UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS DE QUIXADÁ

CURSO DE GRADUAÇÃO EM ENGENHARIA DE SOFTWARE

GUSTAVO CLEMENTE COLOMBO DA ROCHA

UMA ANÁLISE COMPARATIVA ENTRE FRAMEWORKS DE PERSISTÊNCIA DE DADOS EM JAVASCRIPT

QUIXADÁ

2022

GUSTAVO CLEMENTE COLOMBO DA ROCHA

UMA ANÁLISE COMPARATIVA ENTRE FRAMEWORKS DE PERSISTÊNCIA DE DADOS EM JAVASCRIPT

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Software do Campus de Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia de Software.

Orientador: Prof. Dr. Regis Pires Magalhães

QUIXADÁ

2022

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

R573a Rocha, Gustavo Clemente Colombo da.
Uma análise comparativa entre frameworks de persistência de dados em JavaScript / Gustavo Clemente Colombo da Rocha. – 2022.
60 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Quixadá, Curso de Engenharia de Software, Quixadá, 2022.
Orientação: Prof. Dr. Regis Pires Magalhães.

1. Framework (Arquivo de computador). 2. Banco de dados. 3. Persistência (Ciência da computação). 4. Pesquisa quantitativa. 5. Pesquisa qualitativa. I. Título.

CDD 005.1

GUSTAVO CLEMENTE COLOMBO DA ROCHA

UMA ANÁLISE COMPARATIVA ENTRE FRAMEWORKS DE PERSISTÊNCIA DE DADOS EM JAVASCRIPT

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Software do Campus de Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia de Software.

Aprovada em: ____/ ____/ ____.

BANCA EXAMINADORA

Prof. Dr. Regis Pires Magalhães (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. Emanuel Ferreira Coutinho
Universidade Federal do Ceará (UFC)

Prof. Dr. Victor Aguiar Evangelista de Farias
Universidade Federal do Ceará (UFC)

A todos os meus amigos e familiares que me apoiaram e estiveram comigo ao longo de todos esses anos. Pai, por todos esses anos me incentivar e me fazer persistir nos meus objetivos. Mãe, por sempre iluminar e proteger meus caminhos e decisões.

AGRADECIMENTOS

Inicialmente, agradeço aos meus pais, Sérgio Luiz da Rocha e Adiles Colombo por sempre me incentivarem e darem forças para que eu pudesse dar o melhor de mim e continuar ao longo desses anos buscando e conseguido conquistas, mesmo que muitas vezes não estivéssemos próximos, não teria conseguido sem a força que vem de vocês.

Meus sinceros agradecimentos a todas as minhas tias, especialmente a Édina Rocha, Izabel Rocha, Marli Colombo Horn, aos meus tios, Pedro Paulo Rocha, Gilson Rocha, Helberto Alfredo Horn e demais amigos que distantes de mim geograficamente ao longo desses 4 anos, nunca deixaram de me apoiar e me incentivar para que eu conseguisse alcançar meus objetivos acadêmicos e profissionais.

Agradeço aos meus mais leais amigos da PTF, Eric Rodrigues, Marcos Gênesis, Ítalo Lima, Jeferson Gonçalves, Cristiano Junior, Guilherme Rodrigues, Josué Nicholson, Rafael Lima, Victor Lucas, João Teixeira, Aurélio Henrique, Lucas Nascimento, Fabrício Nogueira, Murilo Souza e Lucas Lima que ao longo da minha graduação me incentivaram não somente a me tornar um ser humano melhor, mas também um melhor profissional e que me acompanharam em todos os momentos nos últimos 4 anos, sejam eles bons ou de dificuldade.

Agradeço ao Prof. Dr. Regis Pires Magalhães por ter me aceitado como orientando no Trabalho de Conclusão de Curso (TCC) e me guiado ao longo de dois semestres visando produzir um trabalho que servisse a comunidade acadêmica e industrial de engenharia de software.

Agradeço aos professores Prof. Dr. Emanuel Ferreira Coutinho e Prof. Dr. Victor Aguiar Evangelista de Farias por aceitarem participar da banca avaliadora deste trabalho para que ao final da escrita desse trabalho ele contribuísse com a comunidade de desenvolvedores e a Engenharia de Software.

Agradeço a todos os servidores e professores da UFC campus Quixadá, que dia após dia me inspiraram a continuar na minha jornada pela vontade e paixão transmitida em suas tarefas e orientações.

Também gostaria de agradecer aos meus amigos que fiz ao longo da graduação, sem vocês a minha trajetória não teria sido a mesma nem tão satisfatória quanto foi.

“Às vezes eu começo uma frase sem saber como ela vai terminar. Eu só torço para que ela chegue em algum lugar.”

(Michael Scott)

RESUMO

O JavaScript é uma linguagem de programação criada em 1995 que ganhou força principalmente no início da década de 2010 e que mudou o panorama de desenvolvimento de software, tanto para *web* quanto para *mobile*, se tornando uma ótima opção para quem quer desenvolver tanto no lado do cliente quanto para o lado do servidor e busca flexibilidade. Com o avanço da tecnologia e o uso da internet está atrelado o aumento na criação de aplicações *web* e *mobile* que agilizam tarefas do cotidiano e tiram a necessidade de uma pessoa ir a um estabelecimento realizar uma ação, por exemplo, um banco digital. Com o crescimento dessas soluções digitais em formato de software as exigências do usuário também se elevam, por exemplo, estabilidade do sistema, rapidez em resposta e segurança são atributos básicos e fundamentais para a utilização de um software, logo, no desenvolvimento de um software é necessário ter cuidado no momento de escolher o banco de dados e ferramentas de persistência que manipulam esse banco de dados, já que é nessa área em que todas as regras de negócio e dados devem ser implementados e mantidos de maneira segura e de fácil acesso. Atualmente existem vários *frameworks* de persistência de dados consolidados na comunidade de desenvolvedores para o desenvolvimento de software que podem implementar um ou os dois tipos de bancos de dados mais usados e tradicionais no momento de desenvolver um sistema, o banco de dados relacional e o banco de dados não relacional. Antes de qualquer linha de código ser escrita, é preciso ter em mente quais os requisitos funcionais e não funcionais que precisam ser respeitados e implementados no sistema, para que em fases posteriores do desenvolvimento haja o mínimo possível de retrabalho, atrasos e evitar até o insucesso do projeto. Portanto, o presente trabalho visa realizar uma comparação entre 6 *frameworks* de persistência de dados gerando um artefato em formato de tabela que serve como catálogo para desenvolvedores e instituições de tecnologia tenham uma melhor visão do que utilizar em seus projetos, diminuindo gastos com tempo de escolha de ferramenta, escolha de banco de dados, diminuição de retrabalhos e atrasos.

Palavras-chave: *Framework* de persistência. Banco de dados. JavaScript. Comparação. Critérios qualitativos. Critérios quantitativos

ABSTRACT

JavaScript is a programming language created in 1995 that gained strength mainly in the early 2010s and that changed the landscape of software development, both for web and mobile, becoming a great option for those who want to develop both on the client and server side and seeks flexibility. With the advancement of technology and the use of the internet, there is an increase in the creation of web and mobile applications that streamline everyday tasks and remove the need for a person to go to an establishment to perform an action, for example, a digital bank. With the growth of these digital solutions in software format, user requirements also increase, for example, system stability, quick response and security are basic and fundamental attributes for the use of a software, so in the development of a software it is necessary to be careful when choosing the database and persistence tools that handle this database, since it is in this area where all business rules and data must be implemented and maintained in a secure and easily accessible way. Currently, there are several data persistence frameworks consolidated in the developer community for software development that can implement one or both of the most used and traditional types of databases when developing a system, the relational database and the database. non-relational data. Before any line of code is written, it is necessary to keep in mind which functional and non-functional requirements need to be respected and implemented in the system, so that in later stages of development there is as little rework as possible, delays and even failure to be avoided. from the project. Therefore, the present work aims to make a comparison between 6 data persistence frameworks generating an artifact in table format that serves as a catalog for developers and technology institutions to have a better view of what to use in their projects, reducing spending on choice time. tool, database choice, decrease of reworks and delays.

Keywords: Persistence framework. Database. JavaScript. Comparation. Qualitative criteria. Quantitative criteria

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE QUADROS

SUMÁRIO

1 INTRODUÇÃO

No cenário atual de desenvolvimento de software, a maioria das empresas se encontram firmadas nos princípios da engenharia de software para desenvolver suas soluções com base nos critérios estabelecidos pelo cliente e dentro de prazos estipulados (PRESSMAN; MAXIM, 2016) visto que novas tecnologias emergem rapidamente e também para acompanhar o ritmo acelerado de desenvolvimento e entrega necessário no contexto atual da produção de tecnologia.

De acordo com Duarte (2015) essas soluções criadas para *web* possuem em suma, abstraindo muitos detalhes de implementação e manutenção dois setores principais, um deles é o *front-end* representando o lado do cliente englobando tudo o que é visual para o usuário, como telas, menus que usuários podem clicar, campos que usuários podem preencher com informações, botões, animações, e o outro é o *back-end* representando o lado do servidor, a parte responsável por armazenamento e manipulação dos dados resultados das implementações das regras de negócio que compõem o projeto padronizadamente no banco de dados e de fácil acesso que chegam via *front-end*, resumidamente.

No entanto, para que o *back-end* ou projeto de banco de dados fique padronizado e suporte determinados casos de uso, prazos sejam cumpridos, é necessário primeiro definir corretamente qual categoria de banco de dados e qual ferramenta de persistência será utilizada no projeto antes de qualquer ação e se ele é compatível com as exigências do cliente, para que no futuro após o término da aplicação ou ainda durante o desenvolvimento da mesma, haja o mínimo de retrabalhos ou realocação de recursos (financeiros ou humanos) que podem acarretar atrasos e má funcionalidade do sistema.

Escolher a ferramenta ideal não é uma tarefa simples, além de serem ferramentas com características funcionais próprias que podem ser incompatíveis com outras, podem ter baixa maturidade (DUARTE, 2015). É uma decisão que pode englobar desenvolvedores, arquiteto de software, gerente de projetos e líderes, impacta em definição de prazos, artefatos a serem entregues, tem que ser pensado na melhor opção custo-benefício e pode impactar até no *front-end* em questão de integração. Tomar essa decisão pode levar mais tempo do que uma reunião, por exemplo, se tomarmos como unidade de medida de uma organização, devido ao domínio e ponto de vista de cada um sobre certo *framework* a ser usado e o contexto em que estão inseridos.

A medida que novas ferramentas e tecnologias emergem a cada dia para facilitar e acelerar o desenvolvimento de software, visto que existe um problema de compatibilidade entre técnicas de armazenamento de dados e técnicas utilizadas por programadores para desenvolver,

linguagens de programação também avançam em seu crescimento, evoluindo e ganhando novos adeptos. Uma dessas linguagens de programação é o JavaScript, que pode ser definido como um ecossistema usado tanto no *front-end* quanto no *back-end* (PINHO, 2017). A linguagem JavaScript também é considerada a mais utilizada no mundo desde 2014 segundo Forsgren Kalliamvakou (2021)

A forma de como os dados são armazenados na maioria dos sistemas de bancos de dados é baseado no paradigma de conjuntos onde os dados são organizados em forma de tabelas, colunas e linhas, já os programadores usam o paradigma de orientação a objetos para desenvolver códigos. As ferramentas de persistência de dados analisadas no presente trabalho implementam padrões de persistência de dados diferentes e fazem a tradução do modelo de dado, seja de um objeto (concreto ou abstrato) para linhas do banco relacional ou o caminho inverso, visto que o caminho é bidirecional.

Atualmente no cenário de desenvolvimento globalizado e ágil, o gerenciamento e organização do banco de dados é algo fundamental para o sucesso do projeto, e para que as organizações sigam os princípios propostos pela metodologia ágil de desenvolvimento os recursos humanos e financeiros precisam ser coerentes com prazos definidos pelo escopo do projeto.

Para existir harmonia na relação de desenvolvimento dentro de prazos, e que esse desenvolvimento demanda recursos, é necessário ter definido antes de qualquer ação qual ferramenta e banco de dados utilizados ao longo do projeto e confirmar se as mesmas suportam determinados cenários de uso como escalada de acessos múltiplos, se o código gerado é de fácil legibilidade para manutenção e adição de funcionalidades e principalmente é preciso ter uma padronização no desenvolvimento para que posteriormente haja o mínimo de retrabalhos possível.

Visto que a medida que novas ferramentas surgem rapidamente para suprir as necessidades anteriores de organizações e desenvolvedores, a escolha do *framework* de forma errônea pode gerar problemas futuros no desenvolvimento como estimativas incorretas, retrabalho, curva de aprendizado elevado da equipe e alocação de mais recursos financeiros e/ou humanos. A decisão de usar um *framework* depende de causas como escopo do projeto, a quantidade de recursos disponíveis, capacidade de adaptabilidade, e por parte dos desenvolvedores é a capacidade de manutenção e atividade da comunidade de acordo com Gizas *et al.* (2012).

Uma vez apresentados fatores importantes para a criação e finalização de um soft-

ware de qualidade, o presente trabalho visa uma comparação entre uma série ferramentas de persistência de dados, sejam ferramentas que seguem o paradigma relacional ou não relacional (ou os dois paradigmas) avaliando características fundamentais para a escolha correta da solução a ser implementada por desenvolvedores ou organizações. Na comparação são avaliados atributos que podem contribuir no sucesso do projeto como, por exemplo, *insights* para desenvolvedores de quando e qual recurso o escolher, o tamanho da comunidade e sua proatividade, legibilidade, desempenho e curva de aprendizado e entre outros.

