento.

Ademais, este estudo propõe a realização de experimentos práticos para avaliar o impacto das tecnologias JDBC, Hibernate, JPA e Spring Data em termos de desempenho, considerando a complexidade das operações e o volume de dados. A comparação entre essas soluções é fundamental para fornecer recomendações sobre a tecnologia mais apropriada para diferentes cenários, levando em conta tanto os requisitos de desempenho quanto a necessidade de escalabilidade do sistema.

1.3 Contribuições Esperadas

Ao final, espera-se que este trabalho ofereça uma visão clara e fundamentada sobre as vantagens e desvantagens de cada tecnologia, auxiliando os desenvolvedores na tomada de decisão ao escolher a melhor estratégia para seus projetos. Além disso, os resultados poderão servir como base para futuras pesquisas sobre otimização de acesso a dados em Java, contribuindo para a discussão sobre desempenho e produtividade no desenvolvimento de software.

1.4 Estrutura do Documento

- O Capítulo 2 apresenta o arcabouço teórico necessário para a compreensão aprofundada do trabalho.
- O Capítulo 3 descreve e explica a estrutura metodológica adotada para a condução da pesquisa.
- O Capítulo 4 expõe e discute os resultados obtidos, enfatizando os principais aspectos de avaliação.

- Por fim, o Capítulo 5 conclui o estudo e apresenta sugestões de trabalhos futuros que podem dar continuidade à pesquisa desenvolvida.

2 REFERENCIAL TEÓRICO

Este capítulo tem como objetivo apresentar as bases conceituais e técnicas que sustentam a análise comparativa entre diferentes abordagens de persistência de dados em aplicações Java. Para compreender as vantagens e limitações de cada tecnologia, é necessário contextualizar o papel dos sistemas de gerenciamento de banco de dados na arquitetura de sistemas modernos, assim como os mecanismos utilizados para realizar a integração entre aplicações Java e bancos relacionais.

Neste contexto, são exploradas as principais alternativas adotadas para persistência de dados: JDBC, Hibernate, JPA e Spring Data JPA, com destaque para seus fundamentos, modos de operação e implicações no desempenho e na produtividade do desenvolvimento. Também são discutidos aspectos relacionados à eficiência, como consumo de *Central Processing Unit* (CPU) e memória, tempo de execução e escalabilidade, com base em estudos empíricos e conceituais recentes. Adicionalmente, abordam-se divergências na literatura técnica quanto à adoção dessas tecnologias em diferentes contextos de aplicação, incluindo fatores como a complexidade da lógica de negócios, a necessidade de controle sobre as transações e o perfil da equipe de desenvolvimento.

Por fim, o capítulo é guiado por uma perspectiva crítica, considerando tanto os benefícios quanto os desafios associados a cada abordagem de persistência. Essa análise busca oferecer subsídios teóricos para a avaliação criteriosa das ferramentas disponíveis, contribuindo para decisões mais informadas no projeto e na manutenção de sistemas orientados a dados na linguagem Java.

2.1 Abordagens de Persistência em Java

A persistência de dados é um conceito fundamental em sistemas de software, relacionado à capacidade de armazenar informações de modo que permaneçam acessíveis mesmo após o término do processo que as gerou. Em aplicações corporativas, essa característica é indispensável para garantir a confiabilidade das informações, especialmente em cenários que envolvem dados sensíveis, alta concorrência e grande volume de acessos. Segundo Xu & Lei (2025), mecanismos adequados de persistência contribuem diretamente para a consistência, a segurança e a escalabilidade das aplicações. Nesse contexto, a escolha da estratégia de persistência deve considerar as necessidades específicas do projeto, bem como aspectos como desempenho,

segurança e capacidade de crescimento do sistema.

No caso da linguagem Java, a persistência de dados apresenta desafios adicionais relacionados ao ciclo de vida dos objetos, que é distinto daquele observado em algumas outras linguagens de programação. Objetos Java existem apenas em memória durante a execução da aplicação e são gerenciados automaticamente pela *Java Virtual Machine* (JVM) por meio do *Garbage Collector* (GC), sendo descartados quando não há mais referências ativas. Essa natureza transitória torna necessário o uso de mecanismos específicos para mapear o estado dos objetos para um meio persistente, como bancos de dados, de forma a garantir sua recuperação em execuções futuras da aplicação.

Essa persistência de dados em aplicações Java tem sido tradicionalmente realizada por meio do JDBC, uma *Application Programming Interface* (API) de baixo nível que permite o envio direto de comandos SQL ao banco de dados. Em contrapartida, Bonteanu *et al.* (2023) enfatizam que, apesar de fornecer controle total sobre as operações realizadas, o JDBC impõe grande responsabilidade ao desenvolvedor, que precisa lidar manualmente com conexões, declarações preparadas e transações. Esse modelo pode resultar em código excessivamente verboso, ou seja, um código com muitas linhas e escrita repetitiva para realizar tarefas que poderiam ser feitas de forma mais simples ou concisa, podendo dificultar a manutenção e aumentar a propensão de erros.

A evolução natural dessa abordagem foi o surgimento das tecnologias de mapeamento objeto-relacional, como JPA, Hibernate e, mais recentemente, o Spring Data. A principal proposta dessas tecnologias é abstrair a complexidade do acesso ao banco de dados, permitindo que o desenvolvedor interaja diretamente com objetos Java, enquanto o *framework* se encarrega de sincronizar o estado desses objetos com o banco de dados ao longo de seu ciclo de vida. Estudos recentes da Luxoft (2022) destacam que, embora o Hibernate e o JPA proporcionem uma camada de abstração eficiente, o Spring Data JPA oferece uma solução ainda mais simplificada, reduzindo significativamente a quantidade de código necessária para operações *Create, Read, Update, Delete* (CRUD). Essa abordagem facilita a manutenção e a evolução do código, especialmente em ambientes de desenvolvimento ágeis.

Apesar das vantagens evidentes na adoção de ORM, como JPA, Hibernate e, em especial, o Spring Data JPA, que simplificam o desenvolvimento e tornam o código mais limpo e sustentável, é importante reconhecer que essa abstração não vem sem custos. Conforme apontam Jovanov *et al.* (2023), essas ferramentas podem introduzir gargalos de desempenho, sobretudo

em cenários que exigem alto volume de transações ou acesso intensivo ao banco de dados. Tais limitações evidenciam que a escolha entre o controle manual proporcionado pelo JDBC e a conveniência oferecida pelos ORM não deve ser feita apenas com base na produtividade imediata, mas também considerando as demandas de escalabilidade e desempenho do sistema a longo prazo. Assim, cabe ao desenvolvedor avaliar criticamente o contexto da aplicação para decidir entre maior abstração e facilidade de uso, ou controle granular e eficiência operacional.

2.2 Comparativo Crítico entre as Tecnologias

A escolha da tecnologia de persistência de dados é uma das decisões mais importantes no desenvolvimento de sistemas em Java. A seguir, discute-se criticamente as seis principais abordagens utilizadas: JDBC, Hibernate, JPA, OpenJPA, DataNucleus e Spring Data JPA, considerando suas características técnicas, vantagens, limitações e os contextos em que cada uma se mostra mais apropriada.

2.2.1 JDBC

O JDBC (*Java Database Connectivity*) foi criado pela Sun Microsystems e lançado oficialmente em 19 de fevereiro de 1997 como parte do *Java Development Kit* (JDK) 1.1, passando desde então a integrar de forma padrão a plataforma Java. A principal característica técnica do JDBC é fornecer uma API de baixo nível para execução direta de comandos SQL, gestão de conexões, controle transacional e manipulação de resultados. Essa flexibilidade, apesar de vantajosa em determinados cenários, também representa uma de suas principais limitações: a complexidade de desenvolvimento e manutenção do código, além de exigir maior cuidado na construção manual de comandos SQL, o que pode expor a aplicação a vulnerabilidades como a *SQL Injection* quando boas práticas não são adotadas. Ainda assim, conforme Żuchnik & Kopniak (2021), o JDBC obteve o melhor desempenho bruto nas conexões diretas com o banco de dados, mas apresentou maiores custos de desenvolvimento devido à ausência de abstrações para persistência de dados.

Embora seu uso direto esteja diminuindo, Sharma (2025) defende que ele permanece essencial como base para *frameworks* mais avançados e continua sendo uma escolha viável quando é necessário controle detalhado sobre as interações com o banco de dados. O JDBC também se destaca em cenários em que o desempenho é fator crítico, especialmente em ambientes

Java nativos ou sistemas embarcados. Um estudo comparativo de Morusu (2025) apontou que, em cargas de trabalho de leitura e escrita de alto volume, o JDBC apresentou desempenho superior em relação a *frameworks* baseados em ORM, graças à ausência de camadas adicionais de abstração.

Em resumo, o JDBC é mais apropriado para sistemas legados, aplicações com requisitos rigorosos de desempenho e cenários em que o desenvolvedor precisa de controle total sobre o SQL e a gestão das conexões. No entanto, sua complexidade, maior propensão a erros de implementação, incluindo falhas de segurança quando boas práticas não são seguidas, e baixa produtividade em comparação com *frameworks* modernos justificam sua substituição em muitos projetos atuais.

2.2.2 Hibernate e JPA

A JPA é uma especificação da plataforma Java voltada à padronização do mapeamento objeto-relacional e da persistência de dados em aplicações corporativas. De acordo com um artigo publicado pela DevMedia (2014), a JPA foi introduzida oficialmente em 2006, como parte do componente *Enterprise JavaBeans* (EJB) 3.0 da plataforma Java EE 5, com o objetivo de simplificar e padronizar o processo de persistência de dados em aplicações Java, superando as limitações e a complexidade das abordagens anteriores baseadas diretamente em JDBC, oferecendo um modelo declarativo baseado em anotações, gerenciamento automático do ciclo de vida das entidades e uma linguagem de consulta orientada a objetos, a *Java Persistence Query Language* (JPQL). Desde suas primeiras versões, a JPA evoluiu para promover maior portabilidade entre diferentes provedores de persistência, permitindo que aplicações Java sejam desacopladas de implementações específicas de acesso a dados.

O Hibernate, uma das implementações mais populares da especificação JPA, teve sua origem ainda antes da própria padronização, sendo criado em 2001 como uma solução independente para mapeamento objeto-relacional. Com a consolidação da JPA, o Hibernate passou a se alinhar à especificação, tornando-se um de seus principais provedores. Atualmente, é amplamente reconhecido por sua robustez, escalabilidade e riqueza de funcionalidades para o mapeamento objeto-relacional. Entre seus principais recursos técnicos estão o cache de segundo nível, *lazy loading*, e o uso da linguagem JPQL, que juntos permitem otimizar consultas e melhorar o desempenho de aplicações de grande porte.

Estudos conduzidos por Bonteanu *et al.* (2023) evidenciam que o *framework* pode

ocasionar alto consumo de CPU em consultas complexas, além de gerar *overhead* considerável na JVM devido ao carregamento excessivo de classes e gerenciamento de *threads*. Esse *overhead* é particularmente notável em consultas em lote com JPQL mal otimizadas, o que pode impactar negativamente a escalabilidade e a responsividade do sistema.

A complexidade do Hibernate também se reflete na incidência de *bugs* em aplicações que o utilizam. Liu *et al.* (2024), em uma análise empírica de 423 *bugs* relacionados ao acesso a banco de dados em projetos Java de código aberto, identificaram que 54% dos erros estavam associados a consultas SQL diretas (com JDBC), enquanto 38,7% estavam relacionados a API de persistência, incluindo o Hibernate. Esses dados sugerem que, apesar do maior controle oferecido pelo JDBC, sua verbosidade e complexidade tendem a gerar mais erros. Por outro lado, *frameworks* como o Hibernate, embora mais seguros e produtivos, não eliminam completamente as fontes de *bugs*, especialmente quando as consultas JPQL não são cuidadosamente otimizadas.

No geral, o Hibernate é mais apropriado para sistemas que demandam alta escalabilidade, manutenção facilitada por abstrações e otimizações internas, além de suporte robusto para operações complexas em bancos de dados relacionais. Contudo, seu uso requer atenção especial à forma como as consultas são escritas e estruturadas, bem como ao gerenciamento do consumo de recursos, para evitar problemas de desempenho e garantir a estabilidade da aplicação.

2.2.3 OpenJPA e DataNucleus

As implementações OpenJPA e DataNucleus oferecem propostas alternativas dentro do ecossistema JPA, cada uma com características técnicas, vantagens e limitações distintas. O OpenJPA tem suas origens no produto de persistência Kodo, desenvolvido pela empresa SolarMetric, Inc. em 2001. Posteriormente, a SolarMetric foi adquirida pela BEA Systems em 2005 e, em 2006, grande parte do código do Kodo foi doada à Apache Software Foundation, dando origem ao projeto OpenJPA. Essa implementação é reconhecida por sua integração com outras ferramentas do ecossistema Apache e por possuir documentação abrangente. No entanto, é frequentemente criticada por sua instabilidade e pela ocorrência de *bugs* recorrentes em versões específicas, o que compromete sua confiabilidade em ambientes de produção com alta carga. Estudos recentes, como o de Połeć *et al.* (2022), evidenciam que, embora o OpenJPA possa apresentar bom desempenho em contextos específicos, seu desempenho geral tende a ser inferior ao de outras soluções, especialmente em cenários com operações intensivas de atualização ou uso *multithreaded*.

Por outro lado, o DataNucleus teve sua marca e plataforma de acesso lançadas oficialmente em abril de 2008. O *framework* se destaca por sua robustez técnica e pelo suporte a uma ampla variedade de bancos de dados relacionais e *Not Only SQL* (NoSQL). Sua arquitetura oferece desempenho superior em operações de leitura e em ambientes com múltiplas *threads*, tornando-o adequado para aplicações mais exigentes em termos de desempenho e flexibilidade. Contudo, sua principal limitação reside na complexidade de configuração, o que pode afastar desenvolvedores menos experientes ou com prazos curtos de implantação. A versão 6.0 do DataNucleus AccessPlatform trouxe melhorias significativas de desempenho, sobretudo em operações de *commit* em bancos de dados relacionais, graças à adoção de técnicas de *batching*, que otimizam a execução de operações em lote (DataNucleus, 2021).

Assim, a escolha entre OpenJPA e DataNucleus deve considerar não apenas critérios técnicos, mas também o perfil da equipe de desenvolvimento, os requisitos de desempenho da aplicação e a complexidade do ambiente de persistência. Enquanto o OpenJPA pode ser vantajoso em projetos menores ou que já utilizem tecnologias do ecossistema Apache, o DataNucleus se mostra mais adequado para sistemas mais complexos, que exigem alto desempenho e maior controle sobre a configuração do *framework*.

2.2.4 Spring Data JPA

O Spring Data JPA é uma extensão do Spring Framework, cuja primeira versão estável foi lançada em 2011. Seu principal objetivo é simplificar a implementação da camada de persistência em aplicações Java, abstraindo grande parte da complexidade envolvida na interação com bancos de dados relacionais por meio de JPA. De acordo com Bonteanu *et al.* (2023), o principal diferencial do Spring Data JPA é a redução significativa da quantidade de código que fornece uma estrutura básica para um tipo específico de tarefa, facilitando operações CRUD, consultas customizadas e paginação, além de integrar-se naturalmente com o ecossistema Spring.

Estudos recentes utilizando o YourKit Java Profiler 2022.9 revelaram que, apesar de sua facilidade de uso, o Spring Data JPA pode introduzir um *overhead* considerável em operações específicas, especialmente em exclusões em lote. Bonteanu *et al.* (2023) identificaram que o gerenciamento padrão de transações do Spring Data JPA, embora eficiente para a maioria dos casos, pode causar gargalos de desempenho em cenários que demandam alta taxa de operações em lote. Para mitigar esses impactos, a pesquisa propõe a substituição da gestão de transações padrão pela utilização direta do `EntityManagerFactory` com estratégias personalizadas, obtendo

melhorias de desempenho de até 10%.

Além disso, Gessert *et al.* (2020) enfatizam que o Spring Data JPA, por meio da especificação JPA e de implementações como o Hibernate, oferece uma integração robusta com mecanismos de *cache* de segundo nível, que funciona como um "intermediário" entre a CPU e a *Random Access Memory* (RAM), o que pode otimizar significativamente o tempo de acesso a dados em aplicações com alto volume de leitura. Todavia, sua simplicidade de uso vem acompanhada de limitações, especialmente para projetos que exigem configurações altamente customizadas, onde o nível de abstração pode dificultar o controle fino sobre operações específicas do banco.

Em termos de contexto, o Spring Data JPA é particularmente indicado para aplicações empresariais que priorizam agilidade no desenvolvimento e facilidade de manutenção, especialmente em equipes já inseridas no ecossistema Spring. Contudo, Mihalcea (2016) e Souza (2018) advertem que em projetos com demandas críticas de desempenho, especialmente em operações de escrita massiva ou consultas altamente complexas, a abstração pode gerar gargalos. Ou seja, o Spring Data JPA, quando utilizado sem otimizações de baixo nível, como *JDBC batching* ou controle explícito do contexto de persistência, pode impactar negativamente o desempenho, sendo recomendável, nesses casos, a adoção de abordagens mais diretas, como o uso do JPA com `EntityManager`, JDBC puro ou *frameworks* alternativos.