Os aspectos mencionados acima com os outros avaliados no presente trabalho, podem auxiliar a redução de custos da empresa dado que estimativa de recursos a serem alocados são calculados por meio da experiência da organização que realizará o desenvolvimento, ambiente e software a ser desenvolvido (PRESSMAN; MAXIM, 2021) sendo uma etapa fundamental para o planejamento já que é nessa fase que prazos e organização das fases posteriores ocorrerão.

O presente trabalho é direcionado a organizações que desenvolvem software de maneira ágil e que precisam ou visam reduzir custos ao escolher um *framework* de persistência de dados que satisfaça os requisitos cruciais para produzir um produto de qualidade no contexto de banco de dados. Também é direcionado a desenvolvedores de software *back-end* que pretendem aprender novas tecnologias ou analisar qual a ferramenta que melhor se encaixa em seu projeto ou estudo.

Diferente de outras análises de desempenho disponíveis na literatura, este trabalho visa uma comparação mais abrangente em número de ferramentas JavaScript, tanto em operações padrão do banco de dados (criar, buscar, atualizar, deletar) não se prendendo a contextos específicos, tornando possível que o leitor consiga escolher a ferramenta ideal de acordo com sua necessidade, mas também são considerados na comparação aspectos mais atrativos para o possível desenvolvedor como documentação, tamanho e atividade da comunidade, por exemplo, em Zyl *et al.* (2006) é analisada o desempenho em operações típicas de banco de dados o *framework* de persistência e mapeamento objeto-relacional Hibernate, em Java, e o banco de dados para objetos db4o, e em Barmpis e Kolovos (2012) são comparados *frameworks* de persistência que seguem o paradigma relacional e não relacional, mas no contexto e direcionada a aplicações industriais com problemas em escalabilidade que seguem *Model-Driven Engineering* (MDE).

1.1 Objetivos

Nesta Seção, são descritos o objetivo (geral) e objetivos específicos do presente trabalho, que guiam e embasam a realização do trabalho que visa contribuir com a comunidade acadêmica e de desenvolvimento.

1.1.1 Objetivo geral

O objetivo do trabalho é realizar uma análise comparativa entre *frameworks* de persistência de dados mais populares em JavaScript com o intuito de apontar qual o mais adequado a determinado contexto de modo a reduzir custos e aumentar eficiência em projetos de software.

1.1.2 Objetivos específicos

- Definir as características comparativas em diferentes *frameworks* de persistência em JavaScript;
- Definir os *frameworks* a serem comparados
- Comparar diferentes características de cada *framework* de persistência de dados em ações habituais de banco de dados em um mesmo código JavaScript;

Os seguintes capítulos estão dispostos em: Capítulo 2 apresentando os trabalhos relacionados a este trabalho e que servem de embasamento de entendimento e realização do presente trabalho; No Capítulo 3 será apresentada a fundamentação teórica onde estará presente termos técnicos e embasamento literário para a realização da comparação proposta pelo trabalho; No Capítulo 4 será apresentada a metodologia usada e descrição dos passos seguidos para a realização deste trabalho; No Capítulo 5 são apresentados os resultados obtidos a partir da metodologia usada; Por último, no Capítulo 6 são apresentadas as conclusões a partir dos resultados obtidos, com as contribuições do presente trabalho e trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo serão mostrados e explicados os principais conceitos técnicos para o entendimento deste trabalho. Primeiro será introduzido o conceito de bancos de dados e seus tipos na Seção 2.1; Será posteriormente apresentado o problema da Impedância Objeto-Relacional na Seção 2.2 para contextualizar o motivo de se usar *frameworks* de persistência; Em seguida será apresentado o conceito e funcionamento de *frameworks* de persistência de dados na Seção 2.4 como sub-itens os *frameworks* usados como objeto de estudo neste trabalho; Como último ponto a ser conceituado, há os padrões de conservação dos valores que os *frameworks* comparados no estudo implementam, e seus sub-itens que representam as categorias de padrão que as ferramentas implementam.

2.1 Bancos de dados

Segundo Silberschatz *et al.* (2008), um banco de dados é o valor resultante de um conjunto de dados inter-relacionados que juntos representam informações e sentido sobre um domínio específico, ou seja, se valores associados a um escopo forem persistidos mesmo que implicitamente, este se torna um banco de dados. Como apenas dados relacionados fazem sentido, valores mesmo que possam ser armazenados manualmente em listas, folhas ou computadorizadamente, mas de forma aleatória e sem sentido, não formam um banco de dados.

Os bancos de dados automatizados que são os tipos estudados no presente trabalho, são mantidos com os dados persistidos por programas chamados Sistema Gerenciador de Banco de Dados (SGBD) que permitem o usuário criar, sustentar e realizar ações em seu banco de dados (ELMASRI *et al.*, 2005). Saber no que vai se trabalhar ou qual o escopo do projeto é essencial para a definição do banco de dados, qual seu tipo e o dado que ele armazena e no contexto de desenvolvimento de software atual estão compreendidos majoritariamente em duas alas, os bancos de dados relacional e os bancos de dados não-relacional.

2.1.1 Bancos de dados relacionais

Os bancos de dados relacionais (RDBMs) surgiram primeiramente na década de 1970, idealizados pelo pesquisador Edgar Frank Codd da IBM e se firma no paradigma dos conjuntos que representa o estudo da relação dos conjuntos e seus elementos e o modelo relacional que

considera um número finito de relações. A estrutura do banco relacional é disposta em tabelas que representam as entidades reais, colunas que representam os atributos dessas entidades, e as linhas (ou tuplas) responsáveis por armazenar o valor que serão característicos aos atributos. Cada tabela possui uma chave identificadora que a torna única em comparação as demais, é chamada chave primária.

Anteriormente citado, o modelo relacional permitiu que as aplicações otimizassem suas estruturas de dados não armazenando valores em uma estrutura única, mas de uma forma separada assim pessoas com menos conhecimento do projeto ou organização das tabelas pudessem realizar buscas. Nos primeiros anos de bancos de dados relacionais haviam duas formas de realizar consultas, distintas mas equivalentes, a primeira forma era baseada na lógica de primeira ordem, com consultas criadas unindo fórmulas atômicas, conectivos lógicos e quantificadores (totais ou existenciais).

Já na segunda maneira era baseada na álgebra relacional e os dados de uma relação que satisfazem a condição no banco de dados (YANNAKAKIS, 1996). No meio empresarial surgiu a *Structured Query Language* (Linguagem de Consulta Estruturada), que baseada em matemática relacional e teoria dos conjuntos fornecem respostas a consultas efetuadas no banco.

2.1.2 Bancos de dados não relacionais

Com o avanço da *internet* e a necessidade de escalar as soluções computacionais de empresas, surgiu uma alternativa de organizar e escalar o banco de dados de maneira mais econômica em relação ao modelo relacional, com seu primeiro conceito apresentado em 1998 por Carlo Strozzi.

No banco de dados não relacional os dados não são agrupados em forma de tabelas, mas em formas mais maleáveis na estruturação dos dados, como, por exemplo, *key-values stores* onde um par de chave e valor é armazenado, *document stores* onde um objeto é criado para englobar os atributos que podem ser de tipo misto, *graph stores* os valores se relacionam a partir de grafos que é a interligação de arestas e nós, entre outras formas. Essa categoria de banco de dados também é chamado de *No Only Structured Query Language* (NoSQL), isto é, não é utilizado SQL para realizar consultas nos dados e são geralmente utilizados a um domínio mais específico devido a sua forma de realizar consultas e forma de armazenar elementos.

Diferente do modelo relacional, a forma de escalar o armazenamento no banco é horizontalmente, diferente do modelo relacional, e não dependem estritamente do *hardware*

disponível (STRAUCH *et al.*, 2011).

2.2 Impedância Objeto-Relacional

Ao utilizar diferentes tecnologias para construir um software o projeto está sujeito a problemas de incompatibilidades devido a funções específicas que cada solução usada foi projetada para realizar e o paradigma que cada uma segue (IRELAND *et al.*, 2009). A impedância objeto-relacional é um problema de incompatibilidade constatado entre modelos de dados relacionais usados em RDBMs que seguem o paradigma de conjuntos e a metodologia de desenvolvimento orientada a objetos utilizada pelos desenvolvedores onde a estruturação e persistência de dados não é estabelecida por tabelas, colunas e linhas, mas sim por um objeto real com características chamadas atributos.

Ainda segundo Ireland *et al.* (2009) neste problema as soluções seguem paradigmas diferentes, cada paradigma com seus princípios e abstrações, assim tornando as metodologias não equivalentes, visto que o modelo relacional é utilizado para desenvolver bancos de dados e a orientação a objetos é utilizada como metodologia de desenvolvimento de software, mas devido à necessidade de criar soluções complexas em algum momento será preciso usar às duas metodologias em conjunto.

2.3 Padrões de persistência de dados

Partindo de um conceito mais amplo, um padrão pode ser conceituado como um conjunto de passos ou instruções que seguidos solucionam ou evitam um problema que é recorrente, um padrão pode corrigir mais de um problema além do que ele foi criado especificamente para corrigir. No desenvolvimento de software seja em aplicações corporativas ou em projetos pessoais também existe a necessidade de implementação de padrões, onde é preciso tomar decisões importantes e rápidas no contexto de desenvolvimento ágil.

Entender o que é um padrão é entender a essência da problemática, para que em tentativas de corrigir empecilhos de desenvolvimento seja encontrada a chave para a resolução do problema. Nas categorias de projetos anteriormente citados, mas principalmente em aplicações empresariais, existe a necessidade de persistir dados para que funções no sistema sejam executadas sendo muitas vezes execuções simultâneas.

Também é muito provável que a estrutura de armazenamento desses dados ou que fazem a manipulação deles sofram alterações durante o ciclo de seu desenvolvimento, e as

novas mudanças devem garantir que novos dados sejam persistidos sem afetar os já presentes no sistema (FOWLER, 2012).

Qual padrão de persistência de dados utilizar depende da decisão de como o projeto será arquitetado e do domínio da aplicação, como, por exemplo, se a camada de domínio e regras de negócio poderá ter acesso direto ao banco de dados em comando SQL ou essa parte será abstraída para que entidades sejam apenas manipuladas com seus elementos e não manipuladoras diretas.

2.3.1 Active record

Segundo Fowler *et al.* (2003) este padrão se caracteriza pela não abstração da camada de regras de negócio (ou domínio da aplicação) da entidade criada. A entidade ou modelo a ser criado representa um registro ativo e cada atributo da classe é equivalente uma coluna no banco de dados, além disso, cada registro ativo consegue realizar ações de manipulação e persistência dos dados aos quais ela é responsável por envolver, além da possibilidade de conectar-se ao banco de dados diretamente, o que pode tornar em certos casos o código ambíguo visto que a camada de acesso a elementos e lógica do domínio são fortemente ligadas.

Em primeiro plano, este padrão é recomendado em projetos com regras de negócio mais simples com manipulações típicas no banco de dados como criar, ler, atualizar e deletar dados (CRUD), para ações mais complexas é recomendado padrões que abstraiam a camada de negócio da camada de dados. Outra característica deste padrão é o forte acoplamento do projeto de objetos com o banco de dados (FOWLER, 2012), o que pode resultar em dificuldades de refatoração e manutenção do software.

O *Active record* representa o padrão de persistência mais simples dado que as manipulações são de responsabilidade do modelo, mas acaba por exemplo ferindo o Princípio de Responsabilidade Única ou *Single Responsability Principle* (SRP) que forma o primeiro pilar do SOLID (conjunto de 5 padrões que facilitam o desenvolvimento e entendimento de um software orientado a objetos), visto que a camada de dados ou arquivo que modela os dados pode realizar manipulações diretas sobre os atributos que ela envolve, além também de poder se conectar diretamente no banco de dados.

2.3.2 Data mapper

O padrão mapeador de dados atua como uma camada intermediária entre o banco de dados relacionais e a entidade do negócio em forma de objeto, isolando às duas, mas transferindo elementos entre elas, assim abstraindo as regras de domínio da entidade fazendo com que ela não se comunique diretamente com o banco de dados atribuindo desta forma apenas a modelagem de elementos a entidade isolando ela de manipulações diretas em SQL (FOWLER *et al.*, 2003).

Este padrão visa solucionar principalmente o problema da impedância objeto-relacional, onde citado anteriormente, há a incompatibilidade entre a estrutura de dados de objetos e bancos de dados relacionais. Diferente do padrão *Active record* a responsabilidade da entidade é bem definida e não fere o SRP do SOLID, aqui as manipulações de dados são feitas por outras classes chamadas repositórios e os objetos criados na memória não conhecem o esquema do banco de dados, reduzindo as chances de mudanças se propagarem de um modelo para o outro.

2.3.3 Indentity Map

Este padrão mantém em um mapa, objetos que já foram encontrados em consultas anteriores e retornando ele quando necessário, verificando primeiro no mapa se o objeto está presente, assim ele garante que objetos não sejam buscados várias vezes reduzindo chamadas desnecessárias no banco de dados. Para que esse padrão possa ser implementado deve-se tomar decisões acerca de seu uso, como por exemplo, o que pode identificar um objeto no mapa de identidade. Esse atributo geralmente é a chave primária da tabela, já que é um valor que a distingue de outras tabelas. Outra decisão a ser tomada é definir se um mapa de identidade é explicito ou genérico, na forma explícita um método específico de acesso é usado para cada categoria de objeto e na forma genérica apenas um método é criado para todos os tipos (FOWLER, 2012).

Ferramentas de persistência que implementam esse padrão tendem a ter melhoria de desempenho visto que esse padrão permite que todos os objetos lidos sejam guardados em uma única transação comercial, assim se uma chamada for feita duas vezes ao banco de dados será observado no mapa de identidade se o valor da busca consta em memória (FOWLER *et al.*, 2003), ou seja, o aplicativo terá um melhor tempo de resposta ao descartar chamadas ao banco desnecessárias e está menos propenso a carregar e escrever dados repetidos.

2.3.4 Unit of Work

Alterações no banco de dados são operações típicas que precisam ser executadas, como adição de um novo registro, edição ou exclusão de um já existente. O funcionamento deste padrão se dá pela criação de uma unidade de trabalho sempre que alguma ação impacta o banco de dados alterando seus valores, mantendo um registro das ações que afetaram os dados. Segundo Fowler (2012), para que as alterações sejam incluídas na unidade e confirmadas existem duas maneiras:

1. Fazer com que o solicitante do objeto inclua as mudanças na unidade de trabalho;
2. O objeto se inclua na unidade de trabalho;

Na primeira forma apesar de ser um esforço manual, possibilita que desenvolvedores não incluam alterações indesejadas ou erradas no banco de dados, pois não estão incluídos na unidade. Assim como os outros padrões já citados, ele também independe do funcionamento de outros padrões, mas pode funcionar como um complemento do *Identity Map* já que ele cria uma lista com os dados afetados por uma transação e coordena a persistência simultânea desses dados eliminando erros nessa parte.

Garantir que mudanças no banco de dados sejam feitas é uma parte importante do desenvolvimento e pode dificultar a medida que chamadas escalam, mesmo que pequenas, se tornam cumulativas e acaba afetando o tempo de resposta de consultas. Segundo Fowler *et al.* (2003) uma Unidade de Trabalho analisa todas as alterações que podem afetar o banco de dados e ao final das alterações, a Unidade saberá o que modificar, por exemplo, garantir que um dado seja criado ou excluído de fato.

2.4 *Frameworks* de persistência de dados

Na literatura existente existem muitas definições para *frameworks*, como, por exemplo, para Fayad *et al.* (1999), *frameworks* são um conjunto de classes que colaboram entre si e são responsáveis por para executar ações de domínio específico de um subsistema de aplicação. Então para auxiliar os desenvolvedores *back-end* no problema de tradução de um objeto para linhas do banco de dados relacional, ou vice-versa, surgiram os *frameworks* de persistência de dados que definem como os valores serão mapeados, isto é, traduzem um dado de um modelo para o outro e isolam a responsabilidade do desenvolvedor de conhecer operações e regras específicas do banco de dados usado, mantendo a orientação a objetos o paradigma para desenvolver

código.

Qual ferramenta escolher varia conforme a linguagem de programação usada e o banco de dados escolhido, mas também são valores que podem mudar dependendo do escopo do projeto e requisitos propostos pelo cliente. Estas ferramentas que mapeiam um objeto para o modelo relacional são chamados *Object-relational Mapping* (ORM), mas também existem os *Object-document Mapping* (ODM) voltados para bancos de dados não relacionais.

Cada uma das técnicas de mapeamento implementam suas regras específicas de tradução e padrões de persistência, e traduzem um objeto para outra estrutura de dados, por exemplo, um objeto é traduzido (convertido) em linhas do banco de dados usando ORMs e um objeto é traduzido em um grafo ou documento usando ODMs.

O motivo para ODMs possuírem mais modelos disponíveis para traduzir um objeto é dado que as estruturas de dados não relacionais são mais variadas que o modelo relacional, mas dão suporte a situações e categorias de elementos específicos, como anteriormente citado na Subseção 2.1.

2.4.1 TypeORM

Diferente de todos os outros *frameworks* de persistência em JavaScript, ele possibilita não somente usuários desenvolverem aplicações simples com regras de negócio ligadas a modelos mas também aplicações mais robustas e complexas com cada responsabilidade definida, visto que este ORM consegue suportar ambos os padrões de persistência *Active record* e *Data mapper*. Ele pode ser executado nos *runtime environments* NodeJs e navegador, são ambientes que dão todo o suporte para que essas aplicações criadas em JavaScript possam rodar no lado do cliente. Além de oferecer suporte a diversos bancos de dados (relacionais e não relacionais) esta ferramenta pode ser usada com JavaScript puro (ES5, ES6, ES7, ES8) ou com TypeScript, um *superset* para desenvolver códigos JavaScript com mais eficiência, adicionando recursos que não existem por padrão na linguagem. A documentação do *framework* está disponível em <https://typeorm.io/>

2.4.2 Prisma ORM

O Prisma é um projeto de código de aberto que se diferencia de ORMs tradicionais, mas ambos tem o mesmo objetivo que é isolar o desenvolvedor de conhecer detalhes do banco de dados e contornar o problema da impedância objeto-relacional. Ele se diferencia pela quantidade de “produtos” disponíveis do próprio Prisma que podem ser utilizados como facilitadores no

desenvolvimento, são eles o Prisma Client responsável pela conexão no banco de dados e consulta sobre os dados; O Prisma Migrate responsável por ler o *schema* (arquivo onde todos os modelos são definidos) e gerar as migrações que atuam como histórico do banco de dados e definem como as entidades estão modeladas, então se houver alguma alteração no *schema*, o Prisma Client lê e atualiza as migrações; O Prisma Studio atua como um SGBD possibilitando o desenvolvedor visualizar e manipular o banco de dados sem a necessidade de instalar uma ferramenta de terceiros e com uma linha simples de comando. Além de oferecer a bancos de dados relacionais e não relacionais este *framework* utiliza o *Data mapper* como padrão de persistência. A documentação do *framework* está disponível em <https://prisma.io/>

2.4.3 Mongoose

É uma ferramenta modeladora de dados que funciona como ODM de uso exclusivo do banco de dados não relacional MongoDB. Seu objetivo como ODM é traduzir os atributos de uma classe para um documento que representa um registro em uma *collection* (comparando ao banco de dados relacional é uma tabela) no banco de dados e cada atributo do documento se comparando ao banco de dados relacional representa uma coluna.

É um *framework* usado para criar aplicações JavaScript recomendado principalmente para soluções que possuem menos complexidade e relacionamentos, dado que o *Active record* é o padrão de persistência implementado, mas que podem receber quantidades abundantes de dados devido como o Mongoose estrutura os dados em *collections*, o que torna este *framework* muito recomendado em aplicações simples pelo fácil uso (curva de aprendizado, configuração) principalmente para quem é iniciante em desenvolvimento pela possibilidade de entidades no MongoDB (*collections*) realizarem consultas no banco diretamente. A documentação do *framework* está disponível em <https://mongoosejs.com/docs/guide.html>

2.4.4 Typegoose

Igualmente ao Mongoose, é um ODM para o banco de dados MongoDB, mas a sua diferença para o anterior é a possibilidade de usar o TypeScript, o *superset* já citado para adicionar recursos extras para desenvolver códigos JavaScript. Ele também visa reduzir o código e trabalho de manter os modelos atualizados, visto que ao utilizar o Mongoose e TypeScript é preciso definir em dois locais os atributos e sincronizá-los casa haja alguma mudança, na *interface* do TypeScript e no modelo do Mongoose, já com Typegoose é necessário apenas definir apenas em um arquivo

que é a *interface* TypeScript que funciona como uma classe. A documentação do *framework* está disponível em <https://typegoose.github.io/typegoose/docs/guides/quick-start-guide>

2.4.5 Sequelize

É um ORM para aplicações JavaScript que disponibiliza a possibilidade de integração com TypeScript e oferece suporte apenas a bancos de dados relacionais. O diferencial deste ORM é que ele segue o *Semantic versioning*, onde existe uma definição de como as versões de dependências sejam incrementadas e definidas para evitar o *"dependecy hell"*, um conceito em engenharia de software para projetos que a medida que escalam e incrementam cada vez mais as ferramentas de terceiros (chamadas dependências) se arriscam a travar a liberação do projeto ou por uma dependência atualizada logo quando lançada ou pelo projeto ou outros fatores não suportarem mais a dependência definida a algum tempo. A documentação do *framework* está disponível em <https://sequelize.org/docs/v6/getting-started/>

2.4.6 MikroORM

Por último, o MikroORM é um *framework* de persistência que utiliza por padrão TypeScript para desenvolvimento de código sem abertura para o JavaScript puro, e assim como outros ORMs citados ele oferece suporte a bancos de dados relacionais e não relacionais, mas o que difere este de outros são os padrões de persistência em que ele é baseado, sendo a maior quantidade de todos os descritos até então ele implementa o *Indetity Map*, *Unit of Work* e o *Data Mapper*.

Este ORM também usa migrações como definição dos campos de tabelas e histórico do banco de dados, também auxilia os desenvolvedores a sincronizar o banco de dados com o conceito de *DRY Entities*, ele analisa o código para não ser preciso atualizar manualmente entidades. A documentação do *framework* está disponível em <https://mikro-orm.io/docs/installation>

2.5 Paralelismo e Concorrência

São dois conceitos que podem aparecer tanto em baixo nível no mundo da tecnologia como em estudos de Sistemas Operacionais (SO) ao nível de *hardware*, em operações de entrada e saída que precisam ser executadas pelo sistema operacional, quanto a mais alto nível em

desenvolvimento de software com tarefas a serem executadas. Apesar de serem áreas distintas e cada uma com suas peculiaridades, os conceitos de paralelismo e concorrência podem ser aplicadas nas duas áreas igualmente.

A diferença básica entre às duas é que paralelismo dentro do sistema operacional ou software, é um conceito que se preocupa em realizar tarefas ao mesmo tempo. Para esse conceito podemos fazer a analogia de um pedágio, onde diversos veículos podem fazer a mesma ação ao mesmo tempo. Segundo Tilkov e Vinoski (2010), com a evolução das aplicações e desenvolvimento de software as técnicas de paralelismo usadas em SOs também passaram a ser usadas em sistemas *web*, tornando sistemas que possuem grandes operações de entrada e saída sejam mais eficientes, visto que operações podem ser executadas de forma simultânea. Apesar de trazer benefícios como otimização em tempo de resposta ao usuário e troca de contexto, que consiste na mudança de recursos alocados entre processos para que o processador não fique ocioso, o paralelismo em desenvolvimento de software pode ser uma característica complicada de atingir, dado que entre os processos a serem executados quem é o primeiro escolhido depende do sistema operacional, tirando o controle do desenvolvedor.

Já a concorrência em desenvolvimento de software, de acordo com Rocha (2022), é um conceito relacionado a como lidar com tarefas simultâneas, então podemos fazer a seguinte analogia, pense em um sistema de *drive thru* onde cada carro terá um tempo determinado para fazer o seu pedido e recebê-lo, caso não receba ele terá mesmo assim de sair da fila para que posteriormente quando o atendente fique livre, possa atendê-lo novamente, isso é um exemplo de concorrência. Com essa analogia podemos perceber como tarefas são concorrentes entre si, quando todas tem a mesma chance de ser executada e caso não tenham sido completadas por motivos de necessidade de execução de outra tarefa, por exemplo, os recursos alocados para ela são direcionados a outra tarefa que está pronta para ser executada.

Este último conceito é fortemente atrelado a outro conceito chamado de *multithreading*, que consistem em fluxos de execução concorrente (SILVEIRA *et al.*, 2021), e podem ser vistos tanto ao nível de sistemas operacionais quanto a tarefas a serem executadas em um software. Para aplicar o conceito de *multithreading* o desenvolvedor deve ter em mente o fluxo de execução de tarefas, dado que se pensarmos uma operação simples de entrada e saída onde no meio do fluxo existe o processamento que necessita da entrada, e a saída necessita dos dados processados, não há como haver uma concorrência entre as tarefas por que uma depende da outra.

2.5.1 Concorrência em JavaScript

Ainda segundo Rocha (2022), a concorrência entra tarefas no JavaScript difere das outras (Java, por exemplo), visto que essa linguagem de programação não conta com o mecanismo de troca de contexto, chamado de mecanismo de preempção, forçando uma tarefa ser executada por completo para avançar para a próxima, essa estratégia no JavaScript se chama *Run-to-completion*. Essa estratégia permite a criação de aplicações concorrentes em JavaScript mais simples, onde o desenvolvedor tem mais controle do que pode ser executado concorrentemente e diminui casos de execução possíveis a se pensar.

2.6 Assincronismo e Sincronismo

A partir do crescimento do JavaScript nos últimos anos, o Node.Js, um ambiente de execução (*runtime environment*) para JavaScript que elimina a necessidade de o navegador para construir aplicações, ganhou notoriedade entre desenvolvedores para desenvolver software para *back-end*. Em paralelo ao crescimento do JavaScript, a comunidade de desenvolvedores buscou e busca até hoje melhores formas de criar aplicações para *back-end* que performem melhor no quesito resposta ao usuário (LORING *et al.*, 2017). Ainda de acordo com Loring *et al.* (2017) essa busca por melhoras tanto em softwares quanto no próprio Node.Js não se dá somente para criar softwares mais performáticos, mas para padronizar as estratégias de assincronismo disponibilizadas pelo Node.Js, como APIs, promessas (*promisses*) e *callbacks*.

Conforme mostrado por Belson *et al.* (2019), no contexto de sistemas embarcados mas também no desenvolvimento de software no geral, a maior parte dos códigos possui um fluxo assíncrono de execução que consiste basicamente que tarefas possam ser executadas independentes das outras (ou pelo menos a maioria), que se assemelha bastante ao conceito de Concorrência mostrado em seção 2.5, inclusive o modelo de programar assíncrono é uma maneira de se alcançar a concorrência em um software que não oferece previsibilidade.

Ainda seguindo a linha de raciocínio de Belson *et al.* (2019) existem duas formas de aplicar assincronismo em código que são mais confiáveis (simples e seguras) e que são mais difundidas na comunidade, como o padrão do *callback* e o padrão *async-await*. Este último padrão atualmente mais utilizado prescrevendo que todas as operações são assíncronas mas mantendo uma ordem de execução mesmo que as operações sejam separadas de arquivos ou blocos.