2.3 Divergência na Literatura

Embora haja consenso acerca da superioridade do Hibernate em termos de escalabilidade, alguns estudos indicam a preferência por tecnologias mais simples, como JDBC, em sistemas que demandam desempenho extremo e apresentam baixa complexidade lógica (Baran; Muryjas, 2023; Bonteanu *et al.*, 2023; Caban *et al.*, 2024). Baran & Muryjas (2023) concluem que, embora o Hibernate reduza significativamente o esforço de desenvolvimento, ela também pode introduzir impactos negativos no desempenho em razão da camada adicional de abstração. Bonteanu *et al.* (2023) avaliaram o impacto das camadas adicionais de abstração no desempenho de aplicações Java, destacando que, apesar da facilidade proporcionada no desenvolvimento, *frameworks* como o Spring Data podem ocasionar *overheads* imperceptíveis que comprometem o desempenho em ambientes críticos. Os autores também observaram que, enquanto tecnologias como JDBC e Hibernate demonstram maior longevidade e coexistência em aplicações maduras, soluções como o Spring Data tendem a ser substituídas conforme os

projetos evoluem em complexidade.

Em ambientes onde o controle fino sobre as operações de banco de dados é crucial, o uso direto de JDBC tem ganhado destaque. Raj (2025) argumenta que, embora o JPA e o Hibernate ofereçam abstrações convenientes, eles podem gerar consultas inesperadas e consumir mais memória devido ao contexto de persistência. Em contrapartida, o JDBC permite a execução de consultas otimizadas diretamente, proporcionando maior previsibilidade e eficiência.

Discussões em comunidades de desenvolvedores, também refletem essa tendência. Profissionais relatam experiências onde o uso de JDBC ou bibliotecas como o JDBCTemplate resultou em melhor desempenho e maior controle, especialmente em sistemas com grandes volumes de dados ou requisitos específicos de performance.

Portanto, a escolha entre JDBC, Hibernate e Spring Data JPA deve considerar não apenas a escalabilidade e facilidade de desenvolvimento, mas também os requisitos específicos de desempenho e a complexidade da aplicação. Em sistemas com alta demanda de desempenho e baixa complexidade, o JDBC pode ser a opção mais eficiente, enquanto *frameworks* como Hibernate e Spring Data JPA são mais adequados para aplicações com lógica de negócio mais complexa e onde a produtividade do desenvolvimento é uma prioridade.

3 METODOLOGIA

A presente pesquisa é classificada como aplicada, pois seu foco está na geração de conhecimento voltado à resolução de problemas concretos e específicos relacionados à persistência de dados em sistemas utilizando a linguagem de programação Java. Em vez de buscar apenas ampliar o conhecimento teórico sobre tecnologias de acesso a dados, esta investigação propõe soluções e orientações práticas que possam ser diretamente aplicadas por desenvolvedores e equipes de software em contextos reais de projeto e implementação.

A abordagem adotada é quantitativa, uma vez que os dados serão coletados e analisados com base em métricas objetivas e mensuráveis, tais como tempo de execução, uso de memória, *throughput* e eficiência transacional. Essa abordagem permite a comparação precisa entre diferentes tecnologias de persistência de dados e assegura a reprodutibilidade dos resultados, o que confere maior robustez e confiabilidade às conclusões obtidas.

Em relação aos objetivos da pesquisa, o estudo possui caráter descritivo e explicativo. É descritivo porque visa mapear o comportamento de tecnologias como JDBC, Hibernate, OpenJPA e Spring Data em cenários controlados, apresentando dados empíricos sobre seu desempenho em diferentes tipos de operações e volumes de dados. Ao mesmo tempo, é explicativo, pois busca compreender os fatores que influenciam essas variações de desempenho, investigando aspectos como o *overhead* introduzido por camadas de abstração, a forma como cada tecnologia lida com conexões e transações, e as implicações de seu uso em contextos específicos.

A estratégia metodológica central consiste na implementação de cenários práticos de uso, planejados para simular operações comuns em aplicações Java corporativas, como inserções em massa, consultas simples e complexas, atualizações e remoções de dados. Esses cenários foram executados com diferentes tecnologias de acesso a dados, em ambientes controlados, variando-se o volume de dados e a complexidade das operações. Conforme demonstrado por Bonteanu *et al.* (2023), em um *benchmark* com volumes entre 10 mil e 500 mil registros, o Spring Data JPA apresentou *overhead* significativo em operações de persistência em lote, sobretudo devido a chamadas repetidas ao método `executeQuery()`. Este estudo busca observar e validar esse tipo de comportamento em novos contextos experimentais. A coleta de dados ocorre por meio de ferramentas de medição de desempenho e análise de *logs*, com o uso de recursos nativos dos próprios *frameworks*, o que possibilita a observação de métricas como tempo médio de resposta, uso de CPU e memória, taxa de transferência de dados e comportamento transacional.

A escolha pela experimentação como método se justifica pela sua adequação à área

de Engenharia de Software, onde é comum a necessidade de avaliar ferramentas, *frameworks* e práticas em situações específicas de uso. Segundo Torkar *et al.* (2022), a experimentação controlada na Engenharia de Software deve ir além da coleta de dados, buscando fundamentar a significância prática das soluções tecnológicas em cenários realistas, o que justifica a análise comparativa entre diferentes *frameworks* de persistência.

Com base nesse modelo experimental, a pesquisa visa identificar as vantagens, desvantagens e contextos ideais de aplicação de cada uma das tecnologias analisadas, oferecendo um referencial prático para desenvolvedores e arquitetos de software na escolha da abordagem mais adequada às necessidades de seus projetos.

3.1 Ambiente de Testes

Os experimentos desta pesquisa foi conduzida em um ambiente controlado, projetado para garantir a confiabilidade e a reprodutibilidade dos resultados obtidos. Para isso, foi utilizado o notebook Lenovo IdeaPad Gaming 3i Gen 6 (15", Intel), equipado com processador Intel Core i7-10750H, 32 GB de memória RAM DDR4, unidade de armazenamento SSD NVMe de 512 GB e placa de vídeo dedicada NVIDIA GeForce GTX 1650. O sistema operacional adotado foi o Windows 11 Home, e o ambiente de desenvolvimento feito na *Integrated Development Environment* (IDE) IntelliJ IDEA. O DBMS utilizado sendo o PostgreSQL, em sua versão 16, selecionado por sua reconhecida robustez, desempenho estável e ampla aceitação tanto em contextos acadêmicos quanto corporativos.

As aplicações de testes foi desenvolvida na linguagem Java, utilizando o JDK 23, com o gerenciamento de dependências realizado por meio do Apache Maven, ferramenta amplamente empregada no ecossistema Java por sua eficiência e integração com diversos *frameworks*. Encontram-se implementadas quatro abordagens distintas de acesso a dados: JDBC puro, que oferece controle total sobre a execução de instruções SQL; Hibernate ORM versão 6, que fornece um mapeamento objeto-relacional mais completo e flexível; Jakarta Persistence (JPA 3), como especificação padrão para persistência no Java; e Spring Data JPA com Spring Boot 3, com foco na produtividade do desenvolvedor e na redução de código repetitivo.

Essa diversidade de tecnologias permite a comparação entre diferentes níveis de abstração e sua influência sobre o desempenho, a escalabilidade e a complexidade de implementação das aplicações. Dessa forma, o ambiente experimental estará devidamente preparado para oferecer suporte técnico adequado às análises propostas, garantindo a integridade metodológica

dos testes e a validade dos dados coletados.

3.2 Definição dos Cenários de Teste

Para garantir uma análise robusta e representativa das tecnologias de acesso a dados no ecossistema Java, esta pesquisa definiu três cenários experimentais principais, que abrangem operações fundamentais em sistemas de informação corporativos: leitura em massa, escrita em massa e operações transacionais mistas. A escolha desses cenários baseia-se na observação das demandas comuns em aplicações reais, tais como sistemas de gestão empresarial, plataformas de *e-commerce* e serviços web, onde o acesso eficiente e confiável ao banco de dados é crucial para o desempenho e a escalabilidade da aplicação.

Estudos recentes, como o de Bonteanu *et al.* (2023), ressaltam a importância de se utilizar *benchmarks* contextualizados e específicos para a avaliação de *frameworks* ORM, destacando que a análise deve ir além do simples teste de desempenho para considerar as particularidades do ambiente e dos casos de uso.

Além disso, a definição dos cenários foi inspirada por práticas recomendadas na literatura especializada, como o estudo de Traini *et al.* (2022), que enfatiza a necessidade de simular cargas de trabalho realistas e de garantir que os experimentos possam ser repetidos e comparados de forma justa. Essa abordagem metodológica possibilita a criação de testes que não apenas medem o desempenho bruto, mas também observam aspectos como estabilidade, consumo de recursos e comportamento em condições de concorrência.

3.2.1 Leitura em Massa (bulk read)

Neste cenário experimental, o objetivo principal é avaliar o desempenho das tecnologias de acesso a dados no ecossistema Java durante a recuperação de grandes volumes de informações armazenadas no banco de dados PostgreSQL. Para isso, foram realizadas consultas simples do tipo `READ` em tabelas populadas com diferentes quantidades de registros, contemplando as faixas de 100 mil a 500 mil de entradas. Essa variação controlada do volume de dados visa simular situações reais encontradas em aplicações corporativas que lidam com a extração massiva de dados, permitindo a análise do comportamento das tecnologias sob diferentes cargas.

Este estudo contribuirá para a compreensão das *trade-offs* inerentes a cada tecnologia de acesso a dados em cenários de leitura massiva, fornecendo diretrizes para a escolha da

abordagem mais adequada em termos de desempenho e recursos computacionais para aplicações Java que interagem com o PostgreSQL. A otimização de desempenho em aplicações com banco de dados é um campo vasto, e este trabalho se insere nessa área, buscando oferecer *insights* práticos e empiricamente validados.

3.2.2 Escrita em Massa (bulk insert)

Para garantir a eficiência na escrita em massa de dados, a integração de diversas estratégias de otimização é fundamental. Um pilar central é o uso de transações, que permite agrupar múltiplas operações de inserção em uma única unidade lógica. Ao fazer isso, não só reduzimos o *overhead* de comunicação com o banco de dados – já que o banco processa um bloco de trabalho de uma só vez, mas também garantimos a atomicidade das operações. Isso significa que, ou todas as inserções são bem-sucedidas, ou nenhuma delas é, preservando a integridade dos dados.

Complementar ao uso de transações, o modo *batch* é uma funcionalidade crucial, especialmente com JDBC. Ele possibilita que múltiplos comandos SQL sejam enviados ao banco de dados em uma única comunicação de rede. Essa abordagem minimiza a latência e aumenta drasticamente o *throughput*, evitando que cada inserção individual precise de uma ida e volta completa ao servidor de banco de dados.

Quando trabalhamos com ORM como Hibernate ou JPA, o gerenciamento do contexto de persistência se torna vital para operações em larga escala. É essencial implementar estratégias para gerenciar o `EntityManager` (ou `Session` no Hibernate), incluindo a limpeza periódica e a remoção de objetos do contexto. Isso impede que o contexto de persistência cresça indefinidamente, o que poderia levar a estouros de memória (*out-of-Memory errors*) e degradação de desempenho em operações com milhões de registros.

3.2.3 Operações Transacionais Mistas

Neste cenário experimental, o foco é emular um fluxo de trabalho transacional mais realista e complexo, que reflete as operações típicas de sistemas de informação em produção. Diferentemente dos cenários de leitura e escrita puras, este envolve uma combinação dinâmica de inserções, leituras e atualizações de dados, todas encapsuladas dentro de transações atômicas. O objetivo principal é avaliar não apenas o desempenho das tecnologias de acesso a dados em operações combinadas, mas a robustez da gestão de transações, concorrência e isolamento em

um ambiente *multithread*.

Para simular o comportamento de aplicações reais, o experimento teve uma série de elementos interligados. Primeiro, o fluxo de operações combinadas: cada transação simulada executará uma sequência predefinida de operações. Por exemplo, uma transação pode envolver a inserção de um novo registro, a leitura de dados relacionados, a atualização de um status e, finalmente, a gravação de um *log* da operação. Essa combinação visa testar a capacidade das tecnologias de gerenciar diferentes tipos de operações dentro de um único escopo transacional.

Em seguida, o acesso concorrente por múltiplas *threads*, ou seja, o cenário foi executado com um número configurável de *threads* simultâneas, cada uma tentando realizar suas operações transacionais. O intuito foi simular o acesso concorrente de múltiplos usuários ou processos a recursos compartilhados no banco de dados. A interação entre essas *threads* pode levar a condições de corrida, *deadlocks* e problemas de consistência de dados se o gerenciamento transacional não for adequado.

A gestão de transações e isolamento também será um ponto de atenção. Isso inclui o uso de anotações como `@Transactional` (comumente no Spring Framework) para demarcar o escopo transacional, e a forma como o `EntityManager` e a `Session` orquestram o trabalho com o banco de dados. A escolha e o comportamento dos níveis de isolamento transacional, como `Read Committed`, `Repeatable Read` e `Serializable`, foram investigados, pois eles determinam o grau de visibilidade das alterações de uma transação para outras transações concorrentes e são críticos para evitar anomalias como leituras sujas, não repetíveis e fantasmas.

Por fim, o cenário permitirá observar como as tecnologias lidam com a concorrência e estratégias de bloqueio a nível de banco de dados. Isso inclui a eficácia de bloqueios otimistas, com versionamento de dados, e pessimistas, com bloqueios explícitos no banco de dados, para garantir a integridade e consistência dos dados sob alta carga.

3.3 Métricas Avaliadas

Durante a execução dos testes, foram avaliadas métricas específicas que permitam uma comparação precisa entre as tecnologias. Entre elas, destacam-se o tempo de execução das operações, o uso de memória do sistema, a quantidade de linhas de código-fonte necessárias para a implementação de cada solução, o grau de complexidade de configuração e a taxa de *throughput*, ou seja, operações por segundo, especialmente em cenários concorrentes. Para uma comparação objetiva e técnica, foram utilizadas métricas quantitativas que refletem a performance

e a complexidade de implementação das tecnologias analisadas.

3.3.1 Tempo de Execução (ms)

O tempo de execução é uma métrica fundamental que mensura a duração total, em milissegundos, de cada operação ou conjunto de operações em cada cenário de teste. O tempo foi registrado com alta exatidão utilizando `System.nanoTime()`, que oferece granularidade em nanossegundos. Essa métrica é essencial para determinar a velocidade e a eficiência bruta de cada tecnologia na execução de tarefas de leitura, escrita e operações transacionais mistas.

3.3.2 Uso de Memória (MB)

O uso de memória, expresso em *megabytes*, é mensurado diretamente durante a execução dos testes por meio da API *Runtime* da linguagem Java. A métrica considera a diferença entre a memória total alocada pela JVM e a memória livre disponível, permitindo estimar o consumo efetivo da heap ao longo do *benchmark*. Esse monitoramento possibilita a identificação de comportamentos relacionados ao uso excessivo de memória, como criação excessiva de objetos ou ineficiências na gestão de recursos. A análise do consumo de memória é fundamental para avaliar a escalabilidade e a estabilidade das tecnologias testadas, especialmente em cenários de alta concorrência.

3.3.3 Throughput (operações por segundo)

O *throughput* quantifica o número de operações que uma solução consegue realizar por unidade de tempo, normalmente expresso em operações por segundo. Essa métrica é particularmente útil e reveladora em testes concorrentes, onde múltiplas *threads* ou processos acessam o banco de dados simultaneamente. Avaliar o *throughput* nos permite mensurar a capacidade de processamento da solução sob carga, oferecendo *insights* valiosos sobre sua escalabilidade horizontal e vertical, e sua habilidade em lidar eficientemente com a concorrência.

3.3.4 Linhas de Código-fonte (LoC)

A métrica de linhas de código-fonte (LoC) foi utilizada para quantificar a quantidade de código específico de acesso a dados necessário para implementar cada abordagem. Ao contabilizar o LoC, foi possível mensurar de forma tangível o nível de abstração fornecido por

cada tecnologia, a quantidade de código *boilerplate* exigido e, consequentemente, a produtividade que ela pode oferecer ao desenvolvedor. Menos LoC para a mesma funcionalidade pode indicar uma API mais concisa e de uso mais fácil.

3.3.5 Complexidade de Configuração

Por fim, a complexidade de configuração é avaliada de forma qualitativa, com base em critérios técnicos e de usabilidade. Consideram-se os passos necessários para configurar cada tecnologia, como a definição de *beans* do Spring, a criação de arquivos de configuração específicos (.xml ou .properties) ou o uso de classes de configuração Java. Essa avaliação fundamenta-se em critérios de qualidade de software, como os propostos pela norma ISO/IEC 9126, que contempla fatores como compreensibilidade, operabilidade e adaptabilidade. Uma menor complexidade de configuração associa-se, em geral, a menor tempo de *setup* e a menor propensão a erros, impactando positivamente a agilidade no desenvolvimento e na manutenção do projeto.

3.4 Execução dos Testes

Para garantir que os resultados deste estudo sejam estatisticamente confiáveis e representem com fidelidade o desempenho das tecnologias avaliadas, adota-se um protocolo de execução de testes rigoroso e padronizado. O objetivo central consiste em eliminar interferências externas que possam distorcer as medições, assegurando comparações justas e objetivas. Cada cenário de teste é repetido cinco vezes consecutivas. Essa estratégia de múltiplas execuções permite capturar a variabilidade natural do sistema e aumentar a significância estatística dos dados. Para evitar efeitos residuais de execuções anteriores, aplica-se uma rotina de limpeza completa entre cada repetição: o banco de dados é reiniciado a partir de um *snapshot* inicial limpo, garantindo a ausência de resquícios de dados ou estados de *cache* que possam influenciar a performance. Paralelamente, a aplicação Java é reiniciada a cada ciclo. Essa dupla reinicialização neutraliza efeitos como o aquecimento da JVM, o *cache* da aplicação e contenções de recursos acumuladas, assegurando que cada medição se inicie a partir de um estado limpo. A rastreabilidade e a reprodutibilidade dessa preparação são garantidas pelo armazenamento e versionamento de todos os *scripts* de limpeza e preparação do ambiente, escritos em SQL, no repositório do projeto, o que possibilita a replicação exata das condições de teste por terceiros.

A execução dos testes ocorre em um ambiente de controle estrito e isolado. A máquina de teste permanece sem outras aplicações ativas em paralelo e sem conexão com a internet, minimizando fontes potenciais de ruído ou interferência externa que possam comprometer a precisão dos resultados. A coleta de dados de *Garbage Collection* mantém-se habilitada, permitindo a análise da interação de cada tecnologia com o ciclo de vida da memória. Para a simulação de cenários realistas de concorrência, comuns em sistemas de produção, utilizam-se múltiplos *threads* orquestrados por um `ExecutorService`. Essa estratégia possibilita controle preciso do paralelismo e assegura a sincronização adequada entre operações executadas simultaneamente, emulando o comportamento de múltiplos usuários interagindo com o sistema.

Por fim, a transparência e a reprodutibilidade do estudo constituem princípios centrais da metodologia. Todos os testes desenvolvidos, os *scripts* de preparação do ambiente e as dependências do projeto encontram-se documentados no arquivo `pom.xml` e versionados por meio do Git. Essa prática assegura a rastreabilidade das versões do código-fonte e das configurações, tornando o experimento passível de replicação, verificação e expansão por outros pesquisadores ou profissionais da área.

3.5 Análise Comparativa

A análise dos dados coletados nesta pesquisa é conduzida por meio de uma abordagem mista, que combina métodos estatísticos descritivos e avaliação qualitativa, de modo a proporcionar uma compreensão abrangente dos resultados. Os dados de desempenho organizam-se em tabelas estruturadas contendo indicadores como médias, desvios padrão, valores mínimos e máximos referentes a cada rodada de testes. Esses indicadores possibilitam uma comparação precisa entre as tecnologias de acesso a dados analisadas, considerando diferentes cenários operacionais, como leitura em massa, escrita em massa e operações transacionais mistas.

Com base nesses dados, elaboram-se visualizações gráficas, como gráficos de barras, com o propósito de ilustrar comparações relacionadas ao tempo médio de execução, uso de memória e *throughput*. Ferramentas como Microsoft Excel e Google Sheets são utilizadas para a construção dessas visualizações, em razão de sua acessibilidade e eficiência na organização e apresentação de dados experimentais. Esse tratamento visual facilita a interpretação dos resultados e evidencia padrões de comportamento entre as abordagens adotadas (JDBC, Hibernate, OpenJPA e Spring Data JPA).

Paralelamente à análise quantitativa, conduz-se uma avaliação qualitativa com foco

na experiência de desenvolvimento proporcionada por cada tecnologia. Essa etapa considera critérios como facilidade de configuração inicial, clareza e legibilidade do código produzido, suporte à depuração, documentação disponível e simplicidade na manutenção do código-fonte. Tais aspectos mostram-se especialmente relevantes em ambientes corporativos, nos quais a produtividade da equipe e a sustentabilidade do projeto ao longo do tempo constituem fatores críticos.

A adoção dessa abordagem mista encontra respaldo em estudos metodológicos contemporâneos, como os de Yin (2018), Flick (2022), que defendem o uso de estratégias híbridas em pesquisas aplicadas na área de engenharia de software. Segundo esses autores, a combinação entre análise estatística objetiva e avaliação qualitativa contribui para uma visão mais rica e contextualizada dos fenômenos investigados, permitindo que os resultados não se limitem a métricas numéricas, mas também incorporem fatores humanos e organizacionais. Desse modo, os achados da pesquisa fornecem subsídios sólidos tanto do ponto de vista técnico quanto prático, auxiliando desenvolvedores, arquitetos de software e tomadores de decisão na escolha das tecnologias de persistência de dados em aplicações Java.

4 RESULTADOS

Este capítulo tem como propósito apresentar os resultados obtidos durante o desenvolvimento e a implementação do projeto, destacando aspectos relacionados à arquitetura, à execução dos testes de desempenho e à padronização das operações de persistência. Além disso, busca-se evidenciar as escolhas de design e as estratégias adotadas para garantir modularidade, reprodutibilidade e extensibilidade, permitindo uma análise consistente entre diferentes tecnologias de persistência.

4.1 Aspectos de Desenvolvimento

Esta seção tem como objetivo apresentar os aspectos técnicos de implementação e elucidar como as tecnologias utilizadas foram empregadas no desenvolvimento e na realização dos requisitos do *benchmark*. Ao longo do desenvolvimento, buscou-se criar uma ferramenta que possibilitasse a execução sistemática de testes de desempenho, garantindo modularidade, extensibilidade e reprodutibilidade dos resultados, assim como consistência entre os diferentes módulos de persistência avaliados.

Foram aplicados princípios de reutilização de código, padronização e separação de responsabilidades, garantindo que cada módulo fosse implementado de forma independente e integrável ao projeto. A arquitetura do *benchmark* permite a execução automatizada dos testes, com registro detalhado de métricas de desempenho, como tempo de inserção, atualização, exclusão e consulta, assegurando reprodutibilidade e confiabilidade dos resultados. Além disso, a estrutura modular favorece a extensibilidade, permitindo a inclusão de novas tecnologias sem alterações significativas no código-base, e a manutenibilidade, com responsabilidades claras para cada módulo.

Em síntese, o desenvolvimento do *benchmark* resultou em uma ferramenta confiável, modular e extensível, capaz de fornecer métricas precisas e servir como base sólida para comparações sistemáticas entre diferentes tecnologias de persistência em Java.

4.1.1 Estrutura Geral do Projeto

O projeto foi desenvolvido com o objetivo de realizar uma análise comparativa entre diferentes tecnologias de persistência de dados na plataforma Java, com foco em desempenho, simplicidade e manutenção do código. Para isso, foi adotada uma arquitetura modular, utilizando

o sistema de construção Maven, que permitiu a separação de cada tecnologia em um módulo independente, mas com uma base de código comum. A estrutura geral do projeto foi dividida nos seguintes módulos principais:

- **common**: contém as entidades, classes utilitárias, configurações e *scripts* SQL compartilhados entre os demais módulos. Esse módulo garante a padronização das estruturas de dados e facilita a reutilização de código.
- **jdbc**: implementa operações de persistência utilizando a API padrão Java Database Connectivity.
- **hibernate**: implementa o mesmo conjunto de operações utilizando o *framework* Hibernate, que fornece mapeamento objeto-relacional e abstração de SQL.
- **jpa**: contém a implementação com a Java Persistence API padrão, fornecendo uma camada de abstração sobre o Hibernate, mas mantendo um controle mais explícito sobre o ciclo de vida das entidades.
- **spring-data**: implementa a mesma lógica utilizando o Spring Data JPA, com o objetivo de avaliar o ganho em produtividade e simplicidade em relação às demais abordagens.

O projeto segue o paradigma orientado a objetos, empregando boas práticas de programação e o padrão de projeto *Data Access Object* (DAO). Além disso, foi utilizado o Git como sistema de controle de versão, e o ambiente de desenvolvimento foi configurado na IDE IntelliJ IDEA.

4.1.2 Módulo Common

O módulo *common* atua como o núcleo de recursos compartilhados entre os demais módulos do sistema. Ele centraliza funcionalidades genéricas, reutilizáveis e independentes de contexto, evitando duplicação de código e promovendo uma arquitetura mais limpa, modular e fácil de manter.

A base de qualquer experimento de desempenho é uma configuração de banco de dados consistente e controlada. Para isso, o módulo *common* disponibiliza as classes `DbConfig` e `DbUtils`, responsáveis por centralizar as definições de acesso ao PostgreSQL e preparar o ambiente antes de cada rodada de testes.

A classe `DbConfig` define constantes como *host*, porta, nome do banco, usuário e senha, além de fornecer métodos para criar um objeto `DataSource` configurado com `PGSimpleDataSource`, uma implementação nativa do *driver* PostgreSQL. Essa abordagem

elimina a necessidade de configurações manuais repetidas e facilita o gerenciamento de conexões de forma segura e padronizada. Já a classe `DbUtils` disponibiliza métodos auxiliares que auxiliam na preparação do banco de dados, como o método `truncatePersonTable()`, que limpa completamente a tabela `person` e reinicia o contador de identidade antes de cada teste. Dessa forma, cada execução do *benchmark* ocorre em condições idênticas, evitando interferências de dados residuais de execuções anteriores. De acordo com Richards & Ford (2020), o encapsulamento das configurações de infraestrutura, como o acesso a bancos de dados, e a clara separação de responsabilidades são práticas essenciais para garantir manutenibilidade, consistência e facilidade de evolução de sistemas de software.

Com o objetivo de centralizar os tipos de operações que podem ser avaliadas pelo sistema de *benchmark*, foi criada a enumeração `BenchmarkType`. Essa estrutura define os valores `INSERT`, `READ`, `UPDATE` e `MIXED`, permitindo que cada execução seja configurada para focar em um tipo específico de operação ou em uma combinação delas. Dessa forma, o sistema torna-se mais flexível e modular, possibilitando comparações controladas entre diferentes perfis de carga e estratégias de acesso ao banco de dados.

A execução dos testes é configurada por meio da classe `BenchmarkConfig`, que define os parâmetros que controlam o comportamento de cada experimento. Essa classe segue o padrão de *data class*, reunindo variáveis essenciais como o número de registros processados (*records*), o tamanho dos lotes (*batchSize*), a quantidade de *threads* simultâneas (*threads*) e o tipo de teste a ser executado (*type*) — podendo ser de inserção, leitura, atualização ou misto. Além disso, o atributo `warmup` permite a execução de uma rodada de aquecimento, essencial para *benchmarks* em Java, uma vez que o compilador *Just-In-Time* (JIT) otimiza o código dinamicamente após algumas iterações iniciais. Essa etapa é indispensável para evitar que as otimizações graduais do JIT distorçam os resultados obtidos, garantindo medições mais precisas. A classe também inclui o atributo `verbose`, responsável por controlar o nível de detalhamento das informações exibidas durante a execução dos testes, permitindo maior transparência na análise dos resultados. Conforme apontam Traini *et al.* (2022), práticas adequadas de *warm-up* e coleta de métricas são fundamentais para assegurar medições consistentes e representativas em aplicações Java, visto que a ausência de um aquecimento apropriado pode levar a interpretações incorretas sobre a eficiência do sistema e comprometer a validade dos experimentos.

Após a execução dos testes, os resultados são representados pela classe `BenchmarkResult`, que armazena informações detalhadas sobre o desempenho obtido. Entre os dados registrados

estão o tempo total de execução, o uso de memória e o número de operações realizadas em cada categoria. A classe também fornece métodos utilitários que facilitam a análise de desempenho, como `getDurationMillis()` (Código-fonte 1), responsável por calcular a duração total do experimento, e `getThroughputRecordsPerSec()` (Código-fonte 1), que determina a taxa de transferência de registros por segundo. Essas métricas permitem comparar diferentes configurações de execução e avaliar a eficiência do sistema sob diversas condições. Segundo Fruth *et al.* (2021), métricas de *throughput* e latência média são essenciais em *benchmarks* de bases de dados, pois refletem tanto a capacidade de processamento quanto a responsividade da aplicação sob carga.

Código-fonte 1 – Alguns Métodos Utilitários da Classe BenchmarkResult

```
public long getDurationMillis() {
    return endTimeMillis - startTimeMillis;
}

public double getThroughputRecordsPerSec() {
    double seconds = getDurationMillis() / 1000.0;
    int total = inserts + reads + updates;
    return seconds > 0 ? total / seconds : 0.0;
}
```

A interface `BenchmarkScenario`, apresentada no Código-fonte 2, define o contrato para qualquer cenário de teste de desempenho. Ela exige apenas um método: `run(BenchmarkConfig config)`, responsável por executar o *benchmark* de acordo com as configurações fornecidas e retornar um objeto `BenchmarkResult`. Essa abstração permite a criação de diferentes tipos de cenários de *benchmark*, sem acoplar o código à implementação específica.

Código-fonte 2 – Interface BenchmarkScenario

```
public interface BenchmarkScenario {

    BenchmarkResult run(BenchmarkConfig config) throws Exception;

}
```

A interface `DatabaseWorker`, que pode ser visualizada no Código-fonte 3, define

o contrato para as operações que serão realizadas no banco de dados durante os testes de desempenho. Ela abstrai a camada de persistência, permitindo que diferentes implementações, como os escolhidos por esse trabalho: JDBC puro, JPA, Hibernate e Spring Data JPA, possam ser testadas de forma intercambiável, desde que implementem os métodos definidos. Cada método: `doInserts`, `doReads`, `doUpdates`, recebe um objeto `BenchmarkConfig`, o que garante que as operações sejam executadas de acordo com os parâmetros do experimento.

Código-fonte 3 – Interface DatabaseWorker

```
1 public interface DatabaseWorker {
2
3     int doInserts(BenchmarkConfig config) throws Exception;
4
5     int doReads(BenchmarkConfig config) throws Exception;
6
7     int doUpdates(BenchmarkConfig config) throws Exception;
8
9 }
```

A execução é coordenada pela classe `ConcurrentBenchmarkExecutor`, que implementa a interface `BenchmarkScenario` e utiliza a API `ExecutorService` do Java para gerenciar um *pool* de *threads*, garantindo que as operações sejam realizadas em paralelo de acordo com o número de *threads* definido na configuração. Essa separação entre o executor de *benchmark* (`ConcurrentBenchmarkExecutor`) e o trabalhador de banco (`DatabaseWorker`) segue o princípio de inversão de dependência, facilitando a extensibilidade e a comparação entre diferentes abordagens. Cada *worker* é responsável por executar um conjunto de operações no banco de dados e retornar resultados individuais, que são então consolidados em um único `BenchmarkResult`. O uso de *threads pools* e *futures* segue boas práticas modernas de concorrência em Java, conforme descrito por Goetz *et al.* (2021) no livro Java Concurrency in Practice.

4.1.3 Módulo JDBC

O módulo JDBC representa a forma mais tradicional e de baixo nível de acesso a bancos de dados relacionais em Java. Nesta implementação, as operações de inserção, atualização, exclusão e busca foram realizadas por meio de comandos SQL explícitos, executados através

de objetos `Connection`, `PreparedStatement` e `ResultSet`. Embora o JDBC proporciona controle total sobre o SQL executado, ele também exige maior esforço de codificação e tratamento manual de exceções e conexões.

Para inserir grandes volumes de dados de forma eficiente, foi utilizada a abordagem de inserções em massa com *batch insert*, que agrupa múltiplas instruções SQL em uma única chamada ao banco. Esta estratégia reduz a sobrecarga de comunicação e aumenta o desempenho, especialmente em cenários com milhares de registros. A classe `JdbcBulkInserter`, apresentada no Código-fonte 4, implementa essa funcionalidade, conectando-se ao PostgreSQL via `DriverManager` e executando lotes de inserções com `PreparedStatement`. Cada lote é processado e confirmado com *commit*, garantindo consistência transacional e eficiência operacional. Komanov (2021) demonstrou que o uso de *batch inserts* em JDBC aumenta significativamente o *throughput* de inserção, comparado a execuções individuais de `INSERT` em massa.

Código-fonte 4 – Classe JdbcBulkInserter

```
1  public class JdbcBulkInserter {
2
3      public int run(BenchmarkConfig config) throws Exception {
4          int total = config.getRecords();
5          int batchSize = config.getBatchSize();
6          int processed = 0;
7          try (Connection conn = DriverManager.getConnection(url, user,
       pass)) {
8              conn.setAutoCommit(false);
9              try (PreparedStatement ps = conn.prepareStatement(
10                     "INSERT INTO person (name, age) VALUES (?, ?)"))
       {
11                 for (int i = 1; i <= total; i++) {
12                     ps.setString(1, "Pessoa_" + i);
13                     ps.setInt(2, 20 + (i % 50));
14                     ps.addBatch();
15                     if (i % batchSize == 0) {
16                         ps.executeBatch();
17                         conn.commit();
18                         processed += batchSize;
19                         if (config.isVerbose())
20                             System.out.println("[JDBC] Inseridos: " +
```

```
    processed);
21                      }
22                  }
23                  ps.executeBatch();
24                  conn.commit();
25              }
26          }
27          return total;
28      }
29
30  }
```

Para leituras em massa, o foco é recuperar grandes volumes de registros de forma eficiente, minimizando o uso de memória. A estratégia adotada configura o `fetchSize` do `PreparedStatement` para 5000 registros, permitindo que os dados sejam carregados em blocos e processados sequencialmente. Isso reduz a pressão sobre a memória e proporciona uma medição mais precisa do *throughput* do banco de dados, sem a sobrecarga de *frameworks* ORM. A classe que implementa essa funcionalidade é a `JdbcBulkReader`, apresentada no Código-fonte 5, que percorre todos os registros da tabela `person` e, opcionalmente, gera *logs* detalhados para acompanhamento do progresso.