Já o sincronismo é uma forma de programar um pouco mais antiga do que o assincronismo e foi introduzia na década de 1980 como uma forma mais segura de criar sistemas embarcados críticos (GAMATIÉ, 2010). Este paradigma de programação tem certos objetivos e especificações para garantir o funcionamento e escalonamento de aplicações caso necessário, como por exemplo a principal diferença entre o assincronismo, que é a previsibilidade de funcionalidades do sistema que pode nos dizer com exatidão qual seria o comportamento adequado de certa função em interação com o usuário. O comportamento da previsibilidade e executar passo após passo do sincronismo parte a partir de um hipótese, que segundo (GAMATIÉ, 2010) a suposição é de que uma linguagem síncrona consegue processar uma saída tão rápido quanto receber outra entrada, onde o tempo tem um papel fundamental para assinalar o desempenho do programa.

3 TRABALHOS RELACIONADOS

Nesta Seção, são descritos os trabalhos relacionados ao presente trabalho de modo a contextualizar em qual domínio o presente trabalho se encaixa. Neste capítulo são apresentados os trabalhos que realizam análises comparativas entre *frameworks* de mapeamento objeto-relacional nos quesitos de desempenho com subitens e práticas de implementação aplicadas em .NET e Java.

3.1 *Comparative analysis of selected object-relational mapping systems for the .NET platform*

No trabalho proposto por Drzazga *et al.* (2020) são avaliados e comparados dois *frameworks* de persistência que realizam mapeamento objeto-relacional em .NET, um ambiente de desenvolvimento que provê todos os recursos (*hardware* e software) para determinado trecho de código rodar. A linguagem de programação utilizada para os testes foi a C#, visto que é a linguagem padrão do .NET sendo testados as ferramentas Entity Framework Core (na versão 2.2.4) e NHibernate (na versão 5.1.1) para validar ou rejeitar a hipótese de que o NHibernate é mais eficiente em ações de *Data Manipulation Language* (DML) que representam as ações padrão de banco de dados de criar, buscar, atualizar e deletar.

Como meio de validar ou rejeitar a hipótese foi desenvolvida uma aplicação *desktop* para medir o tempo nas ações de teste, uma prática semelhante ao trabalho de Wiphusitphunpol e Lertrusdachakul (2017) onde a análise de eficiência dos ORMs era feita a partir do tempo decorrido da memória em ações de criar e buscar dados, neste trabalho a aplicação utilizada não englobava todos os métodos DML, diferentemente do estudo realizado por Drzazga *et al.* (2020) que além de comparar os *frameworks* em todas as ações, equipou o projeto para visualizar os resultados dos testes e armazenamento destes dados.

Para realizar a comparação entre *frameworks* JavaScript proposta no presente trabalho, também será desenvolvida uma aplicação para os testes de desempenho igualmente no trabalho acima, mas no formato *Application Programming Interface* em JavaScript para que testes de eficiência e tempo de resposta em ações DML em cada uma das 6 ferramentas de persistência de dados escolhidas e que são descritas na Seção 2.

3.2 *Comparative Analysis of Data Persistence Technologies for Large-Scale Models*

A Engenharia Orientada a Modelos (MDE) é um conjunto de práticas que permitem o desenvolvimento de softwares de maneira mais barata e mais rápida no meio industrial, mas quando se trata de grandes projetos esse modelo fica comprometido por dois fatores:

1. Persistência dos modelos: A capacidade do sistema armazenar quantidades abundantes de dados (modelos), buscar e atualizar simultaneamente estes dados quando múltiplos usuários os acessam;
2. Buscas e transformações de modelos: Capacidade do sistema realizar buscas em modelos extensos e conseguir transformar estes resultados em dados visíveis para o usuário.

Aplicações industriais que seguem no desenvolvimento de seus projetos a MDE muitas vezes sofrem impedimentos em escalar visto que o armazenamento de dados persistente precisa ser organizado e eficiente.

Para saber qual permite uma melhor padronização do banco de dados e escalabilidade no contexto de sistemas desenvolvidos segundo a MDE, é feito uma comparação no formato *benchmarking* para apontar qual a mais adequada a este domínio entre ferramentas inovadoras na persistência de dados, como NoSQL baseado em grafos com Neo4J, OrientDB e Teneo-Hibernate com MySQL, que neste caso, implementadas na linguagem de programação Java.

Para cada uma das categorias de banco de dados foi implementado um protótipo e testados operações de inserção e busca em modelos de tamanhos diferentes, cada um dos bancos de dados foram submetidos a todos os modelos a modo de comparar seu desempenho e tempo de resposta segundos.

Diferente do trabalho descrito acima, a comparação deste trabalho é referente a ferramentas de persistência de dados e seu desempenho em operações de bancos de dados e não a comparação das categoriais de bancos de dados, e também não visa um contexto específico como aplicações empresariais que seguem a MDE.

3.3 *An Empirical Study on the Practice of Maintaining Object-Relational Mapping Code in Java Systems*

Com o avanço de sistemas computacionais e suas complexidades o uso de ORMs tem se tornado uma ótima alternativa para ajudar desenvolvedores a construírem mais rapidamente e eficiente sistemas, abstraindo a necessidade do desenvolvedor conhecer manipulações diretas no

banco de dados e também trechos repetidos de código.

No entanto, no estudo realizado por Chen *et al.* (2016) em um sistema empresarial parceiro e em outros 3 sistemas de código aberto escritos em Java utilizando ORMs foram identificados custos ocultos de manutenção nessa categoria de código em relação ao código procedural tradicional em Java, como, por exemplo, as alterações em códigos ORM são mais frequentes do que o código procedural devido a questões de segurança e desempenho, mas códigos tradicionais precisam ter funcionalidades a mais para conseguir captar tais situações para melhorar seu código.

Como na literatura existente são poucas as ocorrências para estudos comparativos de ORMs e uma maior parte de trabalhos de comparação entre categorias de bancos de dados como o próprio trabalho indica, foram estipuladas 3 perguntas-chave para guiar o estudo e serem respondidas ao longo da comparação, são elas:

1. Como as mudanças dentro de um código ORM são localizadas?
2. Como o código ORM muda?
3. Porque o código ORM muda?

Para responder às perguntas foi desenvolvido um analisador de código Java que avaliava os 4 sistemas de larga escala com implementações arquiteturais diferentes, mas que tinham o mesmo ORM em comum, o JPA. Como o código ORM pode ter diferentes chamadas ou anotações em sua implementação foram classificadas 3 características para diferenciar a categoria de código ORM, como a forma que o ORM usado modela dados e mapeamentos, o desempenho em configuração e consultas ao ORM para realizar ações de banco de dados.

Para contribuir com a literatura o presente trabalho efetua uma análise comparativa entre diferentes ORMs e ODMs considerando outros fatores quantitativos em relação a desempenho e número de arquivos afetados como também aspectos qualitativos, o que se difere do trabalho supracitado, que fornece uma comparação entre sistemas de larga escala escritos em Java que utilizam ORM em seu desenvolvimento e sistemas tradicionais que não utilizam.

Quadro 1 – Comparação entre trabalho proposto Vs. Trabalhos relacionados

Trabalhos	Número de frameworks comparados	Tipos de bancos de dados comparados	Linguagem usada	Aspectos quantitativos	Aspectos qualitativos	Código-fonte disponibilizado
Drzazga *et al.* (2020)	2	Relacional	C#	Operações DML	Formas de implementar um mesmo código	Não
Wiphusitphunpol, Lertrusdachakul (2017)	2	Relacionais e não relacionais	Java	Inserção, busca, escalabilidade	Padronização do banco de dados	Não
Chen *et al.* (2016)	1	Relacional	Java	Consultas ao ORM, performance em configuração	Modelagem de dados	Não
Este trabalho	6	Relacionais e não relacionais	JavaScript	Operações DML, custo de treino, custo de manutenção, banco de dados suportados, suporte a pool de conexões, tamanho da comunidade	Curva de aprendizado, simplicidade de configuração, facilidade de escrita, documentação, padrões de persistência	Sim

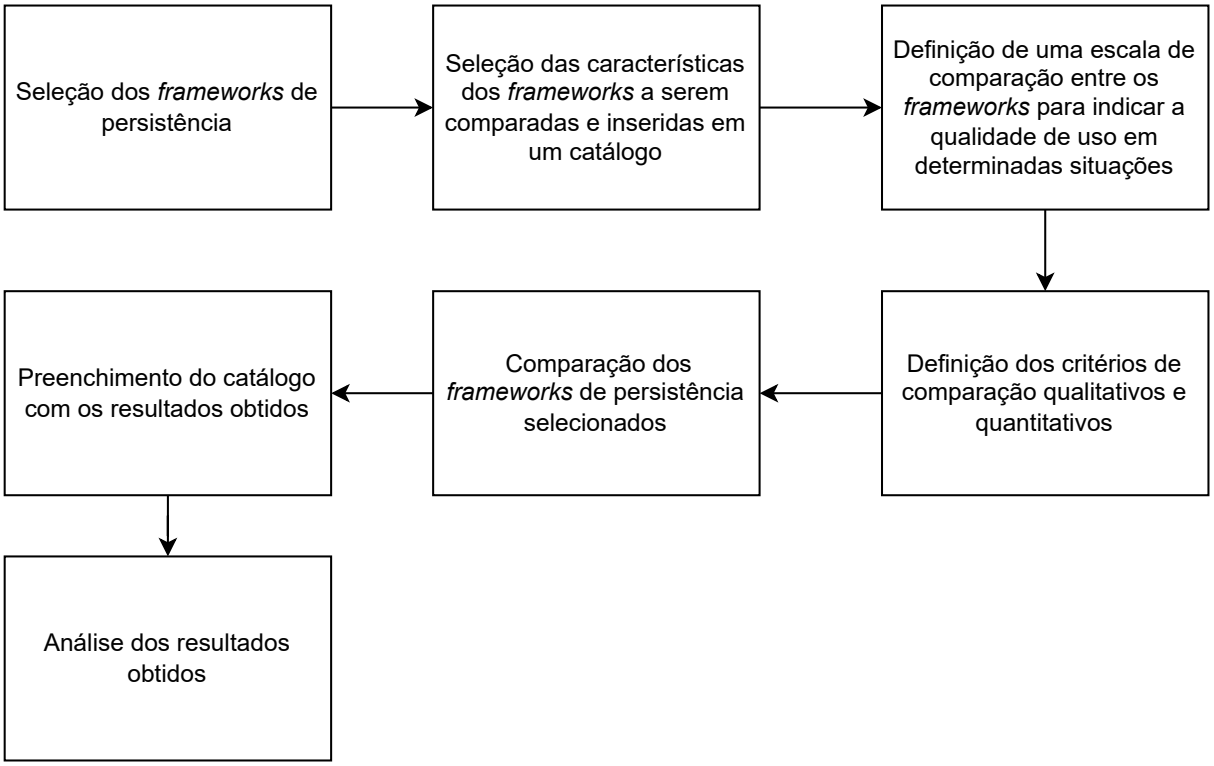
Fonte: elaborada pelo autor.

4 PROCEDIMENTOS METODOLÓGICOS

Neste capítulo são descritos os procedimentos metodológicos necessários para a realização deste trabalho. Ao total, consistem em 7 passos descritos nas Seções a seguir, cada uma direcionada a um procedimento e ao final deste capítulo, é apresentado o cronograma de execução dos passos.

Os procedimentos consistem em: Seleção dos *frameworks* de persistência, seleção das características dos *frameworks* a serem comparadas e inseridas em um catálogo, definição de métricas de comparação entre os *frameworks* para indicar a qualidade de uso em determinadas situações (presentes no catálogo), comparação dos *frameworks* de persistência selecionados, definição dos critérios de comparação qualitativos e quantitativos, preenchimento do catálogo com os resultados obtidos e por último análise dos resultados. A Figura 1 representa ilustrativamente o fluxo dos passos seguidos para realização do presente trabalho.

Figura 1 – Fluxo de execução dos procedimentos metodológicos



Fonte: elaborada pelo autor.

4.1 Seleção dos *frameworks* de persistência

Este passo consiste na escolha dos *frameworks* de persistência de dados a partir da linguagem de programação JavaScript. Conforme o dado citado no Capítulo 1 de que o JavaScript é a linguagem de programação mais usada no mundo desde 2014, é notória a necessidade de estudos comparativos entre ferramentas de persistência de dados nessa linguagem. Para a escolha dos *frameworks* é necessário um estudo exploratório em fóruns de desenvolvimento e sistemas de controle de versão para verificar quais são mais recorrentes nos seguintes tópicos:

- Tópicos de pesquisa;
- Itens encontrados;
- Perguntas;
- Notícias sobre tecnologia.

Através dos resultados obtidos da pesquisa em fóruns como por exemplo *Stackoverflow* e sistemas de controle de versão como o *GitHub*, podemos visualizar quais os mais populares e estão sendo mais usados dentro da comunidade de desenvolvedores e compará-los, para ser possível a criação um catálogo com os resultados da comparação.

4.2 Seleção dos atributos dos *frameworks* a serem comparadas e inseridas em um catálogo

Este passo consiste na escolha dos atributos que servem como guia de comparação entre os *frameworks*. Para se diferenciar da abordagem dos trabalhos descritos no Capítulo 3, o presente trabalho se propõe a analisar e comparar atributos anteriormente não considerados e que são fundamentais para auxiliar desenvolvedores a escolher qual o *framework* ideal para seu trabalho, ou oferecer manutenção ao código que implementa essa ferramenta de persistência. Os atributos são descritos um a um na Subseção 4.4, sendo inseridos em uma tabela com o intuito de ao final do trabalho servir de catálogo, com os dados obtidos da análise comparativa do presente trabalho.