Código-fonte 5 – Classe JdbcBulkReader

```
1  public class JdbcBulkReader {
2
3      public int run(BenchmarkConfig config) throws Exception {
4          int count = 0;
5          try (Connection conn = DriverManager.getConnection(url, user,
       pass);
6               PreparedStatement ps = conn.prepareStatement(
7                       "SELECT id, name, age FROM person LIMIT " +
       config.getRecords()
8               )) {
9              ps.setFetchSize(5000);
10             try (ResultSet rs = ps.executeQuery()) {
11                 while (rs.next()) {
12                     rs.getInt("id");
13                     rs.getString("name");
```

```
14                      rs.getInt("age");
15                      count++;
16                      if (config.isVerbose() && count % 100000 == 0)
17                          System.out.println("[JDBC] Lidos " + count +
    " registros");
18                  }
19              }
20          }
21          return count;
22      }
23
24  }
```

Já a classe `JdbcBulkUpdater`, que pode ser visualizada no Código-fonte 6, realiza atualizações em massa na tabela `person`, utilizando *batch updates*. O funcionamento é semelhante ao `JdbcBulkInserter`, mas aplicando alterações em registros existentes, neste caso incrementando a idade (`age`) de cada pessoa. O processamento em lotes e o controle manual de transações garantem que o desempenho seja maximizado sem comprometer a consistência dos dados.

Código-fonte 6 – Classe JdbcBulkUpdater

```
1  public class JdbcBulkUpdater {
2
3      public int run(BenchmarkConfig config) throws Exception {
4          int total = config.getRecords();
5          int batchSize = config.getBatchSize();
6          int processed = 0;
7          try (Connection conn = DriverManager.getConnection(url, user,
       pass)) {
8              conn.setAutoCommit(false);
9              try (PreparedStatement ps = conn.prepareStatement(
10                     "UPDATE person SET age = age + 1 WHERE id = ?"))
       {
11                 for (int i = 1; i <= total; i++) {
12                     ps.setInt(1, i);
13                     ps.addBatch();
14                     if (i % batchSize == 0) {
```

```
15                          ps.executeBatch();
16                          conn.commit();
17                          processed += batchSize;
18                          if (config.isVerbose())
19                              System.out.println("[JDBC] Atualizados "
    + processed);
20                      }
21                  }
22                  ps.executeBatch();
23                  conn.commit();
24              }
25          }
26          return total;
27      }
28
29  }
```

Para integrar todas essas operações ao executor de *benchmarks*, foi criada uma camada de abstração que encapsula as operações de inserção, leitura e atualização, permitindo que o executor gerencie a execução concorrente sem depender de detalhes específicos de cada operação. A classe `JdbcWorker`, onde os métodos são mostrados no Código-fonte 7, implementa a interface `DatabaseWorker` e delega cada método à respectiva classe de implementação, seguindo o princípio de inversão de dependência. Isso facilita a troca de implementações e a comparação entre diferentes estratégias de acesso ao banco, mantendo a execução padronizada e modular (Goetz *et al.*, 2021).

Código-fonte 7 – Trecho da Classe JdbcWorker

```
1       @Override
2       public int doInserts(BenchmarkConfig config) throws Exception {
3           if (config.isVerbose())
4               System.out.println("[JDBC] Iniciando inserções em massa
    com batchSize=" + config.getBatchSize());
5           return inserter.run(config);
6       }
7
8       @Override
9       public int doReads(BenchmarkConfig config) throws Exception {
```

```
10          if (config.isVerbose())
11              System.out.println("[JDBC] Iniciando leitura em massa ("
    + config.getRecords() + " registros)...");
12          return reader.run(config);
13      }
14
15      @Override
16      public int doUpdates(BenchmarkConfig config) throws Exception {
17          if (config.isVerbose())
18              System.out.println("[JDBC] Iniciando atualização em massa
    ...");
19          return updater.run(config);
20      }
```

Por fim, para testar e consolidar os resultados, o fluxo de *benchmark* é coordenado pela `Main` (Código-fonte 8), que executa todas as operações em paralelo e coleta métricas de desempenho. Primeiro, a tabela `person` é limpa, em seguida, uma configuração de *benchmark* define o número de registros, tamanho de lote, *threads* e tipo de teste. O `ConcurrentBenchmarkExecutor` gerencia a execução concorrente utilizando o `JdbcWorker`, consolidando os resultados em um `BenchmarkResult` exibido no console. Esse processo permite analisar o desempenho do banco sob diferentes cargas e cenários de concorrência.

Código-fonte 8 – Classe Main para o JDBC

```
1  public class Main {
2
3      public static void main(String[] args) throws Exception {
4          DbUtils.truncatePersonTable(DbConfig.getDataSource());
5          BenchmarkConfig config = new BenchmarkConfig(
6                  100000,
7                  1000,
8                  4,
9                  false,
10                 "Teste JDBC Multithread",
11                 true,
12                 BenchmarkType.MIXED
13         );
14         JdbcWorker worker = new JdbcWorker();
```

```
15          ConcurrentBenchmarkExecutor executor = new
    ConcurrentBenchmarkExecutor(worker);
16          var result = executor.run(config);
17          System.out.println(result);
18      }
19
20  }
```

4.1.4 Módulo Hibernate

O módulo Hibernate utiliza o popular *framework* ORM para realizar o mapeamento entre objetos Java e tabelas do banco de dados. A configuração foi feita através de um arquivo `hibernate.cfg.xml`, especificando o dialeto *SQL*, credenciais de acesso e classes mapeadas.

A configuração e o gerenciamento das sessões do Hibernate são centralizados na classe `HibernateUtil`, apresentada no Código-fonte 9, que encapsula a criação do `SessionFactory`. Esse objeto é responsável por fornecer sessões (`Session`) para interação com o banco de dados, garantindo que todas as operações de persistência, leitura e atualização utilizem uma configuração consistente e eficiente. A construção do `SessionFactory` é realizada de forma estática, lendo o arquivo `hibernate.cfg.xml` e registrando mapeamentos como `Person.hbm.xml`. Essa abordagem evita a necessidade de recriar sessões repetidamente, reduzindo o *overhead* e melhorando a performance em *benchmarks* de grande escala. Em caso de falhas na inicialização, a classe lança uma `ExceptionInInitializerError`, garantindo que problemas críticos de configuração sejam detectados imediatamente.

Código-fonte 9 – Classe HibernateUtil

```
1  public class HibernateUtil {
2
3      private static final SessionFactory sessionFactory =
    buildSessionFactory();
4
5      private static SessionFactory buildSessionFactory() {
6          try {
7              StandardServiceRegistry registry = new
    StandardServiceRegistryBuilder()
8                      .configure()
```

```
                    .build();
            return new MetadataSources(registry)
                    .addResource("Person.hbm.xml")
                    .buildMetadata()
                    .buildSessionFactory();
        } catch (Throwable ex) {
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }

}
```

As inserções em massa são realizadas pela classe `HibernateBulkInserter`, que utiliza `Session` do Hibernate com *commit* periódico a cada lote definido pelo `batchSize`. Cada objeto `Person` é persistido no banco, e após atingir o tamanho do lote, a sessão é descarregada e limpa, garantindo que a memória utilizada não cresça indefinidamente. Essa estratégia permite alta eficiência na gravação de milhares de registros, mantendo a consistência transacional e evitando sobrecarga de memória.

Código-fonte 10 – Classe HibernateBulkInserter

```
public class HibernateBulkInserter {

    public int run(BenchmarkConfig config) {
        int total = config.getRecords();
        int batchSize = config.getBatchSize();
        int processed = 0;
        try (Session session = HibernateUtil.getSessionFactory().
    openSession()) {
            Transaction tx = session.beginTransaction();
            for (int i = 1; i <= total; i++) {
                session.persist(new Person(
                        "Pessoa_" + i,
                        "pessoa" + i + "@email.com",
```

```
13                          i
14                  ));
15                  if (i % batchSize == 0) {
16                      session.flush();
17                      session.clear();
18                      tx.commit();
19                      tx = session.beginTransaction();
20                      processed = i;
21                      if (config.isVerbose())
22                          System.out.println("[Hibernate] Inseridos: "
    + processed);
23                  }
24              }
25              tx.commit();
26          }
27          return processed;
28      }
29
30  }
```

A leitura em massa é realizada pela classe `HibernateBulkReader`, que utiliza `ScrollableResults` com `ScrollMode.FORWARD_ONLY` e *fetch size* configurado. Essa abordagem permite percorrer grandes conjuntos de dados sem carregar todos os registros na memória de uma só vez, o que é essencial para *benchmarks* de desempenho. A sessão é limpa a cada lote lido, garantindo que a utilização de memória permaneça estável mesmo em leituras de centenas de milhares de registros.

Código-fonte 11 – Classe HibernateBulkReader

```
1  public class HibernateBulkReader {
2
3      public int run(BenchmarkConfig config) {
4          int count = 0;
5          int fetchSize = 5000;
6          try (Session session = HibernateUtil.getSessionFactory().
    openSession()) {
7              Transaction tx = session.beginTransaction();
8              ScrollableResults scrollableResults = session.createQuery
```

```
    ("FROM Person")
                    .setFetchSize(fetchSize)
                    .setMaxResults(config.getRecords())
                    .scroll(ScrollMode.FORWARD_ONLY);
            while (scrollableResults.next()) {
                scrollableResults.get();
                count++;
                if (config.isVerbose() && count % 100000 == 0) {
                    System.out.println("[Hibernate] Lidos " + count +
     " registros");
                }
                if (count % fetchSize == 0) {
                    session.clear();
                }
                if (count >= config.getRecords()) {
                    break;
                }
            }
            tx.commit();
        }
        return count;
    }

}
```

As atualizações em massa são tratadas pela classe `HibernateBulkUpdater`, utilizando `StatelessSession` do Hibernate 6 e `MutationQuery`. Essa técnica permite aplicar alterações diretamente no banco, evitando *overhead* de gerenciamento de entidades pelo contexto de persistência, e executando *commits* periódicos conforme o tamanho do lote. Isso garante que o processamento de grandes volumes de dados seja rápido, mantendo a consistência das transações.

Código-fonte 12 – Classe HibernateBulkUpdater

```
public class HibernateBulkUpdater {

    public int run(BenchmarkConfig config) {
        int total = config.getRecords();
```

```
5           int batchSize = config.getBatchSize();
6           int processed = 0;
7           try (StatelessSession session = HibernateUtil.
    getSessionFactory().openStatelessSession()) {
8               Transaction tx = session.beginTransaction();
9               for (int i = 1; i <= total; i++) {
10                  session.createMutationQuery("UPDATE Person SET age =
    age + 1 WHERE id = :id")
11                          .setParameter("id", (long) i)
12                          .executeUpdate();
13                  if (i % batchSize == 0) {
14                      tx.commit();
15                      if (config.isVerbose()) {
16                          System.out.println("[Hibernate] Atualizados "
     + i);
17                      }
18                      tx = session.beginTransaction();
19                  }
20              }
21              tx.commit();
22              processed = total;
23          }
24          return processed;
25      }
26
27  }
```

A classe `HibernateWorker` implementa a interface `DatabaseWorker` e atua como camada de integração entre o executor de *benchmark* (`ConcurrentBenchmarkExecutor`) e as operações específicas de Hibernate. Ela encapsula a lógica de inserção, leitura e atualização, permitindo que o executor execute os testes de forma padronizada e concorrente. A separação de responsabilidades entre o *worker* e o executor segue princípios de inversão de dependência e facilita a troca de estratégias de acesso a dados sem modificar a lógica do executor.

Código-fonte 13 – Trecho da Classe HibernateWorker

```
1       @Override
2       public int doInserts(BenchmarkConfig config) throws Exception {
```

```
3         if (config.isVerbose())
4             System.out.println("[Hibernate] Iniciando inserções em
    massa=" + config.getBatchSize());
5         return inserter.run(config);
6     }
7
8     @Override
9     public int doReads(BenchmarkConfig config) throws Exception {
10        if (config.isVerbose())
11            System.out.println("[Hibernate] Iniciando leitura em
    massa (" + config.getRecords() + " registros)...");
12        return reader.run(config);
13    }
14
15    @Override
16    public int doUpdates(BenchmarkConfig config) throws Exception {
17        if (config.isVerbose())
18            System.out.println("[Hibernate] Iniciando atualização em
    massa...");
19        return updater.run(config);
20    }
```

Por fim, a classe `Main` demonstra a execução do *benchmark* completo utilizando Hibernate. O `HibernateWorker` é passado para o `ConcurrentBenchmarkExecutor`, que gerencia a execução concorrente e consolida os resultados em um `BenchmarkResult`. Esse fluxo permite avaliar o desempenho do Hibernate em operações de inserção, leitura e atualização sob diferentes condições de carga e concorrência, comparando, assim, o desempenho de um ORM.

Código-fonte 14 – Classe Main para o Hibernate

```
1 public class Main {
2
3     public static void main(String[] args) throws Exception {
4         DbUtils.truncatePersonTable(DbConfig.getDataSource());
5         BenchmarkConfig config = new BenchmarkConfig(
6                 100000,
7                 1000,
8                 4,
```

```
                false,
                "Teste Hibernate Multithread",
                true,
                BenchmarkType.MIXED
        );
        HibernateWorker worker = new HibernateWorker();
        ConcurrentBenchmarkExecutor executor = new
    ConcurrentBenchmarkExecutor(worker);
        var result = executor.run(config);
        System.out.println(result);
     }

}
```

4.1.5 Módulo JPA

O módulo JPA foi desenvolvido com o objetivo de explorar e avaliar o comportamento da especificação oficial de persistência da plataforma Java *Enterprise Edition* (EE), fornecendo uma base sólida para o gerenciamento de dados de forma eficiente e consistente. Para isso, o Hibernate foi adotado como *provider*, permitindo a implementação de estratégias avançadas de persistência.

As inserções em massa são implementadas com uma estratégia de *batch insert*, onde múltiplos objetos `PersonJPA` são persistidos dentro de uma única transação. A cada lote definido pelo tamanho do *batch*, o `EntityManager` é limpo e os dados são gravados no banco com *flush* e *commit*, garantindo consistência transacional e eficiência operacional. Essa abordagem reduz a sobrecarga de comunicação com o banco e melhora o desempenho em cenários de milhares de registros, como discutido por Bonteanu *et al.* (2023), que analisaram a performance de operações CRUD em aplicações Java utilizando JPA e destacaram os benefícios do *batch insert* em grandes volumes de dados.

Código-fonte 15 – Classe JpaBulkInserter

```
public class JpaBulkInserter {

    public int run(BenchmarkConfig config) {
        int total = config.getRecords();
```

```
5           int batchSize = config.getBatchSize();
6           int processed = 0;
7           EntityManager em = emf.createEntityManager();
8           em.getTransaction().begin();
9           for (int i = 1; i <= total; i++) {
10              PersonJPA p = new PersonJPA("Pessoa_" + i, null, 20 + (i
    % 50));
11              em.persist(p);
12              if (i % batchSize == 0) {
13                  em.flush();
14                  em.clear();
15                  em.getTransaction().commit();
16                  if (config.isVerbose()) {
17                      System.out.println("[JPA] Inseridos: " + i);
18                  }
19                  em.getTransaction().begin();
20              }
21          }
22          em.getTransaction().commit();
23          em.close();
24          return total;
25      }
26
27  }
```

A leitura em massa é realizada utilizando consultas paginadas com `TypedQuery`, configurando o número de resultados por lote (`setMaxResults`) e controlando o deslocamento inicial (`setFirstResult`). Cada lote de entidades é carregado sequencialmente, garantindo que o consumo de memória permaneça estável, mesmo em bancos de dados com grande volume de dados. Após processar cada lote, o `EntityManager` é limpo para liberar memória e manter o desempenho consistente durante o *benchmark*.

Código-fonte 16 – Classe JpaBulkReader

```
1  public class JpaBulkReader {
2
3      public int run(BenchmarkConfig config) {
4          int count = 0;
```

```
5           int fetchSize = 5000;
6           EntityManager em = emf.createEntityManager();
7           em.getTransaction().begin();
8           int firstResult = 0;
9           List<PersonJPA> batch;
10          do {
11              TypedQuery<PersonJPA> query = em.createQuery(
12                      "SELECT p FROM PersonJPA p", PersonJPA.class)
13                      .setFirstResult(firstResult)
14                      .setMaxResults(fetchSize);
15
16              batch = query.getResultList();
17              for (PersonJPA p : batch) {
18                  count++;
19                  if (count >= config.getRecords()) {
20                      em.getTransaction().commit();
21                      em.close();
22                      System.out.printf(
23                              "[JPA] Leitura interrompida após %d
    registros (limite configurado)%n",
24                              count
25                      );
26                      return count;
27                  }
28              }
29              if (config.isVerbose() && count % 100000 == 0) {
30                  System.out.println("[JPA] Lidos " + count + "
    registros");
31              }
32              firstResult += fetchSize;
33              em.clear();
34          } while (!batch.isEmpty());
35          em.getTransaction().commit();
36          em.close();
37          return count;
38      }
39
40  }
```

As atualizações em massa seguem lógica similar às inserções, mas aplicando alterações sobre entidades já persistidas. Cada objeto é carregado pelo `EntityManager`, atualizado e, a cada lote, o *flush* e *commit* são executados, garantindo que as alterações sejam gravadas no banco de forma eficiente. *Logs* podem ser ativados conforme a configuração `verbose` para monitorar o progresso das operações.

Código-fonte 17 – Classe JpaBulkUpdater

```
1  public class JpaBulkUpdater {
2
3      public int run(BenchmarkConfig config) {
4          int total = config.getRecords();
5          int batchSize = config.getBatchSize();
6          int processed = 0;
7          EntityManager em = emf.createEntityManager();
8          em.getTransaction().begin();
9          for (int i = 1; i <= total; i++) {
10             PersonJPA person = em.find(PersonJPA.class, (long) i);
11             if (person != null) {
12                 person.setAge(person.getAge() + 1);
13             }
14             if (i % batchSize == 0) {
15                 em.flush();
16                 em.clear();
17                 em.getTransaction().commit();
18                 processed += batchSize;
19                 if (config.isVerbose()) {
20                     System.out.println("[JPA] Atualizados: " +
    processed);
21                 }
22                 em.getTransaction().begin();
23             }
24         }
25         em.getTransaction().commit();
26         em.close();
27         return total;
28     }
29
30 }
```

O `JpaWorker` (Código-fonte 18) atua como camada de integração para o executor de *benchmark*, encapsulando as operações de inserção, leitura e atualização, de modo que o `ConcurrentBenchmarkExecutor` possa executar os testes de forma concorrente e padronizada. Esse *design* permite comparar facilmente o desempenho entre diferentes estratégias de acesso ao banco, mantendo a separação de responsabilidades e a consistência transacional.

Código-fonte 18 – Trecho da Classe JpaWorker

```
1     @Override
2     public int doInserts(BenchmarkConfig config) throws Exception {
3         if (config.isVerbose())
4             System.out.println("[JPA] Iniciando inserções em massa
    com batchSize=" + config.getBatchSize());
5         return inserter.run(config);
6     }
7
8     @Override
9     public int doReads(BenchmarkConfig config) throws Exception {
10        if (config.isVerbose())
11            System.out.println("[JPA] Iniciando leitura em massa (" +
     config.getRecords() + " registros)...");
12        return reader.run(config);
13    }
14
15    @Override
16    public int doUpdates(BenchmarkConfig config) throws Exception {
17        if (config.isVerbose())
18            System.out.println("[JPA] Iniciando atualização em massa
    ...");
19        return updater.run(config);
20    }
```

Por fim, a execução do *benchmark* é demonstrada na classe `Main` (Código-fonte 19), que inicializa a tabela de teste, define as configurações do *benchmark* e executa as operações concorrentes com múltiplas *threads*. Os resultados são consolidados em um objeto `BenchmarkResult` e exibidos no *console*, permitindo avaliar a eficiência do JPA em cenários de

alta carga.

Código-fonte 19 – Classe Main para o JPA

```
public class Main {

    public static void main(String[] args) throws Exception {
        DbUtils.truncatePersonTable(DbConfig.getDataSource());
        BenchmarkConfig config = new BenchmarkConfig(
                100000,
                1000,
                4,
                false,
                "Teste JPA Multithread",
                true,
                BenchmarkType.MIXED
        );
        JpaWorker worker = new JpaWorker();
        ConcurrentBenchmarkExecutor executor = new
    ConcurrentBenchmarkExecutor(worker);
        var result = executor.run(config);
        System.out.println(result);
    }

}
```

4.1.6 Módulo Spring Data JPA

O módulo Spring Data JPA representa o nível mais alto de abstração entre os quatro avaliados. Essa abordagem elimina grande parte do código repetitivo, utilizando interfaces de repositórios e derivação automática de consultas com base em convenções de nomenclatura. O Spring gerencia as transações, a injeção de dependências e a configuração de forma automática, simplificando a implementação e reduzindo o tempo de desenvolvimento.

O módulo inclui o `PersonRepository` (Código-fonte 20), que é uma interface que estende `JpaRepository` do Spring Data JPA. Essa interface fornece uma abstração completa para operações de persistência, como inserções, consultas, atualizações e deleções, sem a necessidade de implementar manualmente métodos SQL ou JPQL. Ao utilizar o `JpaRepository`, o sis-

tema se beneficia de métodos prontos como o `saveAll`, além da possibilidade de criar consultas personalizadas quando necessário. Esse repositório é fundamental para o `SpringDataBulkInserter`, que realiza as inserções em lote, aproveitando a capacidade do Spring Data JPA de otimizar o envio de múltiplos registros ao banco de dados.

Código-fonte 20 – Interface PersonRepository

```
1 @Repository
2 public interface PersonRepository extends JpaRepository<PersonJPA,
      Long> {
3
4 }
```

O *service* `SpringDataBulkInserter`, apresentada no Código-fonte 21, realiza inserções em massa de registros `PersonJPA`. A estratégia utilizada consiste na criação de lotes de tamanho configurável, armazenando temporariamente os objetos `PersonJPA` em uma lista antes de persisti-los em bloco no repositório `PersonRepository`. Esse procedimento reduz a sobrecarga de comunicação com o banco de dados, pois evita a execução de múltiplas inserções individuais. Além disso, a execução pode ser realizada de forma verbosa, permitindo o acompanhamento do progresso em tempo real. Ao final do processamento, todos os registros são persistidos, garantindo a consistência dos dados.

Código-fonte 21 – Classe SpringDataBulkInserter

```
1  @Service
2  public class SpringDataBulkInserter {
3
4      @Transactional
5      public int run(BenchmarkConfig config) {
6          int total = config.getRecords();
7          int batchSize = config.getBatchSize();
8          int processed = 0;
9          List<PersonJPA> batch = new ArrayList<>(batchSize);
10         for (int i = 1; i <= total; i++) {
11             PersonJPA p = new PersonJPA("Pessoa_" + i, null, 20 + (i
       % 50));
12             batch.add(p);
13             if (i % batchSize == 0) {
```

```
14                  personRepository.saveAll(batch);
15                  batch.clear();
16                  processed += batchSize;
17                  if (config.isVerbose()) {
18                      System.out.println("[Spring Data JPA] Inseridos:
    " + processed);
19                  }
20              }
21          }
22          if (!batch.isEmpty()) {
23              personRepository.saveAll(batch);
24              processed += batch.size();
25          }
26          return processed;
27      }
28
29  }
```

Para a leitura em massa, o componente `SpringDataBulkReader` utiliza o `EntityManager` para buscar registros em lotes de tamanho definido. A leitura é paginada, utilizando os métodos `setFirstResult` e `setMaxResults` para evitar o carregamento de todos os registros na memória de uma só vez, o que é crucial em cenários com grandes volumes de dados. Durante o processamento, o `EntityManager` é limpo periodicamente, garantindo que o consumo de memória permaneça controlado e evitando problemas de `OutOfMemoryError`. O módulo também permite exibir mensagens de progresso para cada quantidade significativa de registros lidos.

O `SpringDataBulkUpdater`, apresentada no Código-fonte 22, realiza atualizações em massa utilizando uma única *query* JPQL, incrementando a idade de todos os registros da tabela `PersonJPA`. Essa abordagem aproveita o poder do banco de dados para realizar a atualização de forma eficiente, evitando a necessidade de carregar individualmente cada registro para memória antes da alteração.

Código-fonte 22 – Classe SpringDataBulkUpdater

```
1  @Component
2  public class SpringDataBulkUpdater {
3
4      @Transactional
```

```
5      public int run(BenchmarkConfig config) {
6          int updated = em.createQuery("UPDATE PersonJPA p SET p.age =
       p.age + 1")
7                  .executeUpdate();
8          if (config.isVerbose()) {
9              System.out.println("[Spring Data JPA] Atualizados: " +
       updated);
10         }
11         return updated;
12     }
13
14 }
```

Todos esses componentes são integrados pelo `SpringDataWorker` (Código-fonte 23), que implementa a interface `DatabaseWorker` e atua como ponto central de execução das operações de *benchmark*. O `SpringDataWorker` recebe as instâncias de inserção, leitura e atualização via injeção de dependência e delega a execução das operações aos componentes correspondentes, mantendo a execução organizada e modular.

Código-fonte 23 – Classe SpringDataWorker

```
1  @Component
2  public class SpringDataWorker implements DatabaseWorker {
3
4      @Override
5      public int doInserts(BenchmarkConfig config) throws Exception {
6          if (config.isVerbose())
7              System.out.println("[Spring Data JPA] Iniciando inserções
       em massa com batchSize=" + config.getBatchSize());
8          return inserter.run(config);
9      }
10
11     @Override
12     public int doReads(BenchmarkConfig config) throws Exception {
13         if (config.isVerbose())
14             System.out.println("[Spring Data JPA] Iniciando leitura
       em massa (" + config.getRecords() + " registros)...");
15         return reader.run(config);
```

```
16      }
17
18      @Override
19      public int doUpdates(BenchmarkConfig config) throws Exception {
20          if (config.isVerbose())
21              System.out.println("[Spring Data JPA] Iniciando atualizaç
    ão em massa...");
22          return updater.run(config);
23      }
24
25  }
```

A execução do módulo é coordenada pela classe `Main` (Código-fonte 24), que serve como ponto de entrada do módulo. Nela, uma instância de `BenchmarkConfig` é criada, permitindo configurar parâmetros como o número total de registros, o tamanho do lote, a quantidade de *threads* e se a execução será verbosa.

Código-fonte 24 – Classe Main para o Spring Data

```
1  @SpringBootApplication
2  public class Main {
3
4      public static void main(String[] args) throws Exception {
5          ApplicationContext context = SpringApplication.run(Main.class
    , args);
6          DbUtils.truncatePersonTable(DbConfig.getDataSource());
7          BenchmarkConfig config = new BenchmarkConfig(
8                  100_000,
9                  1000,
10                 4,
11                 false,
12                 "Teste Spring Data JPA Multithread",
13                 true,
14                 BenchmarkType.MIXED
15         );
16         SpringDataWorker worker = context.getBean(SpringDataWorker.
    class);
17         ConcurrentBenchmarkExecutor executor = new
    ConcurrentBenchmarkExecutor(worker);
```

```
18          var result = executor.run(config);
19          System.out.println(result);
20      }
21
22  }
```

4.2 Aspectos de Funcionais

O projeto foi concebido para executar um conjunto de operações idênticas em todos os módulos desenvolvidos, de modo a assegurar a comparabilidade dos resultados obtidos. As operações realizadas englobaram a inserção de grandes volumes de registros, a execução de atualizações em lote e a realização de consultas parametrizadas. Cada módulo foi submetido aos mesmos conjuntos de dados e executado nas mesmas condições de hardware e banco de dados, garantindo, assim, a equidade dos experimentos.

Para as inserções em massa, foram utilizados mecanismos de processamento em lotes, de forma a reduzir a sobrecarga de comunicação com o banco de dados e otimizar o desempenho. Já as leituras de grandes volumes de dados foram realizadas de forma paginada, com controle do tamanho dos lotes carregados na memória, evitando problemas de consumo excessivo de recursos e garantindo estabilidade durante os testes.

O projeto também contemplou a execução concorrente de operações, simulando cenários realistas de acesso simultâneo ao banco de dados, o que possibilitou avaliar o comportamento de cada módulo sob carga elevada. Todas as medições de desempenho foram realizadas utilizando a biblioteca `System.nanoTime()`, permitindo a captura precisa dos tempos de execução de cada operação e assegurando comparações confiáveis entre os diferentes módulos de persistência.

Além disso, a padronização dos testes garantiu que variáveis externas, como tempo de resposta do banco ou diferenças de configuração do ambiente, tivessem impacto mínimo nos resultados, fortalecendo a validade e a reprodutibilidade das análises realizadas.

4.3 Ferramenta de Avaliação

A ferramenta de avaliação representa o componente central do projeto, responsável por garantir que todos os testes de desempenho fossem executados de forma padronizada, auto-

matizada e controlada. Mais do que um simples conjunto de rotinas de medição, ela foi concebida como um ambiente de experimentação capaz de assegurar a validade, a reprodutibilidade e a comparabilidade dos resultados obtidos entre diferentes tecnologias de persistência.

Sua principal contribuição está na padronização dos experimentos: todas as operações foram realizadas sob as mesmas condições de execução, com os mesmos conjuntos de dados e parâmetros de configuração. Essa abordagem eliminou a influência de fatores externos, permitindo que as diferenças observadas nos resultados refletissem exclusivamente o comportamento de cada *framework* analisado. Assim, a ferramenta atuou como um mediador neutro, garantindo isonomia nas medições e objetividade na comparação entre as abordagens.

Outro aspecto fundamental da ferramenta é a automação do processo de avaliação. Ao permitir a execução sequencial e concorrente de múltiplos testes sem necessidade de intervenção manual, o sistema reduziu significativamente o risco de inconsistências e possibilitou a coleta de dados em larga escala. Essa automação também facilitou a repetição dos testes em diferentes cenários, fortalecendo a confiabilidade das análises estatísticas.

Além de sua função prática, a ferramenta possui um valor metodológico relevante. Ela sistematiza o processo de *benchmark* de modo que o mesmo procedimento possa ser reaplicado em futuras pesquisas, seja para testar novas versões dos *frameworks* já avaliados, seja para incluir novas tecnologias de persistência. Essa característica reforça seu papel como uma solução extensível e reutilizável, contribuindo para o avanço de estudos comparativos na área de desempenho de aplicações Java.

Em síntese, a ferramenta de avaliação consolidou o elo entre a teoria e a prática do projeto, transformando os experimentos isolados em um processo estruturado, mensurável e replicável. Seu desenvolvimento foi essencial para garantir que as conclusões do estudo fossem fundamentadas em dados confiáveis, obtidos a partir de um método rigoroso e consistente.

4.4 Resultados da Pesquisa

Para garantir a confiabilidade dos resultados obtidos nos experimentos de desempenho envolvendo as diferentes estratégias das operações com PostgreSQL, cada cenário de teste foi executado cinco vezes para cada volume de dados analisado, permitindo uma média representativa e reduzindo interferências aleatórias. Após cada ciclo, o serviço do PostgreSQL foi reiniciado manualmente para garantir que não houvesse influência de *cache* interno do banco, como *buffers* compartilhados, páginas já carregadas ou estatísticas otimizadas, enquanto o projeto

Java foi completamente limpo com `mvn clean` e recompilado com `mvn install` antes da nova execução, assegurando que não existissem artefatos compilados ou otimizações residuais da JVM.

Além disso, os testes foram executados utilizando o IntelliJ IDEA, que reinicia automaticamente a JVM a cada execução do projeto, evitando a influência de um JIT Compiler aquecido, otimizações de *runtime* ou resíduos de memória do GC. As médias de consumo de memória foram calculadas a partir dos valores originais em bytes, conforme registrados durante os experimentos, sendo posteriormente convertidas para megabytes (MB) apenas para fins de apresentação e melhor legibilidade dos resultados. Esses procedimentos foram adotados de forma padronizada em todos os testes, com o objetivo de manter o ambiente controlado e garantir que as diferenças observadas fossem decorrentes exclusivamente das técnicas de consulta e dos volumes de dados utilizados.

4.4.1 Leitura em Massa (bulk read)

A primeira bateria de experimentos foi dedicada à avaliação do desempenho em operações de leitura (`BenchmarkType.READ`), utilizando a configuração da classe `BenchmarkConfig` ajustada para processar lotes de 1.000 elementos em diferentes níveis de paralelismo. Para cada cenário, cada experimento foi executado cinco vezes, sendo que os valores apresentados ao longo desta seção correspondem às médias aritméticas obtidas a partir dessas execuções, com o objetivo de reduzir variações pontuais e garantir maior confiabilidade aos resultados. Foram conduzidos cinco cenários distintos, variando a quantidade total de registros lidos (100 mil, 200 mil, 300 mil, 400 mil e 500 mil) e aumentando proporcionalmente o número de *threads* de 1 até 5, de modo a observar a escalabilidade das tecnologias e o impacto no consumo de memória.

Nos testes realizados com Hibernate puro, observou-se uma evolução gradual no desempenho à medida que o volume de leitura e o número de *threads* aumentaram. Considerando os valores médios, para 100 mil registros, o tempo total permaneceu inferior a 1 segundo, com uso médio de 21,1 MB de memória e *throughput* de 136.478 registros por segundo. Com 200 mil registros, o tempo manteve-se abaixo de 1 segundo, consumindo em média 79,7 MB de memória e atingindo 223.890 registros por segundo. Em 300 mil registros, o *throughput* médio aumentou para 298.670 registros por segundo, com uso de 153,6 MB de memória. No teste com 400 mil registros, o desempenho médio subiu para 429.378 registros por segundo, utilizando 68,3 MB de memória. Por fim, com 500 mil registros, o *throughput* médio alcançou 497.611 registros por

segundo, com consumo aproximado de 87,7 MB de memória. Esses resultados indicam que o Hibernate apresenta boa escalabilidade em cenários de leitura concorrente, mantendo tempos de resposta reduzidos.