4.3 Definição de uma escala de comparação entre os *frameworks*

Este passo consiste na definição de uma escala para avaliar a qualidade de uso dos *frameworks* em determinados casos. Com base nos atributos levantados, existem atributos qualitativos e quantitativos que precisam ser avaliados com esta escala para ter uma melhor descrição e clareza quando inseridos no catálogo. Os valores da escala consistem de 1 a 5

partindo da melhor atribuição possível que significa que o *framework* é fortemente recomendado para o cenário até a atribuição mais inferior possível a um determinado cenário, onde o *framework* não é recomendado para ser utilizado. Os valores da métrica consistem em: 1 para Ótimo; 2 para Muito bom; 3 para Bom; 4 para Fraco; 5 para Muito fraco.

4.4 Definição dos atributos de comparação qualitativos e quantitativos

Nesta Seção, são apresentados todos os atributos que servem como guia para a avaliação dos *frameworks* e estão agrupados em dois grupos, os critérios quantitativos e qualitativos. Logo abaixo são descritos os dois tipos principais de critérios com a descrição de cada atributo, que faz parte de um dos grupos. Dentro da maioria das comparações são contabilizados os passos para que o determinado cenário se realize, estes passos são ações que o desenvolvedor precisa desempenhar para que o objetivo seja alcançado, ou seja, quando são contabilizados os passos isso significa que foi contabilizado o número total de ações para que um objetivo se realize.

4.4.1 Critérios quantitativos

Nesta categoria de critério, são descritos os atributos que podem afetar a decisão de uso de um *framework* por desenvolvedor ou por uma empresa, pela quantidade de passos necessários para realizar uma ação, desempenho em tempo de resposta de determinada ferramenta em casos específicos e entre outros fatores que podem ser cruciais para o desenvolvimento correto e bem-sucedido do projeto de um banco de dados.

4.4.1.1 Desempenho em tempo execução

Neste fator, são avaliados o tempo de resposta em milissegundos (ms) das operações típicas do banco de dados, que consistem em criar, buscar, atualizar e deletar registros. Para cada *framework* são testadas 5 chamadas ao projeto de banco de dados (melhor detalhado na Subseção 4.5) e feito a média dos resultados de tempo obtidos para ter o menor comprometimento de respostas possível, visto que a primeira chamada do ORM ou ODM ao banco de dados é a que gasta mais tempo, por abrir a conexão com banco de dados e iniciar uma transação comercial. Para obter resultados mais imparciais possíveis e oferecer uma comparação mais ampla, além dos resultados em tempo de resposta de cada *framework*, será feita uma divisão entre as ferramentas de persistência que implementam apenas bancos de dados não relacional e outra somente com

ferramentas que implementam bancos de dados relacionais, estes podem implementar bancos de dados não relacionais mas no contexto do presente trabalho implementam apenas bancos que seguem a estratégia relacional.

4.4.1.2 Simplicidade de treino

Este atributo se refere ao custo necessário para treinar desenvolvedores, sejam eles trabalhando em projetos independentes de pequeno porte que possam ser pessoais ou desenvolvedores que trabalham para empresas que fabricam software. Neste atributo os *frameworks* são avaliados em duas circunstâncias:

1. Em projetos pequenos que possuem menos complexidade, relacionamentos e possibilidade de escalar;
2. Em projetos maiores com mais complexidade, com mais relacionamentos e possibilidade de escalar.

4.4.1.3 Custo de manutenção

Neste atributo é considerado o custo para realizar manutenções em um projeto de banco de dados que usa ORM e pode eventualmente mudar o ORM por uma necessidade do domínio, ou que necessite migrar para outro banco de dados para atender suas necessidades. Portanto, dois aspectos são considerados, são eles:

1. Quantidade de arquivos modificados ao alterar um campo da tabela;
2. Quantidade de arquivos ao trocar de banco de dados.

4.4.1.4 Bancos de dados suportados

Para este atributo, são contabilizados e comparados entre os *frameworks* a quantidade de bancos de dados suportados que podem ser usados para construir aplicações de software. Os *frameworks* que implementam apenas uma categoria de banco de dados são direcionados a um domínio mais específico de aplicações, como os que implementam apenas bancos relacionais são comumente usados em projetos de maior porte com diversos relacionamentos e necessidade de buscar valores em outras tabelas, já *frameworks* que oferecem suporte a apenas bases de dados não relacionais são mais usados em projetos de pequeno porte e de menor complexidade dado que sua configuração e treino são relativamente mais simples.

4.4.1.5 Tamanho da comunidade

Para avaliar o tamanho da comunidade de cada *framework* é necessária a divisão do atributo em duas subcategorias, são elas:

1. Tamanho da comunidade no *GitHub*;
2. Tamanho da comunidade no *Stack Overflow*.

A divisão é realizada para verificar além do tamanho da comunidade a proatividade de desenvolvedores em cada *framework*. Para o tamanho da comunidade no *GitHub* é preciso extraídos dados como número de *commits* no repositório central, o número de desenvolvedores que contribuíram para o repositório, a quantidade de estrelas fornecidas e o valor de *forks* do projeto. Para o tamanho da comunidade no *Stack Overflow*, o dado foi extraído do número de ocorrências de cada *framework*.

4.4.1.6 Modelos de dados aceitos

Semelhantemente à comparação de Bancos de dados suportados, os *frameworks* que implementam um modelo apenas de dado é mais indicado ou usado em contextos mais específicos. Na coleta de modelos de dados aceitos de cada ferramenta são encontrados *frameworks* que implementam apenas uma categoria de dado, relacional ou não relacional, mas também há ferramentas dentre as comparadas que implementam os dois tipos anteriores e até mais um tipo, o modelo de dado em grafo.

4.4.1.7 Padrões de persistência implementados

Neste atributo a quantidade de padrões de persistência implementados por cada *framework* são contabilizados. A quantidade de padrões pode indicar a quantidade de formas possíveis de trabalhar para armazenar ou acessar dados usando determinado *framework*, também é possível trabalhar com diferentes arquiteturas dependendo dos padrões implementados. Para os ORMs e ODMs que são objeto de estudo do presente trabalho, foi visualizado que os *frameworks* variam entre o número de padrões implementados, que vai de 1 a 3 categorias de padrões diferentes.

4.4.1.8 Início do projeto Vs. Última atualização

Uma maneira de medir a proatividade da comunidade que utiliza e implementa novos recursos aos *frameworks* além de postagens em fóruns e pesquisas, é o início do projeto e quando foi a última vez que ele recebeu alterações. É uma forma de indicar para os desenvolvedores se o *framework* em que eles implementam ou pensam em usar está recebendo atualizações no mesmo ritmo que a comunidade e o JavaScript se desenvolve.

4.4.2 Critérios qualitativos

Semelhantemente à escolha dos atributos quantitativos, os atributos qualitativos agrupados neste critério descritos nesta Subseção são pensados de forma que possam afetar a decisão de uso de um *framework* e garantam qualidade e sucesso para o projeto do banco de dados, mas que não podem apenas ser quantificados e generalizados entre desenvolvedores e projetos.

4.4.2.1 Curva de aprendizado

Como este é um atributo muito subjetivo, pode variar da experiência e entendimento de cada desenvolvedor, é estipulado a quantidade de passos necessários para conseguir realizar uma ação em determinados cenários que contam como subitens, para se poder obter resultados palpáveis desta característica. Os passos contabilizados consideram um cenário ideal sem erros de configuração prévios, ou erros durante o processo. Os cenários estabelecidos são os seguintes:

1. Quantidade de passos para criar primeira Entidade/Modelo no banco;
2. Quantidade de passos para criar primeira Chave Estrangeira/Relacionamento no banco;
3. Quantidade de passos para usar o *framework* de persistência;
4. Quantidade de argumentos passados ao *framework* para realizar operações no banco (Total).

4.4.2.2 Simplicidade de configuração

Semelhantemente ao atributo anterior, a simplicidade de configuração também pode ser um atributo subjetivo, mas de forma que a subjetividade da simplicidade não está no desenvolvedor, mas nas ferramentas que o mesmo usa para configura um projeto, como recursos de *hardware* e documentação disponível. Neste atributo também são estipulados cenários ideais

para que ações possam ser realizadas e a quantidade de passos necessários para a conclusão foi contabilizada. Os cenários levantados para este atributo são:

1. Quantidade de passos para criar conexão com o banco de dados;
2. Quantidade de arquivos necessários para criar conexão com o banco;
3. Quantidade de passos para criar conexões HTTP;
4. Quantidade de arquivos necessários para criar uma Entidade/Modelo.

4.4.2.3 Facilidade de escrita

Este atributo é referente a quão intuitiva, fácil de replicar e legível são as ações utilizadas ou chamadas pelo ORM. Para este atributo cada *framework* é submetido a uma análise de código onde também é considerado a quantidade de argumentos passados ao ORM em operações CRUD como na avaliação de Curva de aprendizado, e o nome de cada argumento/função para verificar o quão descritível é o método. Cada linguagem recebeu um valor de um 1 a 5 representando seu valor neste atributo, segundo a métrica estabelecida na Subseção 4.3

4.4.2.4 Estratégia no carregamento (loading) de dados

Cada *framework* de persistência de dados tem uma maneira padrão de carregar os dados para responder um usuário, dado que ele fez uma requisição no banco de dados e no caso deste trabalho são as APIs desenvolvidas para cada ORM/ODM. Existem dois tipos principais de carregamento de dados sendo possível definir em cada ferramenta de persistência qual estratégia utilizar nas requisições, as estratégias são:

1. *Eager Loading*: Significa que todas as entidades relacionadas são carregadas automaticamente na mesma consulta, ou seja, caso uma entidade se relacione com outra e o tipo do relacionamento é *one-to-many* (um para muitos), quando se consultar a entidade que permite a existência da outra caso ela exista, irá trazer não somente a entidade forte como todos os seus registros ou relacionamentos filhos. O ORM normalmente traz os dados da relação por meio de métodos *Join* entre as tabelas relacionadas;
2. *Lazy Loading*: Significa que as entidades relacionadas são carregadas sob demanda, ou seja, caso o usuário queira trazer entidades relacionadas será necessária outra cláusula ou método pra trazer a relação, por exemplo, no TypeORM existe a possibilidade de selecionar certos campos para serem retornados quando a função é chamada, com a cláusula *select* dentro de um método de busca.

4.4.2.5 Documentação

Para avaliar a qualidade da documentação de cada *framework*, recursos são estipulados para deverem estar presentes na documentação da ferramenta e são imprescindíveis para o entendimento e uso de qualquer ferramenta, seja o desenvolvedor um iniciante ou experiente na área, iniciante ou experiente em usar certo *framework*. Os recursos anteriormente citados são:

1. Quantidade de línguas que a documentação está disponibilizada;
2. Passos para criação de tabela/modelo disponível;
3. Passos para conexão com bancos de dados disponíveis;
4. Operações típicas de banco de dados documentadas (CRUD);
5. Exemplos de projetos.

4.5 Comparação dos *frameworks* de persistência selecionados

Neste passo é descrito como será realizada a comparação atributos descritos acima entre cada *framework*. Para isto, é necessário a criação de um projeto de banco de dados para todas as ferramentas de persistência, totalizando 6 projetos de banco de dados. A arquitetura adotada para todos os projetos é a de *Application Programming Interface* (API), atualmente uma das 3 categorias de API mais popular no desenvolvimento de software em *back-end*.

A escolha da API RESTful para ser o padrão arquitetural seguido em todos os projetos é dado pela arquitetura padronizada, que fornece dados do tipo *JavaScript Object Notation* (JSON) a partir de requisições HTTP e fornecimento de recursos a serviços externos para integração sem necessidade de conhecer as regras de negócio e detalhes do projeto.

Todas as APIs são independentes entre si, entretanto o domínio de aplicação implementado pelos 6 projetos é o mesmo, justamente para alinhar todos os *frameworks* e dar a mesma chance para obter resultados satisfatórios a todos, e não favorecer determinadas ferramentas, mudando o domínio da aplicação para que as favorecidas tenham melhor desempenho enviesando os dados da comparação deste trabalho. As APIs desenvolvidas consistem em projetos simples de gerenciamento de produtos de um mercado, onde é possível o cadastro de categorias e criar produtos e atrelá-los as categorias já existentes, onde uma categoria não pode ser repetida, por exemplo, criar uma categoria de alimentos e atrelar-lhe o produto tomate. Além da possibilidade do cadastro de categorias e produtos, também são contemplados as outras operações típicas de banco de dados, que consistem em buscar, atualizar e deletar. Os projetos são colocados

em um único repositório e armazenados dentro do *GitHub* de forma pública, disponível em <https://github.com/gustavocolombo/APIs-TCC>.

4.6 Preenchimento do catálogo com os resultados obtidos

Com a definição dos atributos e fazer a união deles em dois grupos maiores de critérios que guiam a análise comparativa e inseri-los no catálogo e também com a implementação de todas as APIs, os *frameworks* são independentes um do outro em cada um dos atributos. Após a finalização de cada teste em cada atributo, o resultado obtido da avaliação do *framework* é adicionado a linha correspondente da ferramenta e coluna correspondente ao atributo recém testado.

4.7 Análise dos resultados obtidos

Para a análise dos resultados obtidos, é necessário o preenchimento do catálogo proposto por este trabalho para que cada *framework* seja avaliado de acordo com cenários de uso estipulados anteriormente, e também comparado entre outros *frameworks*, para inferir a qualidade e recomendação de uso da ferramenta de persistência em certos cenários e domínios.

5 EXPERIMENTOS E RESULTADOS

Neste capítulo são apresentados os resultados obtidos a partir da metodologia adotada no presente trabalho.