O JDBC apresentou o melhor desempenho geral nos cenários de leitura, confirmando sua reconhecida eficiência, simplicidade na comunicação com o banco de dados e baixa sobrecarga de abstração. Nos testes com 100 mil registros, o tempo total médio permaneceu inferior a 1 segundo, com 22,1 MB de uso médio de memória e *throughput* de 196.540 registros por segundo. Com 200 mil registros, o *throughput* médio aumentou para 333.020 registros por segundo, utilizando 43,1 MB de memória. No cenário de 300 mil registros, o sistema atingiu média de 490.809 registros por segundo, com consumo de 64,4 MB de memória. Ao testar 400 mil registros, o *throughput* médio cresceu para 625.931 registros por segundo, com aproximadamente 85,9 MB de memória utilizada. Por fim, com 500 mil registros e 5 *threads*, o JDBC obteve o melhor resultado médio de toda a bateria, alcançando 749.397 registros por segundo, com consumo de cerca de 96,2 MB de memória. Esses resultados evidenciam a superioridade do JDBC em cenários de leitura de alto volume, apresentando desempenho previsível e baixa latência, mesmo com aumento progressivo da carga.

Nos testes realizados utilizando JPA, o desempenho apresentou boa consistência e escalabilidade, porém manteve-se abaixo do observado com JDBC e levemente inferior ao Hibernate em cenários com maior paralelismo. No experimento com 100 mil registros, o tempo médio de execução foi inferior a 1 segundo, com 13,2 MB de memória utilizada e *throughput* médio de 119.763 registros por segundo. Com 200 mil registros, o *throughput* médio aumentou para 205.097 registros por segundo, consumindo aproximadamente 26,8 MB de memória. Para 300 mil registros, o desempenho médio subiu para 274.443 registros por segundo, com uso de 26,0 MB de memória. Nos testes com 400 mil registros, o sistema alcançou *throughput* médio de 383.081 registros por segundo, utilizando 39,0 MB de memória. Por fim, com 500 mil registros, o JPA registrou *throughput* médio de 464.027 registros por segundo, com consumo de 59,5 MB de memória. De forma geral, o JPA demonstrou um bom equilíbrio entre desempenho e estabilidade, com crescimento previsível conforme o aumento do paralelismo.

Nos experimentos utilizando Spring Data JPA, observou-se um desempenho consistente, com *throughput* médio superior às demais soluções de mais alto nível, embora acompanhado de maior consumo de memória. Para 100 mil registros, o tempo médio de execução foi inferior a 1 segundo, com 24,3 MB de memória utilizada e *throughput* de 166.944 registros

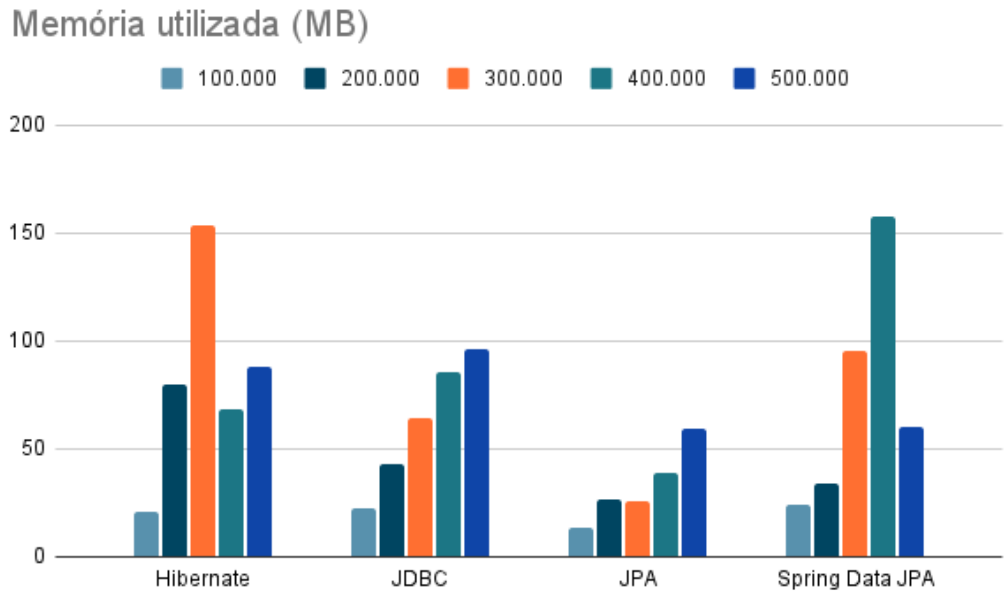
por segundo. Com 200 mil registros, o *throughput* médio aumentou para 306.585 registros por segundo, utilizando 33,9 MB de memória. No cenário de 300 mil registros, o desempenho médio subiu para 450.773 registros por segundo, com 95,3 MB de memória alocada. Ao processar 400 mil registros, o *throughput* médio alcançou 496.427 registros por segundo, consumindo aproximadamente 157,5 MB de memória. Finalmente, com 500 mil registros e uso de 5 *threads*, o Spring Data JPA registrou média de 672.633 registros por segundo, com consumo de cerca de 59,9 MB de memória. Esses resultados indicam que, mesmo adicionando um nível elevado de abstração sobre o JPA, o Spring Data manteve desempenho elevado sob alta concorrência em operações de leitura.

Os valores individuais das cinco execuções referentes ao consumo de memória em cada cenário são apresentados na Tabela 1, possibilitando uma análise da variação entre as execuções. Não foi elaborada uma tabela específica para o tempo total de execução, pois os valores apresentaram variações mínimas entre os testes. Os resultados médios podem ser visualizados nas Figuras 1 e 2. Essa organização facilita a comparação entre os cenários avaliados e direciona a análise quantitativa para os aspectos mais sensíveis às variações experimentais, especialmente o consumo de memória.

Tabela 1 – Consumo de memória (MB) nas operações de leitura para as tecnologias avaliadas

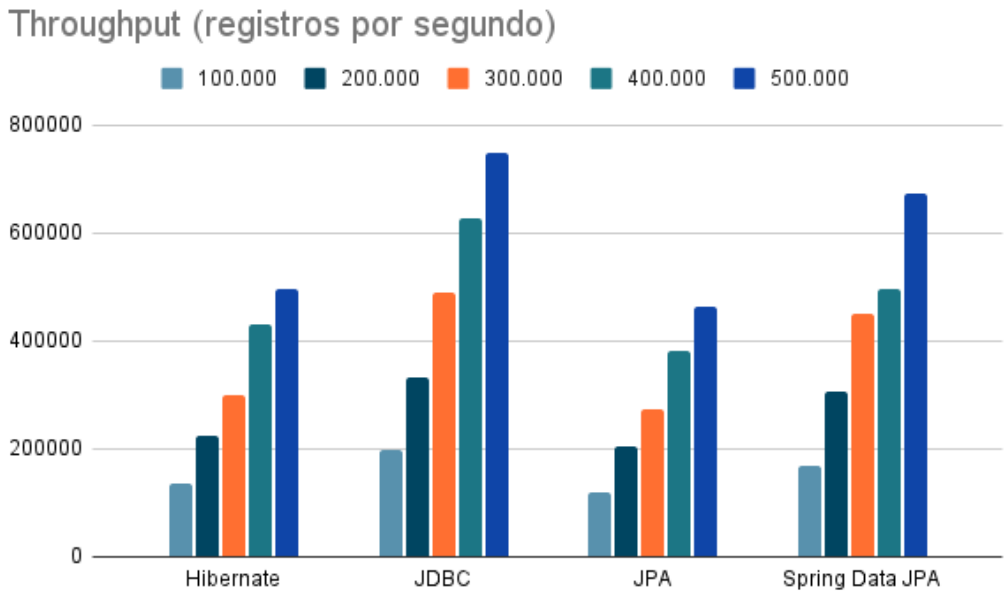
Tecnologia	Registros	Threads	Exec. 1	Exec. 2	Exec. 3	Exec. 4	Exec. 5	Média
Hibernate	100k	1	20,71	20,97	20,88	21,59	21,33	21,11
	200k	2	76,37	75,05	75,64	77,69	75,72	76,11
	300k	3	152,30	150,75	149,71	129,68	148,40	146,57
	400k	4	83,08	27,73	55,40	82,30	79,43	65,20
	500k	5	90,78	39,47	9,79	141,63	137,08	83,35
JDBC	100k	1	21,10	20,78	21,42	21,05	21,27	21,12
	200k	2	41,06	41,09	41,10	41,36	41,20	41,16
	300k	3	61,74	61,38	61,30	61,66	61,30	61,48
	400k	4	81,87	82,21	81,95	81,78	82,09	82,00
	500k	5	102,74	102,72	75,84	102,70	75,21	91,05
JPA	100k	1	12,67	12,72	12,58	12,60	12,62	12,64
	200k	2	15,68	16,03	65,43	15,77	15,24	25,63
	300k	3	19,98	12,36	32,18	25,61	33,88	24,80
	400k	4	51,78	30,28	33,54	35,17	35,45	37,24
	500k	5	61,23	73,87	50,78	43,30	54,94	56,82
Spring Data JPA	100k	1	42,55	18,71	17,56	18,62	18,69	23,24
	200k	2	27,77	51,35	27,80	27,44	27,73	32,42
	300k	3	82,58	102,23	105,75	91,38	82,10	92,81
	400k	4	158,03	145,01	162,05	141,11	145,17	150,27
	500k	5	62,88	57,60	81,06	52,05	32,06	57,13

Figura 1 – Memória Utilizada nas Leituras em Massa



Fonte: Elaborado pela autora.

Figura 2 – Throughtput nas Leituras em Massa



Fonte: Elaborado pela autora.

A partir dos resultados obtidos nos testes de desempenho comparativos envolvendo Hibernate e Spring Data JPA, foi possível observar um comportamento não linear no consumo de memória, especialmente quando analisados os cenários com 300.000, 400.000 e 500.000 operações de leitura concorrente. Embora seja esperado um crescimento progressivo e proporcional do uso de memória conforme o volume de dados aumenta, os experimentos revelaram situações em que o consumo de memória em execuções com 500.000 leituras foi inferior ao registrado

em execuções com cargas menores. Esse fenômeno pode ser compreendido por meio de três fundamentos principais: estratégias internas de otimização adaptativa da JVM, coleta de lixo em ciclos completos, e redução do número de entidades residentes no contexto de persistência.

Primeiramente, é importante considerar que à medida que o volume de requisições aumenta, a JVM aciona mecanismos internos de otimização, incluindo a recompilação adaptativa e o replanejamento de execução, o que pode resultar em maior eficiência computacional e menor consumo efetivo de *heap*, mesmo em cenários com maior carga, documentado pela própria Oracle. Além disso, o menor consumo de memória observado em cenários de maior carga pode estar diretamente relacionado ao comportamento do Garbage Collector, que em situações de intensa pressão de alocação tende a acionar ciclos completos de coleta, removendo objetos temporários e reduzindo o *footprint* da aplicação. Pesquisas demonstram que, sob maiores volumes de operações, o GC pode atuar de maneira mais eficiente do que sob cargas intermediárias devido ao aumento da frequência dos ciclos e à liberação de regiões de memória maiores (Liu *et al.*, 2024). Assim, cargas menores podem gerar resíduos persistentes em *cache*, ao passo que cargas máximas exigem uma reorganização extensiva da *heap*, resultando em menor uso final de memória reportada.

Adicionalmente, o Hibernate e o Spring Data JPA utilizam um mecanismo próprio denominado *First-Level Cache*, associado ao `Persistence Context`, o qual armazena entidades para garantir consistência e rastreamento do estado das instâncias, de acordo com as documentações oficiais do Hibernate e Spring Framework Team. Dessa forma, em cargas moderadas, a retenção de objetos no contexto de persistência tende a ser maior, aumentando temporariamente a ocupação do *heap*. Contudo, quando o volume de requisições e *threads* é elevado, o ciclo de vida dos objetos tende a ser mais curto, reduzindo o número de entidades mantidas simultaneamente na memória e aumentando a frequência de despejo automático desse *cache*.

Portanto, o fato de a execução com 500.000 leituras apresentar menor consumo de memória do que as execuções com 300.000 ou 400.000 não representa anomalia ou erro metodológico. Pelo contrário, corresponde a um comportamento documentado em sistemas baseados na JVM, cujo consumo de memória se ajusta dinamicamente a partir de mecanismos de otimização contínua, reorganização adaptativa e ciclos de coleta intensificados (Putigny *et al.*, 2014). Em síntese, o resultado evidencia que o consumo de memória em aplicações Java baseadas em Hibernate e Spring Data JPA não deve ser interpretado de forma linear, uma vez

que depende tanto da carga de dados quanto da configuração e do momento de atuação dos mecanismos internos de gerenciamento do ambiente de execução.

4.4.2 Escrita em Massa (bulk insert)

A segunda bateria de experimentos avaliou o desempenho em operações de inserção, utilizando a classe `BenchmarkConfig` configurada com o tipo `BenchmarkType.INSERT`. Em todos os cenários, foi realizada a inserção completa do conjunto de registros (100 mil a 500 mil), uma vez que operações de escrita exigem o processamento integral dos dados para garantir a validade dos resultados. Os testes foram executados em lotes de 1.000 elementos (`batchSize`), variando-se o número de *threads* proporcionalmente ao volume de dados.

Nos testes com o Hibernate, o tempo total médio de execução para 100 mil registros foi de 11 segundos, com consumo médio de 20,0 MB de memória e *throughput* de 8.821 registros por segundo. Para 200 mil registros, o tempo médio registrado foi de 12,6 segundos, com consumo de aproximadamente 57,8 MB de memória, resultando em 15.236 registros por segundo. Com 300 mil registros, obteve-se um tempo médio de 12,2 segundos, consumo de 98,0 MB de memória e *throughput* de 23.690 registros por segundo. No cenário com 400 mil registros, o processamento apresentou tempo médio de 12,6 segundos, com 64,3 MB de memória e 30.769 registros por segundo. Por fim, com 500 mil registros, o tempo médio foi de 13,8 segundos, o consumo de memória ficou em torno de 16,1 MB e o *throughput* alcançou 34.463 registros por segundo. Observa-se um comportamento de incremento contínuo no *throughput* conforme o volume de dados cresce, indicando maior eficiência do Hibernate em cenários de maior paralelismo e carga.

O JDBC demonstrou desempenho substancialmente superior nos testes de inserção, apresentando tempos médios reduzidos e *throughput* elevado mesmo com o crescimento da carga. Para 100 mil registros, o tempo médio total foi de aproximadamente 1 segundo, com consumo médio de 0,88 MB de memória e 72.425 registros por segundo. Com 200 mil registros, o tempo manteve-se em torno de 1 segundo, utilizando cerca de 15,99 MB de memória e alcançando 126.151 registros por segundo. No cenário de 300 mil registros, o tempo médio variou entre 1 e 2 segundos, com consumo de 6,18 MB de memória e *throughput* de 155.693 registros por segundo. Para 400 mil registros, o tempo médio foi de 2 segundos, com 17,33 MB de memória e 186.362 registros por segundo. Por fim, com 500 mil registros, o tempo manteve-se em torno de 2 segundos, apresentando consumo médio de 16,67 MB de memória e *throughput* de

213.336 registros por segundo. Esses resultados reforçam a eficiência do JDBC, que se destacou pela excelente escalabilidade e estabilidade, mantendo tempos reduzidos mesmo sob volumes elevados de inserção.

O JPA apresentou resultados consistentes ao longo dos experimentos, embora com desempenho inferior ao observado com o JDBC. Para 100 mil registros, o tempo médio total foi de aproximadamente 12 segundos, com consumo médio de 5,45 MB de memória e *throughput* de 8.500 registros por segundo. Com 200 mil registros, o tempo manteve-se próximo de 12 segundos, utilizando cerca de 6,12 MB de memória e atingindo 15.759 registros por segundo. Nos testes com 300 mil registros, o tempo médio variou entre 12 e 13 segundos, com consumo de 7,25 MB de memória e *throughput* de 23.098 registros por segundo. Para 400 mil registros, o tempo médio foi de aproximadamente 13,4 segundos, com consumo de 49,76 MB de memória e *throughput* de 28.711 registros por segundo. Por fim, com 500 mil registros, o tempo oscilou entre 13 e 14 segundos, com consumo médio de 62,11 MB de memória e *throughput* de 34.811 registros por segundo. Os resultados indicam ganho progressivo de desempenho conforme o aumento do volume de dados, ainda que acompanhado de maior variabilidade no consumo de memória.