5.1 Como os dados foram coletados

Para que os resultados preliminares deste trabalho fossem obtidos, tabelas e quadros de resultados fossem construídos foi necessário como dito anteriormente na Seção 4.5 a construção de 6 APIs REST para submeter os *frameworks* de persistência a cada um dos atributos estabelecidos, seja um atributo quantitativo ou qualitativo. A escolha de arquitetura de desenvolvimento das APIs ser a REST é pela popularidade atual na comunidade de desenvolvedores, visto que os conceitos para implementar uma API de REST já são conhecidos e usados em outras metodologias e arquiteturas, consiste em 5 princípios que são:

1. Interface uniforme
2. Separação de cliente e servidor
3. Armazenamento em *cache*
4. Aplicação *Stateless*
5. Separação em camadas

Utilizar esse tipo de arquitetura é um facilitador para outros estudiosos e desenvolvedores da área que caso se interessarem, podem dar prosseguimento ao presente trabalho com uma curva de aprendizado menor do que se fosse utilizado outro tipo de metologia, como GraphQL ou gRPC. Como os dados foram coletados difere de atributo por atributo pela sua peculiaridade, por exemplo, como a coleta de dados do desempenho dos *frameworks* em tempo de execução foi realizada, foram necessárias 5 requisições em cada uma das rotas responsáveis por chamar métodos que fazer operações típicas de banco de dados nos dados.

5.1.1 Dificuldades encontradas

Para que as APIs fossem desenvolvidas, foi necessário um estudo mais aprofundado em determinados *frameworks* como o *MikroORM* e o *Typegoose*, visto que são ferramentas ainda pouco maduras e não tem uma comunidade tão ativa quanto outros *frameworks* populares como o *TypeORM* ou *Sequelize*. Outro fator que contribuiu para o estudo mais profundo e necessário destes ORMs foi a falta de exemplos ou a falta de recursos disponíveis na própria documentação

da ferramenta, o que dificultou que as APIs fossem desenvolvidas mais rapidamente quanto as que utilizam *TypeORM* e Prisma.

Apesar de serem mais populares, maduras e o escritor já possuir pleno conhecimento das ferramentas, a documentação é um ponto importante, igualmente os exemplos da comunidade disponibilizadas em sistemas de versão de código como o *GitHub* que ajudam principiantes no uso da ferramenta a usá-la. Sem ter nenhum conhecimento prévio do *Sequelize*, apenas pesquisando a partir destes últimos dois atributos mencionados, foi possível que a API que utiliza *Sequelize* fosse rapidamente desenvolvida, diferente do *MikroORM* e *Typegoose* que possuem limitações nestes pontos.

5.2 Seleção dos *frameworks* de persistência

Nesta Seção será descrita como se deu a validação e escolha da linguagem de programação JavaScript e dos *frameworks* baseados na linguagem da análise proposta pelo trabalho.

5.2.1 Validação de uso do JavaScript

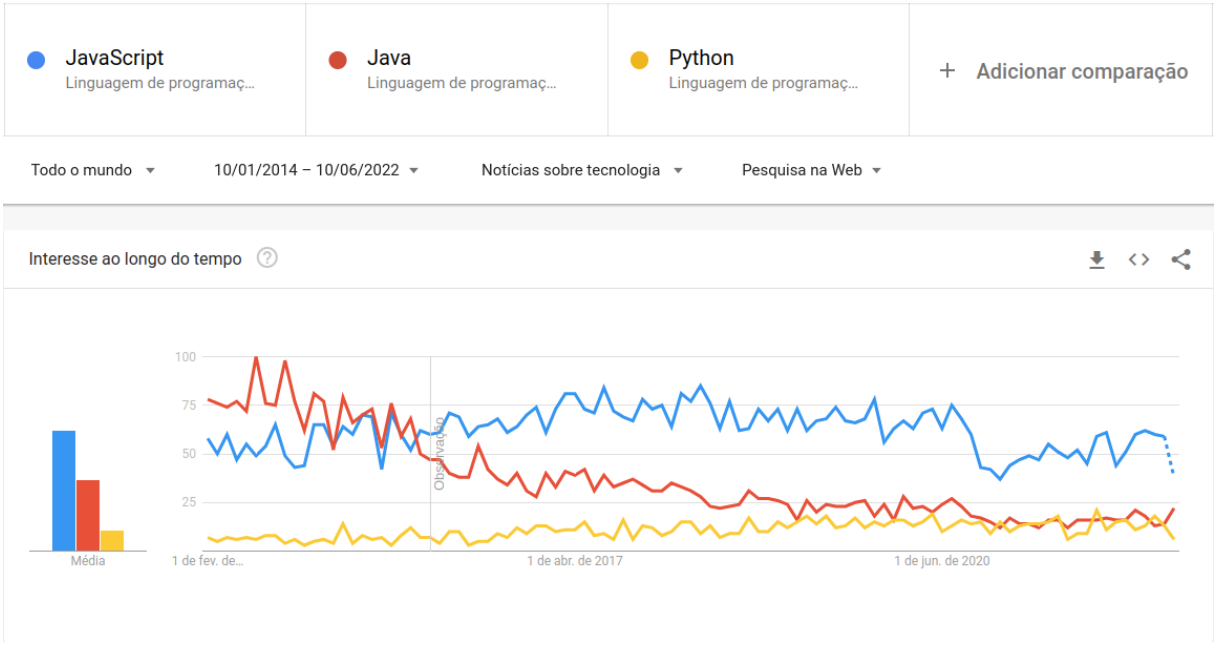
Para selecionar as ferramentas de persistência de dados, primeiramente foi necessário conduzir uma pesquisa para validar o dado de que JavaScript é a linguagem de programação mais utilizada no mundo desde 2014, segundo Forsgren Kalliamvakou (2021).

Para isso, foi conduzido um estudo em fóruns de tecnologia para verificar a ocorrência do JavaScript em comparação com outras tecnologias abordadas no *GitHub Octoverse* ao longo dos anos. O fórum utilizado na pesquisa para visualizar a ocorrência de resultados das linguagens de programação foi o *Stack Overflow* e o Google Trends para medir o nível de popularidade e/ou número de aparições em tópicos de pesquisa por tecnologia, seguem abaixo os resultados obtidos:

5.2.2 Seleção dos frameworks JavaScript

A partir da validação de que JavaScript é a linguagem mais usada e popular, foi necessário escolher quais ferramentas de persistência de dados seriam analisadas e comparadas neste trabalho, e para isto, também foi realizada uma pesquisa para verificar quais são as mais pesquisadas e recorrentes no meio de desenvolvimento.

Figura 2 – Comparação entre as três linguagens de programação mais populares no Google Trends



Fonte: Google Trends.

Tabela 1 – Comparação entre ocorrências das linguagens de programação em *Stack Overflow*

Linguagens de programação	Ocorrências no Stack Overflow
PHP	27.307
JavaScript	27.04
Java	14.570
C#	14.025
Python	9.178
C	4.954
C++	2.399
Ruby	2.078
Shell	1.366
TypeScript	1.230

Fonte: elaborada pelo autor.

Semelhantemente ao estudo realizado na Seção anterior, o *Stack Overflow* foi utilizado para verificar o número de resultados a partir da pesquisa de um *framework* e se o mesmo possuía um número de resultados significantes, era selecionado para realizar a comparação. Como validação do resultado obtido da pesquisa no *Stack Overflow*, foi verificado no *GitHub* o número resultados para as seguintes categorias:

– Repositórios
– Código

– *Commits*

Após a consolidação do *framework* nos resultados anteriores, foi necessário verificar se a ferramenta atendia aos padrões de ORM ou ODM. Dentre as consolidadas em um escopo inicial, apenas uma não contemplou o padrão ORM ou ODM, portanto não foi abordada neste trabalho, tampouco foi abordada em um dos trabalhos descritos em Capítulo 3. A ferramenta é o Knex que consiste em um *query builder* para JavaScript. Os *frameworks* de persistência selecionados para comparação são:

1. *TypeORM*
2. *MikroORM*
3. *Mongoose*
4. *Typegoose*
5. *Sequelize*
6. Prisma

5.3 Definição de uma escala para a comparação entre os *frameworks*

A medida que a análise dos atributos e seus subitens ia progredindo foi necessário estabelecer uma escala para tornar o catálogo mais descritivo e também era necessária uma maneira de indicar a qualidade de determinado *framework* para um cenário de uso, como nos subitens de custo de treino que consiste na qualidade do *framework* em situações de:

1. Projetos pequenos, complexidade reduzida, escalabilidade baixa e poucos relacionamentos;
2. Projetos grandes, maior possibilidade de escalar, complexo e com muitos relacionamentos.

Então, como maneira de indicar a qualidade de uso a partir de outras características (que estão presentes no catálogo) como, por exemplo, quantidade de passos necessários para criar uma entidade, foi estabelecido um conjunto de 5 valores (já descritos na Subseção 4.3) que indicam como o *framework* se comporta a determinadas situações.

5.4 Definição dos atributos de comparação qualitativos e quantitativos

Como passo anterior e necessário para a análise comparativa entre os *frameworks*, foi preciso o estabelecimento de atributos que avaliassem o desempenho da ferramenta em questões de desempenho, mas também critérios que avaliassem a qualidade de uso de uma ferramenta em determinados cenários pensando na perspectiva do desenvolvedor.

Para isso, foram estabelecidos atributos semelhantes aos trabalhos mencionados em Capítulo 3, mas também atributos quantitativos e qualitativos adicionais agrupados em dois grupos maiores, sendo eles o grupo de critérios quantitativos e critérios qualitativos, para contribuição do presente trabalho com a comunidade. Logo abaixo são descritos como os resultados preliminares foram obtidos a partir da análise realizada de cada atributo quantitativo e qualitativo de cada *framework* (apresentados em Capítulo 3) e quais foram os resultados obtidos da análise.

5.4.1 Critérios quantitativos

Nesta Seção são descritos todos os atributos quantitativos escolhidos para realizar os testes em cada ferramente de persistência de dados, sendo agrupados em um dos dois grupos de critérios maiores, o de critério quantitativo. Os atributos agrupados neste critério são os que podem ser quantificados e ter sua qualidade avaliada em determinados cenários por meio de resultados em quantidade de passos necessários para realizar uma ação, desempenho em tempo de resposta de determinada ferramenta em casos específicos, etc.

Para a obtenção dos resultados dos atributos presentes neste grupo de critério que visam tempo de resposta em execução ou passos, foi utilizado um notebook Asus Vivobook 15, com sistema operacional Ubuntu 20.04, com 8GB de RAM e um processador Intel i7 de 11ª geração. As operações que visam tempo de resposta em execução de cada *framework* foram executadas 5 vezes para o menor enviesamento dos dados ou a menor disparidade entre os *frameworks* na primeira conexão com o banco de dados.

5.4.1.1 Desempenho em tempo execução

Para realizar os testes em cada *framework* foi preciso que cada projeto do banco de dados fosse finalizado com as mesmas funcionalidades no mesmo domínio. Este atributo foi dividido em outros 4 atributos que representam as operações típicas do banco de dados, conforme mencionado na Subseção 4.4.1.1.

Para cada subitem do atributo foram executadas 5 requisições para o menor enviesamento de dados possível em cada *framework*. Visando o tempo de resposta de cada *framework* em tempo de execução neste critério, para nortear os leitores que buscam maior velocidade na hora de execução das requisições em seus bancos de dados, foi realizada a coleta do tempo gasto em cada uma das requisições em todos os *frameworks*, e realizada a média dos 5 resultados para

Tabela 2 – Média do tempo de execução dos seis *frameworks*

Frameworks utilizados	Tempo de criação (ms)	Tempo de busca (ms)	Tempo de atualização (ms)	Tempo de deleção (ms)
TypeORM	4.294	1.793	9.121	13.451
PrismaORM	4.681	1.389	8.383	14.593
Mongoose	0.248	65.145	67.639	66.782
Sequelize	12.736	1.860	14.002	3.004
MikroORM	13.063	2.801	7.093	13.972
Typegoose	64.55	64.727	62.783	66.368

Fonte: elaborada pelo autor.

cada uma das ferramentas. Para medir o tempo gasto em cada uma das requisições em milissegundos foi utilizado o método *console.time()* do JavaScript, um colocado antes da instanciação do *service* (arquivo que implementa a regra de negócio) e um depois, para medir o tempo de envio dos dados e resposta obtida. A consulta realizada 5 vezes para obter o tempo médio do tempo de resposta em todos os *frameworks*, foi a requisição de criar categorias, visto que é a primeira requisição em todas as APIs que retorna um valor para o usuário. Abaixo segue um exemplo de como foi montada uma rota no *TypeORM* chamando o *service* e como foi colocado o medidor de tempo do JavaScript:

Figura 3 – Requisição de criação e medição de tempo no *TypeORM*

```
categoriesRoutes.post('/create', async (request, response) => {
  const { name } = request.body;

  const service = new CreateCategoryService();

  console.time('typeormCreate');

  const newCategory = await service.execute({ name });

  console.timeEnd('typeormCreate');

  return response.json(newCategory);
});

```

Fonte: elaborada pelo autor.

A Tabela 2 é mostrado o resultado da média das 5 requisições em todos os *frameworks* de persistência.

Tabela 3 – Média do tempo de execução dos quatro *frameworks* relacionais

Frameworks de persistência relacionais	Tempo de criação (ms)	Tempo de busca (ms)	Tempo de atualização (ms)	Tempo de deleção (ms)
TypeORM	4.294	1.793	9.121	13.451
PrismaORM	4.681	1.389	8.383	14.593
Sequelize	12.736	1.860	14.002	3.004
MikroORM	13.063	2.801	7.093	13.972

Fonte: elaborada pelo autor.

Tabela 4 – Média do tempo de execução dos dois *frameworks* não relacionais

Frameworks de persistência não-relacionais	Tempo de criação (ms)	Tempo de busca (ms)	Tempo de atualização (ms)	Tempo de deleção (ms)
Mongoose	0.248	65.145	67.639	66.782
Typegoose	64.55	64.727	62.783	66.368

Fonte: elaborada pelo autor.

Como dito anteriormente em subseção 4.4.1.1, para obter uma comparação mais imparcial possível e oferecer ao leitor uma quantidade de comparações mais abrangentes que possam atender seus casos, abaixo seguem as tabelas referentes a comparação em tempo de execução dos *frameworks* de persistência que implementam bancos de dados relacionais (Tabela 3) e bancos de dados não relacionais (Tabela 4), respectivamente:

5.4.1.2 Simplicidade de treino

Para realizar a análise e preenchimento do catálogo neste atributo com cada *framework*, foi consultado a documentação oficial de cada ferramenta, os padrões de persistência implementados e a média do tempo de resposta no quesito desempenho em execução, para indicar qual o valor da qualidade de uso de cada *framework* em dois atributos que são subitens do atributo de Custo de treino, sendo:

1. Em projetos pequenos que possuem menos complexidade, relacionamentos e possibilidade de escalar;
2. Em projetos maiores com mais complexidade, com mais relacionamentos e possibilidade de escalar.

Os resultados obtidos a partir da análise acima são apresentados em Quadro 2.

5.4.1.3 Custo de manutenção

Para realizar a análise de cada *framework* neste atributo foram contabilizados a quantidade de arquivos e passos necessários para realizar mudanças no código que implementa

Quadro 2 – Custo de treino dos seis *frameworks*

Frameworks de persistência	Simplicidade de treino	Em projetos pequenos que possuem menos complexidade, relacionamentos e possibilidade de escalar	Em projetos maiores com mais complexidade, com mais relacionamentos e possibilidade de escalar
TypeORM	2 - Muito bom	3 - Bom	**1 - Ótima**
Prisma Client	**1 - Ótimo**	**1 - Ótima**	**1 - Ótima**
Mongoose	3 - Bom	**1 - Ótima**	4 - Fraco
Sequelize	2 - Muito bom	3 - Bom	2 - Muito bom
MikroORM	2 - Muito bom	2 - Muito bom	3 - Bom
Typegoose	3 - Bom	2 - Muito bom	4 - Fraco

Fonte: elaborada pelo autor.

Quadro 3 – Custo de manutenção dos seis *frameworks*

Frameworks de persistência	Custo de manutenção	Quantidade de arquivos modificados ao alterar um campo da tabela	Quantidade de arquivos ao trocar de banco de dados
TypeORM	2 - Muito bom	2	2 - 3
Prisma Client	**1 - Ótimo**	**1**	**1**
Mongoose	**1 - Ótimo**	**1**	0
Sequelize	2 - Muito bom	1 - 2	2
MikroORM	2 - Muito bom	1 - 2	**1**
Typegoose	**1 - Ótimo**	**1**	0

Fonte: elaborada pelo autor.

a ferramenta. A partir da conclusão de cada projeto do banco de dados foi estabelecido em um cenário ideal, sem erros, a quantidade de passos ou arquivos necessários para realizar as seguintes ações:

1. Quantidade de arquivos modificados ao alterar um campo da tabela;
2. Quantidade de arquivos ao trocar de banco de dados.

Para a contagem dos aspectos acima, foi consultada a documentação oficial da ferramenta e o código de cada projeto que implementava os *frameworks*, visto que é um domínio pensado para este trabalho, portanto, os aspectos acima foram estimados a partir da estrutura disposta dos projetos tendo como guia a documentação de cada ferramenta. A Quadro 3 mostra os resultados obtidos da análise deste atributo.

5.4.1.4 Bancos de dados suportados

Para a análise e comparação deste atributo, a documentação oficial de cada *framework* e também a pesquisa de palavras-chave no *GitHub* foram consultadas para reunir a

Quadro 4 – Bancos de dados suportados pelos seis *frameworks*

Frameworks de persistência	Banco de dados suportados	Total de bancos suportados por framework
TypeORM	MySQL, MariaDB, PostgreSQL, CockroachDB, SQLite, Microsoft SQL Server, Oracle, SAP Hana, sql.js, MongoDB	**10**
Prisma Client	PostgreSQL, MySQL, Maria DB, SQLite, AWS Aurora, Microsoft SQL Server, MongoDB, Azure SQL	8
Mongoose	MongoDB	1
Sequelize	PostgreSQL, MySQL, MariaDB, SQLite, MSSQL	5
MikroORM	MongoDB, MySQL, MariaDB, PostgreSQL, SQLite databases	5
Typegoose	MongoDB	1

Fonte: elaborada pelo autor.

quantidade de banco de dados suportados por cada ferramenta de persistência. A pesquisa por termos específicos, *framework* + categoria de banco de dados foi realizada para validação da documentação oficial, pois certas documentações não deixam claras quais bancos de dados suportados nem a quantidade, então existia a possibilidade da ferramenta abordar mais bancos de dados do que mostrado em seu guia. O Quadro 4 apresenta os resultados obtidos a partir da pesquisa mencionada acima.

5.4.1.5 Tamanho da comunidade

Para este atributo, foram reunidos dados provenientes das plataformas *GitHub* e *Stack Overflow* e a partir da reunião dos dados foi preciso a divisão do atributo em dois outros atributos filhos, respectivos ao tamanho da comunidade em cada uma das plataformas. Os dados extraídos do *GitHub* foram os seguintes:

1. Número de *commits* no repositório central;
2. Número de desenvolvedores que contribuíram (*contributors*) para o repositório;
3. Número de estrelas (*stars*) fornecidas;
4. Número de *forks* do projeto.

Para a plataforma do *Stack Overflow* o dado extraído foi o número de ocorrências a

Quadro 5 – Tamanho da comunidade dos seis *frameworks*

Frameworks de persistência	Tamanho da comunidade (Stack Overflow)	Tamanho da comunidade (Github)
TypeORM	7.830 resultados (até o dia 1/05/2022)	4.985 commits, 5.100 Forks, 27.800 stars, 814 contributors
Prisma Client	3.460 resultados (até o dia 1/05/2022)	7.827 commits, 771 forks, 21.800, 161 contributors
Mongoose	10.9167 resultados (até o dia 1/05/2022)	15.949 commits, 3.400 forks, 24.100, 699 contributors
Sequelize	**20.444 resultados** (até o dia 1/05/2022)	**9.267 commits**, **4.000 forks**, **26.000 stars**, **1.008 contributors**
MikroORM	96 resultados (até o dia 1/05/2022)	2.951 commits, 281 forks, 4.400 stars, 27 contributors
Typegoose	298 resultados (até o dia 1/05/2022)	1.702 comits, 120 forks, 1.700 stars, 60 contributors

Fonte: elaborada pelo autor.

partir da pesquisa do nome do *framework*. Os resultados obtidos a partir desta pesquisa, até o dia 01/05/2022, são mostrados no Quadro 5.

5.4.1.6 Modelos de dados aceitos

Para a realização da análise deste atributo, foram buscadas as informações necessárias na documentação oficial de cada *framework* e também pesquisado na plataforma *GitHub* palavras-chave para validação de informações obtidas da documentação como, por exemplo:

1. Nome do *framework* + *relational* ou SQL;
2. Nome do *framework* + *non-relational* ou NoSQL;
3. Nome do *framework* + Neo4j ou *graph*;

Os resultados obtidos a partir da pesquisa acima são detalhados no Quadro 6.

5.4.1.7 Padrões de persistência implementados

Para reunir as informações necessárias para a análise deste atributo, foram buscadas informações na documentação oficial de cada *framework*. Para as ferramentas que não possuem a

Quadro 6 – Modelos de dados aceitos dos seis *frameworks*

Frameworks de persistência	Modelo de dados aceitos
TypeORM	**SQL**, **NoSQL**, **Neo4j**
Prisma Client	SQL, NoSQL
Mongoose	NoSQL, SQL(com plugin), Neo4j(mongo4j)
Sequelize	SQL
MikroORM	SQL, NoSQL
Typegoose	NoSQL

Fonte: elaborada pelo autor.

Quadro 7 – Padrões de persistência implementados pelos seis *frameworks*

Frameworks de persistência	Padrão de persistência
TypeORM	Data Mapper/Active record
Prisma Client	Data Mapper
Mongoose	Active record
Sequelize	Active record
MikroORM	Identity Map/Unit of work/Data mapper
Typegoose	Active record

Fonte: elaborada pelo autor.

informação de qual ou quais padrões de persistência elas implementam, buscas foram realizadas em fóruns de tecnologia como o *Medium*. Os resultados obtidos a partir da pesquisa são apresentados no Quadro 7.

5.4.1.8 Início do projeto Vs. Última atualização

Para realizar a análise deste atributo os dados de início do projeto e última atualização no projeto foram coletados na plataforma *GitHub*. Os dados foram coletados apenas neste sistema de versionamento de código porque todos os repositórios são *open-source* e estão disponíveis nesta plataforma. Os resultados obtidos a partir da pesquisa estão presentes no Quadro 8, visto que a reunião destes dados ocorreu no dia 01/05/2022.

5.4.2 Critérios qualitativos

Nesta Seção são descritos todos os atributos qualitativos escolhidos para realizar os testes em cada ferramenta de persistência de dados, sendo agrupados em um dos dois grupos

Quadro 8 – Início do projeto Vs. Última atualização dos seis *frameworks*

Frameworks de persistência	Início do projeto	Última atualização
TypeORM	21/02/2016	17/03/2022
Prisma Client	2019 (prod: 21/04/2021)	01/04/2022
Mongoose	06/04/2010	04/04/2022
Sequelize	2006	05/04/2022
MikroORM	15/03/2018	24/04/2022
Typegoose	17/03/2017	21/04/2021

Fonte: elaborada pelo autor.

de critérios maiores, o de critério qualitativo. Os atributos agrupados neste critério, são os atributos que mesmo que testes sejam realizados para analisar sua qualidade em certos cenários, não podem ser generalizados entre projetos e desenvolvedores, visto que cada um tem sua peculiaridade.

5.4.2.1 Curva de aprendizado

Como não é possível padronizar um nível de dificuldade que se aplique a todos os desenvolvedores nesta característica, pois como dito na Subseção 4.4.2.1, esse é um atributo que pode variar de desenvolvedor para desenvolvedor, tendo em vista que diferentes desenvolvedores possuem diferentes visões e experiências.

Para obter resultados que não definam um padrão para todos os desenvolvedores e projetos, foram estipulados atributos filhos (subitens deste atributo) que podem servir como uma resposta palpável a curva de aprendizado. Os subitens do atributo foram pensados em um mundo ideal sem erros de configuração ou falta de recursos para usar o *framework* sido mostrados na Subseção 4.4.2.1. O resultado obtido a partir da contagem dos processos que caracterizam os subitens do atributo são expostos na **??**.

Para obter dados mais concretos sobre este atributo, os subitens foram divididos em mais 4 atributos, que representam as operações típicas do banco de dados, (CRUD). Logo, são contabilizados os passos necessários para realizar às quatro ações típicas de manipulação de dados e apresentados na Tabela 5.

5.4.2.2 Simplicidade de configuração

Este atributo igualmente ao anterior é subjetivo, portanto não é possível indicar ou padronizar uma simplicidade, ou dificuldade de configuração para todos os desenvolvedores. A

Tabela 5 – Curva de aprendizado dos seis *frameworks* em operações CRUD

Frameworks de persistência	Criar	Buscar	Atualizar	Deletar	Quantidade de argumentos passados ao framework para realizar operações no banco (Total)
TypeORM	2	1	2	2	7
PrismaORM	1	1	1	1	4
Mongoose	2-4	2-4	2-4	2-4	8-16
Sequelize	2	2	2	2	8
MikroORM	3	2-3	3	3	11-12
Typegoose	2	2-4	2-4	2-4	8-12

Fonte: elaborada pelo autor.

Tabela 6 – Simplicidade de configuração dos seis *frameworks*

Frameworks de persistência	Simplicidade de configuração	Quantidade de passos para conectar com o banco de dados	Quantidade de arquivos necessários para criar conexão com o banco de dados	Quantidade de passos para criar conexões HTTP	Quantidade de arquivos necessários para criar uma Entidade/Modelo
TypeORM	2 - Muito bom	4	2	2	3
PrismaORM	1- Ótimo	3	2	2	1
Mongoose	3 - Bom	3	1	2	2
Sequelize	3 - Bom	3	3	2	3
MikroORM	3 - Bom	3	2	2	2
Typegoose	3 - Bom	3	3	2	2

Fonte: elaborada pelo autor.

subjetividade deste atributo reside nas definições de arquitetura do seu *hardware* e/ou software, externo ao desenvolvedor. Semelhantemente ao aspecto anterior, o atributo foi dividido em 4 subitens que caracterizam outros atributos filhos, para auxiliar a encontrar resultados mais descritivos que possam alcançáveis neste contexto. A Tabela 6 descreve os dados obtidos a partir da divisão e análise da característica em todos os *frameworks*.

5.4.2.3 Facilidade de escrita

Para realizar a análise deste atributo, foi preciso que todos os projetos de banco de dados estivessem concluídos para analisar como os ORMs ou ODMs realizavam operações CRUD no banco de dados. Como este atributo pode variar conforme a experiência do desenvolvedor, foram considerados os seguintes aspectos para a avaliação dos *frameworks* neste atributo:

1. Quantidade de argumentos passados ao ORM em operações CRUD;
2. Nome de cada argumento/função para verificar o quão descritível é o método.