Nos experimentos com Spring Data JPA, o desempenho médio manteve-se próximo ao observado com o JPA tradicional, porém com consumo significativamente mais elevado de memória e resultados ligeiramente superiores em *throughput*. Para 100 mil registros, o tempo médio variou entre 12 e 13 segundos, com consumo médio de 89,22 MB de memória e *throughput* de 7.809 registros por segundo. Em 200 mil registros, o tempo manteve-se no mesmo intervalo, utilizando cerca de 154,42 MB de memória e atingindo 15.228 registros por segundo. Para 300 mil registros, o tempo novamente oscilou entre 12 e 13 segundos, com consumo médio de 235,99 MB de memória e *throughput* de 23.788 registros por segundo. Nos testes com 400 mil registros, o tempo médio aumentou levemente para o intervalo de 13 a 14 segundos, com consumo de 273,02 MB de memória e *throughput* de 29.236 registros por segundo. Por fim, com 500 mil registros, o tempo médio foi de aproximadamente 14 segundos, com consumo expressivo de 346,29 MB de memória e *throughput* de 33.943 registros por segundo. Apesar do alto consumo de memória, o Spring Data JPA apresentou comportamento estável e consistente, com leve superioridade em *throughput* em relação ao JPA tradicional.

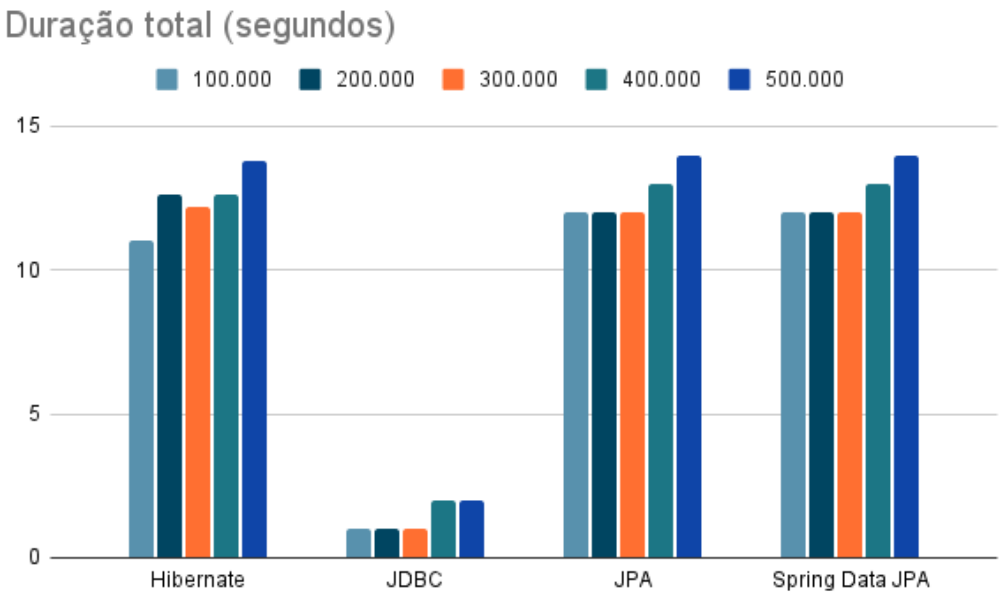
Os valores individuais das execuções, referentes especificamente ao consumo de memória, são apresentados de forma detalhada na Tabela 2. A visualização dos resultados médios

pode ser observada nas Figuras 3, 4 e 5, que sintetizam, respectivamente, a duração total das execuções, o consumo médio de memória e o *throughput* obtidos nos experimentos de inserção.

Tabela 2 – Consumo de memória (MB) nas operações de inserção para as tecnologias avaliadas

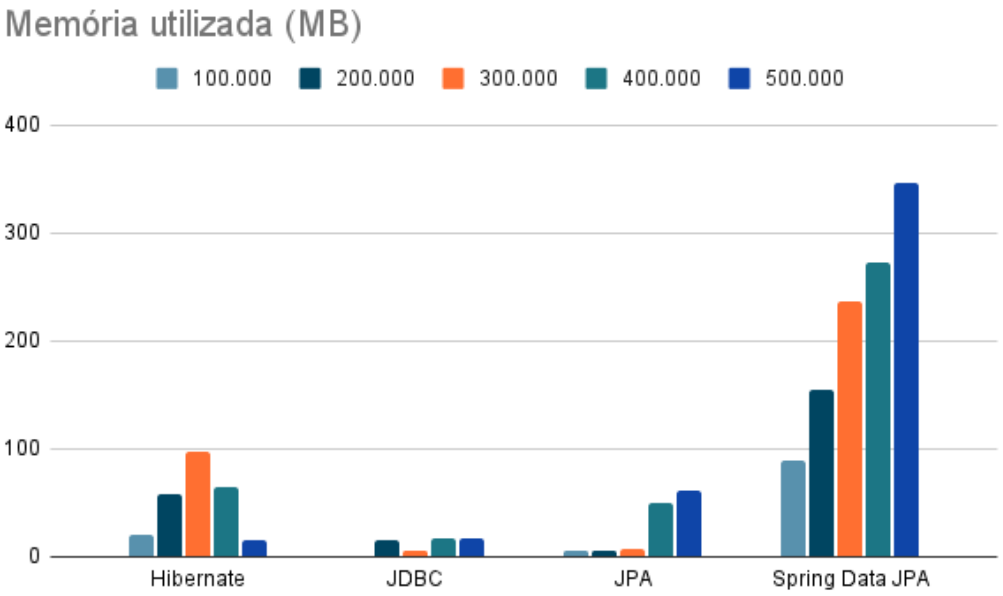
Tecnologia	Registros	Threads	Exec. 1	Exec. 2	Exec. 3	Exec. 4	Exec. 5	Média
Hibernate	100k	1	22,36	21,03	19,63	16,24	16,24	19,10
	200k	2	17,50	124,46	8,12	115,59	10,37	55,24
	300k	3	147,63	16,82	20,41	159,87	123,06	93,56
	400k	4	10,86	14,57	7,18	146,24	127,86	61,34
	500k	5	14,84	17,08	3,97	17,96	23,25	15,42
JDBC	100k	1	0,81	0,79	0,82	0,76	1,06	0,85
	200k	2	15,24	15,29	15,32	15,30	15,12	15,25
	300k	3	6,17	5,79	5,80	5,48	6,24	5,90
	400k	4	16,68	16,15	16,73	16,42	16,67	16,53
	500k	5	14,79	19,16	15,16	15,48	14,91	15,90
JPA	100k	1	2,52	5,20	7,91	8,02	2,33	5,19
	200k	2	2,62	7,11	3,01	6,44	10,05	5,85
	300k	3	10,77	8,62	5,33	0,12	9,83	6,93
	400k	4	1,54	78,23	3,43	83,62	70,58	47,88
	500k	5	11,50	87,63	58,45	49,46	89,18	59,24
Spring Data JPA	100k	1	97,68	114,00	52,20	114,46	47,18	85,11
	200k	2	128,78	210,30	185,95	119,26	92,68	147,99
	300k	3	231,87	202,17	204,06	224,25	263,23	225,12
	400k	4	316,73	329,62	185,33	233,57	236,66	260,38
	500k	5	345,51	229,59	397,77	323,33	355,45	330,33

Figura 3 – Duração Total nas Inserções em Massa



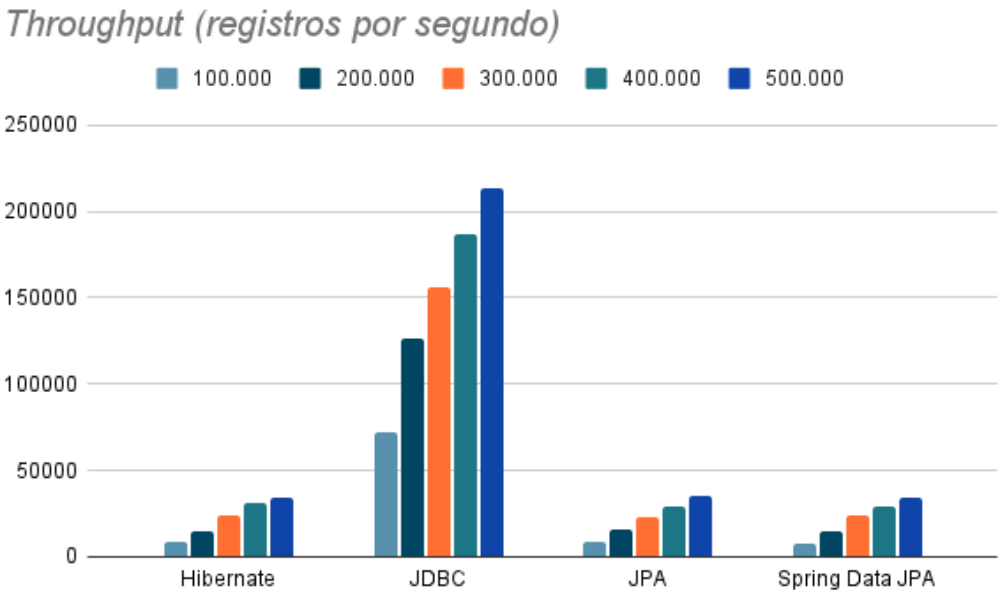
Fonte: Elaborado pela autora.

Figura 4 – Memória Utilizada nas Inserções em Massa



Fonte: Elaborado pela autora.

Figura 5 – Throughput nas Inserções em Massa



Fonte: Elaborado pela autora.

Durante a execução dos experimentos de inserção em larga escala com o Hibernate, observou-se a formação de um padrão triangular no gráfico de consumo de memória, caracterizado por picos de crescimento seguidos de quedas abruptas. Esse comportamento, frequentemente chamado de *sawtooth pattern*, é amplamente documentado na literatura de máquinas virtuais Java e está diretamente relacionado ao funcionamento do GC da JVM e à maneira como o Hibernate gerencia seus objetos internos durante operações intensivas de escrita.

Do ponto de vista da JVM, o padrão triangular é uma consequência direta da estratégia de coleta de lixo adotada pelos coletores modernos utilizados nas versões recentes do Java, especialmente o *Garbage First Garbage Collector* (G1GC), que é o coletor padrão desde o Java 9, além de alternativas de baixa latência como o *Z Garbage Collector* (ZGC) e o Shenandoah. Esses coletores trabalham de forma predominantemente incremental e paralela, realizando ciclos frequentes de alocação no Eden Space seguidos de limpezas periódicas. Durante as operações de inserção, o Hibernate gera uma grande quantidade de objetos temporários, incluindo entidades, *snapshots* e metadados internos, o que leva a um aumento gradual da ocupação do Eden. Quando esse espaço atinge um limiar pré-estabelecido, o coletor executa uma Minor GC, removendo a maioria dos objetos de curta duração. Esse comportamento produz visualmente uma subida contínua de memória seguida por uma queda abrupta, o padrão em forma de "triângulo" característico. Conforme descrito por Goetz *et al.* (2021), essa dinâmica é típica em aplicações que executam alocações intensivas e rápidas, especialmente em cenários com *frameworks* ORM que geram grande volume de objetos efêmeros.

Além disso, o próprio Hibernate contribui diretamente para essa oscilação ao utilizar o Persistence Context (*1st-level cache*) como repositório temporário das entidades gerenciadas durante o ciclo de inserção. Em operações em lote, o Hibernate acumula entidades até alcançar o tamanho do lote, realizando então o envio dos dados para o JDBC e limpando o contexto interno. Esse processo cria um segundo padrão cíclico dentro do padrão geral do GC: o contexto enche, produzindo um crescimento progressivo da memória, e esvazia, liberando dezenas de milhares de objetos. O livro *Java Persistence with Hibernate* Bauer & King (2016), descreve esse comportamento como um dos principais responsáveis por picos e vales regulares em aplicações ORM de alto volume.

Outro fator que reforça o padrão de serra é o comportamento adaptativo da JVM. Durante a execução de cargas maiores, a máquina virtual ajusta dinamicamente o tamanho dos espaços de memória (Eden, Survivor e Old Generation), conforme previsto pelo algoritmo de ergonomia da JVM. Assim, à medida que o volume de dados aumenta, o GC recalibra a distribuição de memória, tornando seus ciclos mais eficientes. Esse mecanismo explica, inclusive, por que o *throughput* cresce proporcionalmente ao volume de registros inseridos: com mais dados e mais ciclos de execução, o JIT Compiler tende a otimizar trechos críticos do código, e o GC ajusta a memória para maior estabilidade operacional, reduzindo perdas e aumentando a eficiência do processamento.

Em síntese, o formato triangular observado nos gráficos de consumo de memória durante as execuções do Hibernate não representa um comportamento anômalo, mas sim o resultado natural das interações entre o coletor de lixo da JVM, o modelo de gerenciamento de entidades do Hibernate e os princípios de otimização dinâmica da própria máquina virtual. Tais interações são amplamente conhecidas e são típicas de sistemas que processam grandes volumes de dados utilizando *frameworks* ORM sobre a JVM.

4.4.3 Operações Transacionais Mistas

Abrangendo o caso de teste com operações transacionais mistas, mantiveram-se os mesmos parâmetros de execução: 500.000 registros, lotes de 1.000 elementos, 5 *threads* simultâneas, sem rodada de aquecimento e com saída detalhada. O tipo de *benchmark* selecionado foi o misto (`BenchmarkType.MIXED`), englobando inserções, leituras e atualizações.

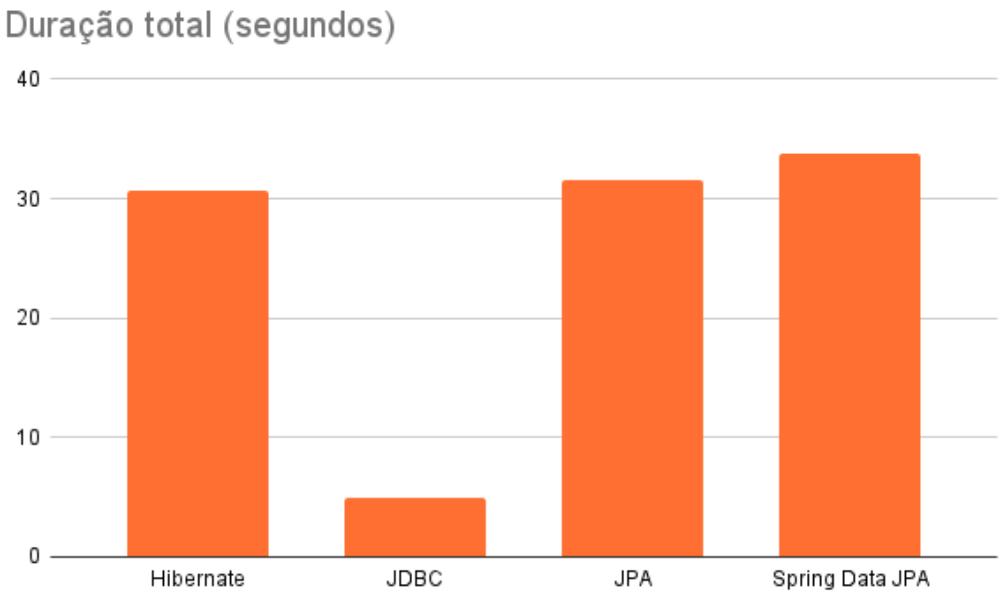
Nos testes com Hibernate, observou-se uma duração média de 30,6 segundos, acompanhada de consumo médio de 116,5 MB de memória e *throughput* aproximado de 48.617 registros por segundo, evidenciando um impacto da sobrecarga de gerenciamento de contexto em cenários transacionais combinados.

Em contraste, o teste com JDBC puro demonstrou desempenho amplamente superior, apresentando uma duração média de 5 segundos, memória média utilizada de 78,36 MB e *throughput* de aproximadamente 278.729 registros/segundo. Já o teste com JPA (Hibernate como *provider*) obteve desempenho semelhante ao Hibernate, apresentando duração média de 31,6 segundos, consumo médio de 61,75 MB e *throughput* médio de 47.136 registros/segundo.

Por fim, o teste com Spring Data JPA apresentou um comportamento distinto, em função do volume maior de atualizações no cenário misto: duração média de 33,8 segundos, uso de memória significativamente superior (307,6 MB) e *throughput* de 102.404 registros/segundo, evidenciando maior *overhead* interno, porém com maior velocidade relativa nas operações de atualização.