A partir da análise dos trechos de código de cada *framework*, os seguintes dados foram obtidos sendo apresentados no Quadro 9, com a divisão do atributo em um adicional, que representa a legibilidade da ferramenta. Cada *framework* também teve sua qualidade de indicada

Quadro 9 – Facilidade de escrita dos seis *frameworks*

Frameworks de persistência	Facilidade de escrita	Legibilidade (o quão ação é descritiva)
TypeORM	2 - Muito bom	**1 - Ótimo**
Prisma Client	2 - Muito bom	**1 - Ótimo**
Mongoose	2 - Muito bom	2 - Muito bom
Sequelize	2 - Muito bom	2 - Muito bom
MikroORM	3 - Bom	3 - Bom
Typegoose	2 - Muito bom	2 - Muito bom

Fonte: elaborada pelo autor.

Quadro 10 – Estratégias no carregamento de dados dos seis *frameworks*

Frameworks de persistência	Eager Loading	Lazy Loading
TypeORM	Sim	Sim
Prisma Client	Sim	Sim
Mongoose	Sim	Sim
Sequelize	Sim	Sim
MikroORM	Sim	Sim
Typegoose	Sim	Sim

Fonte: elaborada pelo autor.

pelas métricas mencionadas na Subseção 4.3.

5.4.2.4 Estratégia no carregamento (loading) de dados

Apesar de ser um atributo determinante no tempo de resposta na execução de operações típicas no banco de dados, é um fator que não pode ser generalizado em todas as consultas e o tempo de resposta pode e vai ser diferente entre desenvolvedores e suas máquinas. Então foi consultada a documentação oficial de cada *framework* de persistência para validar qual tipo de estratégia de carregamento de dados é disponibilizada por cada ferramenta, assim é possível saber qual *framework* implementa determinadas estratégias para usar em diferentes contextos e situações no desenvolvimento de software.

No Quadro 10 podemos visualizar que todos os *frameworks* de persistência implementam os dois tipos de carregamento de dados, ou seja, é possível implementar em consultas do banco de dados as duas estratégias, mas qual implementar depende do contexto e regras de negócio da consulta. Também consultando a documentação oficial de todas as ferramentas, é possível visualizar que todos os *frameworks* por padrão implementam o *Lazy Loading*, e com a adição de cláusulas que funcionam como *joins* em consultas, podemos implementar a estratégia de *Eager Loading*. As consultas realizadas nas APIs desenvolvidas neste trabalho e tempo de resposta obtidos em Tabela 2 foram somente com consultas do tipo *Lazy Loading*.

Tabela 7 – Avaliação da documentação dos seis *frameworks*

Frameworks de persistência	Línguas disponibilizadas (Quantidade)	Passos para criação de tabela/modelo disponível	Passos para conexão com bancos de dados disponíveis	Operações típicas de banco de dados documentadas (CRUD)	Exemplos de projetos
TypeORM	2	1	2	2	7
PrismaORM	1	1	1	1	4
Mongoose	2-4	2-4	2-4	2-4	8-16
Sequelize	2	2	2	2	8
MikroORM	3	2-3	3	3	11-12
Typegoose	2	2-4	2-4	2-4	8-12

Fonte: elaborada pelo autor.

5.4.2.5 Documentação

Para avaliar a qualidade da documentação de cada *framework*, todos os guias disponíveis e oficiais das ferramentas avaliadas neste trabalho. O atributo principal foi dividido em outros 5 atributos filhos com o intuito de alcançar valores concretos e descritivos para serem inseridos no catálogo principal. Os resultados obtidos a partir da análise das documentações é apresentado no Tabela 7.

5.5 Catálogo com os resultados obtidos

Nesta seção, está disponibilizado o link com todos os resultados obtidos no presente trabalho reunidos em apenas um local em formato de tabela, com o intuito de servir como um catálogo para a comunidade industrial de desenvolvimento de software e desenvolvedores autônomos. A tabela disponibilizada no link reúne todos os dados anteriores disponibilizados neste trabalho em tabelas menores. O link consistem em: <https://docs.google.com/spreadsheets/d/1sRU7mi_TJDuUeOZNt2pRyugSzjGADbviCyTsnbV_mdg/edit?usp=sharing>.

5.6 Ameaças a validade

Como ameaças a validade deste trabalho, existe a não generalização ou padronização de resultados obtidos para todos os projetos e desenvolvedores que utilizam *frameworks* de persistência de dados em JavaScript, visto que os dados obtidos no Capítulo 5 foram obtidos a partir de uma única visão de um desenvolvedor (o escritor), mesmo existindo mecanismos ou técnicas utilizadas na obtenção dos dados para ter o menor enviesamento de resultados e visando cenários ideais para execuções de tarefas, existem outros desenvolvedores com visões diferentes que poderiam ser abordadas, além também do poder computacional disponível para

cada desenvolvedor, que executando o código disponibilizado de cada *framework* poderia e é bem provável ter um resultado diferente do apresentado. Por exemplo, os resultados obtidos neste trabalho foram obtidos a partir de um único *hardware* e poderia ter resultados diferentes caso as APIs fossem testadas em outro dispositivo; As especificações do *hardware* usado nos testes já foi mencionado na Subseção 5.4.1.

Também é valido mencionar nesta seção como ameaça a validade o processo de aprender e dominar certos *frameworks* os quais, até o antes da escrita do presente trabalho, não tinham sido utilizados pelo desenvolvedor/escritor. Dada a situação anterior, há implementações dos *frameworks* aprendidos ao longo do desenvolvimento deste trabalho não terem sido a "melhor"implementação ou mais performática para a situação abordada no presente trabalho, o que nos leva mais uma vez ao convite de desenvolvedores e acadêmicos a replicarem o experimento realizado para consolidar os dados obtidos.

Também se faz necessária a colocação nessa Seção o número de requisições feitas em todas as requisições de manipulações típicas de banco de dados (criar, pesquisar, atualizar, deletar), que como dito anteriormente, foram 5 requisições para tentar obter o menor enviesamento de dados e disparidade entre os *frameworks*. É possível que esse número de requisições feitas de forma local a API não se comporte igual a uma API que está no ar, ou seja, disponível para inúmeros usuários fazerem requisições simultaneamente.

6 CONCLUSÃO E TRABALHOS FUTUROS

Com base nos resultados obtidos a partir dos experimentos mostrados no Capítulo 5, mais especificamente em Quadro 5, Quadro 8 e seção 5.2, podemos consolidar o quão grande é a comunidade de desenvolvedores que utilizam o JavaScript nos últimos anos e também podemos inferir que essa comunidade não irá parar de crescer, tornando o presente trabalho válido no cenário de desenvolvimento industrial e acadêmico, no Brasil por exemplo, de 2020 a 2021 a comunidade de desenvolvedores aumentou em 40% e a comunidade sul-americana cresceu em 1%, sendo a comunidade que mais cresceu dentre todas as outras do mundo, mesmo representando apenas 5.9% de desenvolvedores do mundo segundo Forsgren Kalliamvakou (2021).

A partir dos dados obtidos da condução do experimento, podemos visualizar e inferir não haver um *framework* de persistência que se sobressaia sobre todos os outros, cada *framework* teve um desempenho melhor em relação aos outros em determinadas operações no banco de dados, mas não houve um que se saiu melhor em todos os atributos e critérios, sejam quantitativos e/ou qualitativos. Então, com base na afirmação anterior e nos próprios resultados obtidos, para se obter o melhor resultado no projeto de software não somente em tempo de execução, que seria necessário usar a maioria dos *frameworks* em um único projeto (oque é praticamente impossível atualmente no contexto de desenvolvimento de software), mas em critérios quantitativos e qualitativos como um todo, seria necessário usar a maioria das ferramentas de persistência ou todas para ter o melhor resultado em um sistema, ou seja, atualmente não é possível utilizar essa estratégia em aplicações que possuam apenas um banco de dados ou não sigam uma arquitetura de microsserviços, então se faz necessária a escolha de um *framework* de persistência que satisfaça os requisitos do sistema e das partes interessadas, que seja de fácil manutenção e entre outros fatores que garantem o sucesso do projeto.

Também podemos concretizar que a partir dos experimentos do presente trabalho as operações típicas de banco de dados de buscar, atualizar e deletar registros são melhores em tempo de resposta ao usuário quando feitas em ferramentas de persistência que usam bancos de dados relacionais em relação a bancos de dados não relacionais, visto que essas operações são mais performáticas em bancos de dados relacionais dada a estruturação de tabelas e suas relações, uma vez que esse tipo de banco de dados é mais direcionado a sistemas que possuem regras de negócio complexas e exigem mais rapidez em operações desse tipo.

Para realização de trabalhos futuros é válida a criação de testes de carga para aplicar

nas APIs desenvolvidas no presente trabalho, com o intuito e oferecer mais segurança em relação aos resultados obtidos, principalmente ao desempenho de cada *framework* de persistência no quesito tempo de resposta em operações típicas de banco de dados. Com a implementação dos testes de carga em cada uma das APIs também podemos obter novos dados e adicioná-los ao catálogo de atributos, como, por exemplo, a escalabilidade de sistemas que implementam determinado *framework* de persistência.

REFERÊNCIAS

BARMPIS, K.; KOLOVOS, D. S. Comparative analysis of data persistence technologies for large-scale models. In: **Proceedings of the 2012 Extreme Modeling Workshop**. [S.l.: s.n.], 2012. p. 33–38.

BELSON, B.; HOLDSWORTH, J.; XIANG, W.; PHILIPPA, B. A survey of asynchronous programming using coroutines in the internet of things and embedded systems. **ACM Transactions on Embedded Computing Systems (TECS)**, ACM New York, NY, USA, v. 18, n. 3, p. 1–21, 2019.

CHEN, T.-H.; SHANG, W.; YANG, J.; HASSAN, A. E.; GODFREY, M. W.; NASSER, M.; FLORA, P. An empirical study on the practice of maintaining object-relational mapping code in java systems. In: IEEE. **2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)**. [S.l.], 2016. p. 165–176.

DRZAZGA, K.; BOBEL, M.; SKUBLEWSKA-PASZKOWSKA, M. Comparative analysis of selected object-relational mapping systems for the. net platform. **Journal of Computer Sciences Institute**, v. 16, p. 285–292, 2020.

DUARTE, N. F. B. **Frameworks e Bibliotecas Javascript**. Dissertação (Mestrado) — Curso de Engenharia Informática — Instituto Superior de Engenharia do Porto, Porto, 2015.

ELMASRI, R.; NAVATHE, S. B.; PINHEIRO, M. G. *et al.* **Sistemas de banco de dados**. Pearson Addison Wesley: São Paulo, 2005.

FAYAD, M. E.; SCHMIDT, D. C.; JOHNSON, R. E. **Building application frameworks**: object-oriented foundations of framework design. [S.l.]: John Wiley & Sons, Inc., 1999.

FORSGREN KALLIAMVAKOU, E. . N. **The 2021 State of the Octoverse**. [S.l.: s.n.], 2021.

FOWLER, M. **Patterns of Enterprise Application Architecture**: Pattern enterpr applica arch. [S.l.]: Addison-Wesley, 2012.

FOWLER, M.; RICE, D.; FOEMMEL, M.; HIEATT, E.; MEE, R.; STAFFORD, R. **Catalog of patterns of enterprise application architecture**. 2003. 25–26 p. Acesso em: 16 jun. 2022. Disponível em: <http://martinfowler.com/eaaCatalog/index.html>.

GAMATIÉ, A. Synchronous programming: overview. **Designing embedded systems with the SIGNAL programming language**, Springer, p. 21–39, 2010.

GIZAS, A.; CHRISTODOULOU, S.; PAPATHEODOROU, T. Comparative evaluation of javascript frameworks. In: **Proceedings of the 21st International Conference on World Wide Web**. [S.l.: s.n.], 2012. p. 513–514.

IRELAND, C.; BOWERS, D.; NEWTON, M.; WAUGH, K. A classification of object-relational impedance mismatch. In: **2009 First International Confernce on Advances in Databases, Knowledge, and Data Applications**. [S.l.: s.n.], 2009. p. 36–43.

LORING, M. C.; MARRON, M.; LEIJEN, D. Semantics of asynchronous javascript. In: **Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages**. [S.l.: s.n.], 2017. p. 51–62.

PINHO, D. M. de. **ECMAScript 6**: Entre de cabeça no futuro do javascript. [S.l.]: Editora Casa do Código, 2017.

PRESSMAN, R.; MAXIM, B. **Engenharia de Software**. 8. ed. [S.l.]: McGraw Hill Brasil, 2016.

PRESSMAN, R. S.; MAXIM, B. R. **Engenharia de software**. 9. ed. [S.l.]: McGraw Hill Brasil, 2021.

ROCHA, R. **Programação Concorrente**. Salvador: [s.n.], 2022.

SILBERSCHATZ, A.; KORTH, H. F.; SUDARSHAN, S. **System Concepts**. [S.l.: s.n.], 2008.

SILVEIRA, S. R.; VIT, A. R. D. de; BERTOLINI, C.; PARREIRA, F. J.; CUNHA, G. B. da. **Paradigmas de programação**: Uma introdução. Belo Horizonte: Synapse, 2021.

STRAUCH, C.; SITES, U.-L. S.; KRIHA, W. Nosql databases. **Lecture Notes, Stuttgart Media University**, v. 20, p. 24, 2011.

TILKOV, S.; VINOSKI, S. Node. js: Using javascript to build high-performance network programs. **IEEE Internet Computing**, IEEE, v. 14, n. 6, p. 80–83, 2010.

WIPHUSITPHUNPOL, W.; LERTRUSDACHAKUL, T. Fetch performance comparison of object relational mapper in. net platform. In: IEEE. **2017 14th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)**. [S.l.], 2017. p. 423–426.

YANNAKAKIS, M. Perspectives on database theory. **ACM SIGACT News**, ACM New York, NY, USA, v. 27, n. 3, p. 25–49, 1996.

ZYL, P. V.; KOURIE, D. G.; BOAKE, A. Comparing the performance of object databases and orm tools. In: **Proceedings of the 2006 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries**. [S.l.: s.n.], 2006. p. 1–11.