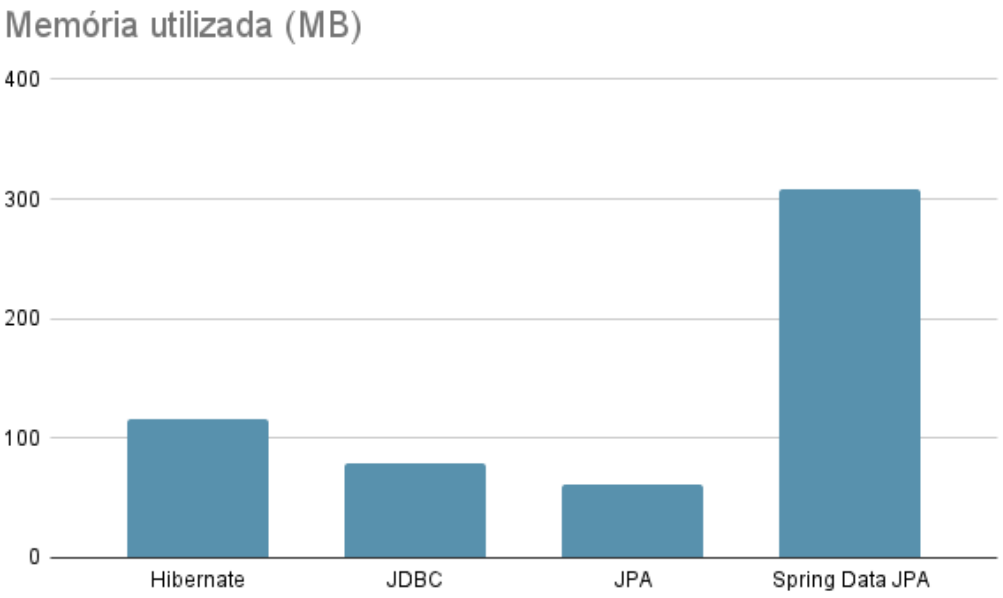
As Figuras 6, 7 e 8 apresentam a representação gráfica das métricas de duração total, consumo de memória e *throughput* obtidas nos experimentos com operações transacionais mistas. O propósito dessas figuras é exclusivamente facilitar a visualização dos dados quantitativos já discutidos, servindo como apoio à análise apresentada, sem agregar interpretações adicionais aos resultados.

Figura 6 – Duração Total nas Operações Transacionais Mistas



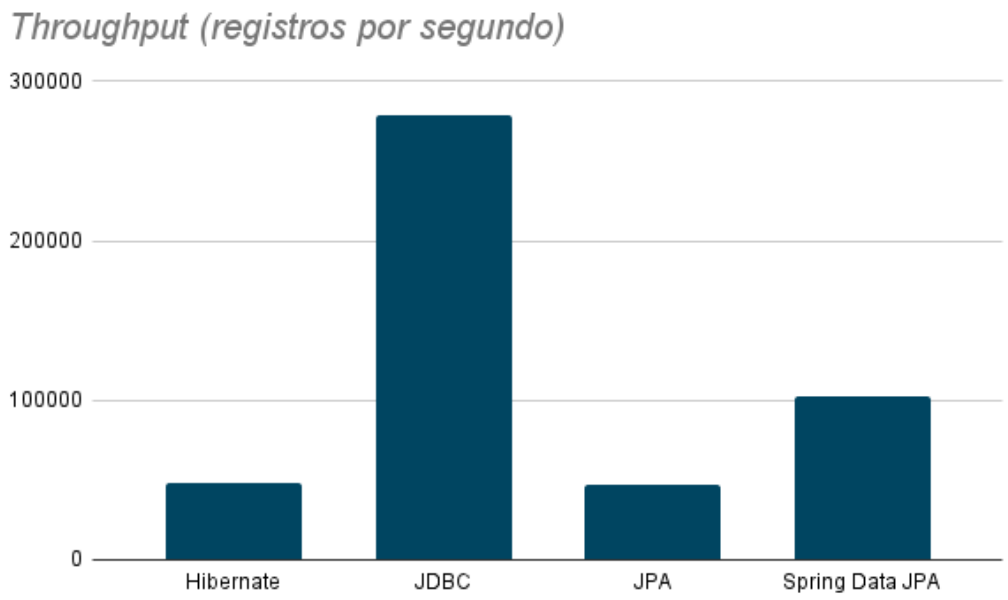
Fonte: Elaborado pela autora.

Figura 7 – Memória Utilizada nas Operações Transacionais Mistas



Fonte: Elaborado pela autora.

Figura 8 – Throughput nas Operações Transacionais Mistas



Fonte: Elaborado pela autora.

5 CONCLUSÃO

Este trabalho teve como propósito analisar comparativamente quatro tecnologias amplamente utilizadas no ecossistema Java para manipulação de dados — JDBC, Hibernate, JPA e Spring Data JPA — considerando seu desempenho, consumo de recursos, complexidade de implementação e adequação a diferentes cenários de uso. A estrutura do trabalho foi organizada de modo a garantir que todos os objetivos definidos no Capítulo 1 fossem sistematicamente desenvolvidos e plenamente atendidos.

Esses objetivos começaram a ser atendidos por meio do levantamento teórico detalhado das tecnologias JDBC, Hibernate, JPA e Spring Data JPA, bem como de outras alternativas mencionadas na literatura. Dentro do Capítulo 2, foram analisados seus princípios, funcionamento interno, limitações e vantagens, cobrindo integralmente o primeiro objetivo específico. No **Capítulo 4**, todos os experimentos planejados na Metodologia, foram realizados e analisados, cumprindo totalmente o segundo e o terceiro objetivos específicos.

Os resultados permitiram comparar, com base empírica, o comportamento das tecnologias estudadas. Observou-se que o JDBC obteve o melhor desempenho bruto, especialmente em inserções de grande volume, apresentando menor consumo de memória e maior *throughput*. Por sua vez, Hibernate, JPA e Spring Data JPA, embora mais lentos e com maior uso de recursos, destacaram-se pela produtividade, facilidade de manutenção e significativa redução de código, especialmente no caso do Spring Data. Assim, verificou-se que diferentes tecnologias se adequam melhor a diferentes contextos, reforçando a importância de alinhamento entre requisitos e escolha arquitetural.

Dessa forma, conclui-se que todos os objetivos propostos na introdução foram integralmente atendidos: a investigação teórica foi realizada, os experimentos foram conduzidos com rigor metodológico e os resultados permitiram uma análise comparativa completa, possibilitando responder à pergunta de pesquisa. Durante o desenvolvimento desta pesquisa, surgiram diversas oportunidades de ampliação do estudo que extrapolam o escopo definido. Assim, como trabalhos futuros, propõe-se:

a) Expandir o conjunto de tecnologias avaliadas, incluindo MyBatis, Apache OpenJPA, *Java Object Oriented Querying* (jOOQ), EclipseLink e soluções reativas como *Reactive Relational Database Connectivity* (R2DBC);
b) Investigar o impacto de diferentes Garbage Collectors da JVM (G1GC, ZGC, Shenandoah, Parallel GC) sobre o desempenho das tecnologias, especialmente em cargas de trabalho

intensivas;

c) Avaliar cenários de alta concorrência, com múltiplas *threads*, *pools* de conexões otimizados e contenção de *locks* em operações simultâneas;
d) Executar testes em ambientes distribuídos, incluindo microsserviços, *containers* Docker e orquestração em Kubernetes, para avaliar impactos de latência, rede e escalabilidade horizontal;
e) Expandir as métricas utilizadas, incluindo latência percentil (p95, p99), aquecimento da JVM, impacto energético e *overhead* de serialização/deserialização;
f) Testar bancos de dados NoSQL, ampliando a análise para persistência de documentos, grafos e chave-valor em aplicações Java modernas;
g) Investigar otimizações avançadas, como *tuning* de Hibernate, uso de cache de segundo nível, *flush* diferido no JPA e configuração avançada do PostgreSQL.

REFERÊNCIAS

BARAN, J.; MURYJAS, P. The analysis of java orm frameworks performance in terms of analytical data processing. **Journal of Computer Sciences Institute**, Lublin University of Technology, Poland, v. 27, n. 1, p. 178–185, may 2023. ISSN 2544-0764. Disponível em: <https://doi.org/10.35784/jcsi.3632>.

BAUER, C.; KING, G. **Java Persistence with Hibernate**. 2. ed. Shelter Island, NY: Manning Publications, 2016. ISBN 978-1617290459. Disponível em: <https://www.manning.com/books/java-persistence-with-hibernate>.

BONTEANU, A.-M.; TUDOSE, C.; ANGHEL, A. M. Multi-platform performance analysis for crud operations in relational databases from java programs using spring data jpa. In: **2023 13th International Symposium on Advanced Topics in Electrical Engineering (ATEE)**. Bucharest, Romania: IEEE, 2023. p. 1–6. ISBN 979-8-3503-3193-6. Disponível em: <https://doi.org/10.1109/ATEE58038.2023.10108212>.

CABAN, K.; CZUCHRYTA, P.; PAŃCZYK, B. Performance analysis of working with relational and non-relational databases in java applications. **Journal of Computer Sciences Institute**, Lublin University of Technology, Poland, v. 33, n. 1, p. 298–305, dec. 2024. ISSN 2544-0764. Disponível em: <https://doi.org/10.35784/jcsi.6332>.

DATANUCLEUS. **DataNucleus AccessPlatform 6.0 Release Notes**. 2021. Disponível em: <https://www.datanucleus.org/documentation/news/access_platform_6_0.html>. Acesso em: 29 out. 2025.

DevMedia. **Java JPA: A evolução da persistência em Java**. 2014. Disponível em: <https://www.devmedia.com.br/java-jpa-a-evolucao-da-persistencia-em-java/29694>. Acesso em: 20 jan. 2026.

FLICK, U. **An Introduction to Qualitative Research**. 7th. ed. London: SAGE Publications Ltd, 2022. 632 p. ISBN 978-1-5297-8132-8. Disponível em: <https://us.sagepub.com/en-gb/eur/an-introduction-to-qualitative-research/book278983>.

FRUTH, M.; SCHERZINGER, S.; MAUERER, W.; RAMSAUER, R. Tell-tale tail latencies: Pitfalls and perils in database benchmarking. In: **Performance Evaluation and Benchmarking: 13th TPC Technology Conference, TPCTC 2021, Copenhagen, Denmark, August 20, 2021, Revised Selected Papers**. Berlin: Springer-Verlag, 2021. p. 119–134. ISBN 978-3-030-94436-0. Disponível em: <https://doi.org/10.1007/978-3-030-94437-7_8>.

GARCIA, V. C. **Plataformas de Processamento de Dados em Tempo Real: Desenvolvendo Arquiteturas Eficientes**. 2023. Disponível em: <https://viniciusgarcia.me/architecture/plataformas-de-processamento-de-dados-em-tempo-real/>. Acesso em: 29 out. 2025.

GEORGE, J. **Optimizing Database Performance in Java and AngularJS Web Applications: Caching and ORM Strategies**. 2021. Disponível em: <https://www.researchgate.net/publication/389710439_Optimizing_Database_Performance_in_Java_and_AngularJS_Web_Applications_Caching_and_ORM_Strategies>. Acesso em: 29 out. 2025.

GESSERT, F.; WINGERATH, W.; RITTER, N. Caching in research and industry. In: GESSERT, F.; WINGERATH, W.; RITTER, N. (Ed.). **Fast and Scalable Cloud Data Management**. Cham: Springer Cham, 2020. p. 85–130. ISBN 978-3-030-43506-6. Disponível em: <https://link.springer.com/book/10.1007/978-3-030-43506-6>.

GOETZ, B.; PEIERLS, T.; BLOCH, J.; BOWBEER, J.; HOLMES, D.; LEA, D. **Java Concurrency in Practice**. 2nd. ed. Boston: Addison-Wesley Professional, 2021. ISBN 978-0321349606.

HALILI, F.; NUHIJI, A.; VELIU, D. M. Polyglot persistence in microservices: Managing data diversity in distributed systems. **arXiv preprint arXiv:2509.08014**, 2025. Acesso em: 20 jan. 2026. Disponível em: <https://arxiv.org/abs/2509.08014>.

JETBRAINS. **Software Developers Statistics 2024 - State of Developer Ecosystem Report**. 2024. Disponível em: <https://www.jetbrains.com/lp/devecosystem-2024/>. Acesso em: 8 out. 2025.

JOVANOV, F.; ZDRAVESKI, V.; GUSEV, M.; KOSTOSKA, M. Optimization and parallelization of object-relational mappers. In: **2023 46th MIPRO ICT and Electronics Convention (MIPRO)**. Opatija, Croatia: IEEE, 2023. p. 1678–1683. ISBN 978-953-233-104-2. Disponível em: <https://doi.org/10.23919/MIPRO57284.2023.10159866>.

KOMANOV, D. **Benchmarking batch JDBC queries**. 2021. Disponível em: <https://dkomanov.medium.com/benchmarking-batch-jdbc-queries-a2b5911ada26>. Acesso em: 29 out. 2025.

LIU, W.; MONDAL, S.; CHEN, T.-H. P. An empirical study on the characteristics of database access bugs in java applications. **ACM Transactions on Software Engineering and Methodology**, Association for Computing Machinery, New York, v. 33, n. 7, p. 1–25, sep. 2024. ISSN 1049-331X. Disponível em: <https://doi.org/10.1145/3672449>.

LUXOFT. **Object-relational Mapping Using JPA, Hibernate and Spring Data JPA. Comparing the performance of persisting entities**. 2022. Disponível em: <https://luxsocialmedia.medium.com/object-relational-mapping-using-jpa-hibernate-and-spring-data-jpa-eb4610f086d0>. Acesso em: 19 nov. 2025.

MIHALCEA, V. **High-Performance Java Persistence**. 1. ed. Seattle, WA, USA: Amazon Digital Services LLC - KDP Print US, 2016, 2016. ISBN 978-9730228236. Disponível em: <https://www.amazon.com/High-Performance-Java-Persistence-Vlad-Mihalcea/dp/973022823X>.

MORUSU, V. **Performance Evaluation of ODBC and JDBC Connections in Big Data Systems**. 2025. Disponível em: <https://medium.com/@morusu.v/testing-publications-10454125446c>. Acesso em: 29 out. 2025.

POŁEĆ, M.; PITERA, J.; KOZIEŁ, G. Comparing the performance of the object-relational mapping program-ming frameworks available in java. **Journal of Computer Sciences Institute**, Lublin University of Technology, Poland, v. 22, p. 59–65, mar. 2022. ISSN 2544-0764. Disponível em: <https://ph.pollub.pl/index.php/jcsi/article/view/2810>.

PUTIGNY, B.; GOGLIN, B.; BARTHOU, D. A benchmark-based performance model for memory-bound hpc applications. In: **2014 International Conference on High Performance Computing & Simulation (HPCS)**. Bologna: IEEE, 2014. p. 943–950. Disponível em: <https://doi.org/10.1109/HPCSim.2014.6903790>.

QU, X. Application of java technology in dynamic web database technology. **Journal of Physics: Conference Series**, IOP Publishing, v. 1744, n. 4, p. 042029, feb. 2021. ISSN 1742-6596. Disponível em: <https://doi.org/10.1088/1742-6596/1744/4/042029>.

RAJ, A. **JDBC vs JPA: Why Performance-Conscious Companies Are Making the Switch**. 2025. Disponível em: <https://medium.com/@ashay11raj/jdbc-vs-jpa-why-performance-conscious-companies-are-making-the-switch-0f242ab01749>. Acesso em: 29 out. 2025.

RICHARDS, M.; FORD, N. **Fundamentals of Software Architecture: An Engineering Approach**. 1. ed. [S.l.]: O'Reilly Media, 2020. ISBN 978-1492043454.

SHARMA, M. **Java & Databases: The Undying Power of JDBC in 2025 and Beyond!** 2025. Disponível em: <https://medium.com/%40sharmamadhusudans337/java-databases-the-undying-power-of-jdbc-in-2025-and-beyond-7ac871fe89da>. Acesso em: 29 out. 2025.

SOUZA, M. B. d. **Spring Data JPA - Persistência Simples e Eficaz**. 1. ed. São Paulo, Brazil: Editora Ciência Moderna, 2018. ISBN 978-85-399-0944-5. Disponível em: <https://www.amazon.com.br/Spring-Data-Persist%C3%AAncia-Simples-Eficaz/dp/8539909448>.

TIOBE. **Tiobe Index**. 2025. Disponível em: <https://www.tiobe.com/tiobe-index>. Acesso em: 8 out. 2025.

TORKAR, R.; FURIA, C. A.; FELDT, R.; NETO, F. Gomes de O.; GREN, L.; LENBERG, P.; ERNST, N. A. A method to assess and argue for practical significance in software engineering. **IEEE Transactions on Software Engineering**, IEEE, v. 48, n. 6, p. 2053–2065, 2022. Disponível em: <https://doi.org/10.1109/TSE.2020.3048991>.

TRAINI, L.; CORTELLESSA, V.; POMPEO, D. D.; TUCCI, M. Towards effective assessment of steady state performance in java software: are we there yet? **Empirical Software Engineering**, Springer, v. 28, n. 13, p. 1–57, nov. 2022. ISSN 1573-7616. Disponível em: <https://doi.org/10.1007/s10664-022-10247-x>.

XU, Z.; LEI, M. An overview of data persistence approaches for enterprise web applications. **ICCK Transactions on Computer Science**, IECE, Spokane, v. 2, n. 1, p. 10–17, dec. 2025. Disponível em: <https://doi.org/10.62762/TCS.2024.529749>.

YIN, R. K. **Case Study Research and Applications: Design and Methods**. 6th. ed. Thousand Oaks: SAGE Publications, Inc., 2018. 352 p. ISBN 9781506336169. Disponível em: <https://uk.sagepub.com/en-gb/eur/case-study-research-and-applications/book250150>.

ŻUCHNIK, M.; KOPNIAK, P. Comparative analysis of connection performance with databases via jdbc interface and orm programming frameworks. **Journal of Computer Sciences Institute**, Lublin University of Technology, Lublin, v. 21, p. 309–315, set. 2021. ISSN 2544-0764. Disponível em: <https://doi.org/10.35784/jcsi.2729>.