

UNIVERSIDADE ESTADUAL PAULISTA “JÚLIO DE MESQUITA FILHO”
FACULDADE DE CIÊNCIAS - CAMPUS BAURU
DEPARTAMENTO DE COMPUTAÇÃO
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

CASSIANO HENRIQUE APARECIDO RODRIGUES

**CRIAÇÃO DE UM MAPEADOR OBJETO-GRAFO PARA
OTIMIZAÇÃO E SIMPLIFICAÇÃO NO GERENCIAMENTO DE
BANCOS DE DADOS BASEADOS EM GRAFOS EM AMBIENTES
DE BIG DATA: UMA ANÁLISE COMPARATIVA COM
MAPEADORES OBJETO-RELACIONAL E OUTROS BANCOS DE
DADOS**

BAURU
Novembro/2023

CASSIANO HENRIQUE APARECIDO RODRIGUES

**CRIAÇÃO DE UM MAPEADOR OBJETO-GRAFO PARA
OTIMIZAÇÃO E SIMPLIFICAÇÃO NO GERENCIAMENTO DE
BANCOS DE DADOS BASEADOS EM GRAFOS EM AMBIENTES
DE BIG DATA: UMA ANÁLISE COMPARATIVA COM
MAPEADORES OBJETO-RELACIONAL E OUTROS BANCOS DE
DADOS**

Trabalho de Conclusão de Curso do Curso de Ciência da Computação da Universidade Estadual Paulista “Júlio de Mesquita Filho”, Faculdade de Ciências, Campus Bauru.

Orientador: Prof. Dr. Aparecido Nilceu Marana

Cassiano Henrique Aparecido Rodrigues

**Criação de um Mapeador Objeto-Grafo para otimização e
simplificação no gerenciamento de bancos de dados
baseados em grafos em ambientes de big data: Uma
Análise Comparativa com Mapeadores Objeto-Relacional
e outros bancos de dados**

Banca Examinadora

Prof. Dr. Aparecido Nilceu Marana

Orientador

Universidade Estadual Paulista “Júlio de

Mesquita Filho”

Faculdade de Ciências

Departamento de Computação

Professora Dra. Simone das Graças

Domingues Prado

Universidade Estadual Paulista “Júlio de

Mesquita Filho”

Faculdade de Ciências

Departamento de Computação

Prof. Dr. Leandro Aparecido Passos

Junior

Universidade Estadual Paulista “Júlio de

Mesquita Filho”

Faculdade de Ciências

Departamento de Computação

Bauru, 14 de Novembro de 2023.

Dedico esse trabalho à minha família, amigos, em especial minha mãe Sandra Lucia Rodrigues como uma homenagem póstuma e minha esposa. Pois sem essas pessoas minha vida não faria sentido.

“Talvez Deus tenha criado um mundo esférico para impedi-los de ver muito longe porque temia pelo futuro deles.”

Yoruhashi

Agradecimentos

Agradeço sobretudo a minha família, em especial meus pais, que sempre me ofereceram apoio e supriram minhas bases e condutas. Me ensinaram o que é amar e assim se deve conduzir a vida, amando as ações de outrem, as suas e as consequências as quais chegamos. Agradeço então meu pai Sidnei Aparecido Rodrigues, agradeço no formato de homenagem póstuma a minha mãe Sandra Lucia Rodrigues que até em seus últimos momentos de vida se importou comigo. A meus irmãos que sempre estiveram comigo me ensinando, advertindo, cuidando de mim: Caio e Camila Rodrigues.

Agradeço enormemente a minha namorada e futura esposa Isadora do Carmo Santos, que me ajuda a reconhecer o que não reconheço e mim, que me faz e fez de mim uma pessoa melhor, por ser uma pessoa que não sente vergonha de mim e que se espelha em mim como um prisma para seus sonhos, medos e desejos. Assim como minha sogra, agradeço por me suportar acordando às sete horas da manhã e me fazer um “café delícia” sempre que estou perto da sua casa.

Ademais, agradeço ao meu orientador Aparecido “Nilceu” Marana, que desde o início do curso, esteve presente em nossa turma, seja com o uso de sua imagem como um deus do panteão do curso de ciência da computação, no formato de “Nilceu Maromba”, como também um professor que adorava conversar na cantina do departamento de computação. Portanto, para mim foi um belo prazer ter sido orientado como provavelmente um dos seus últimos orientandos.

Humildemente, agradeço a todos os meus amigos e colegas que estavam em sala de aula comigo desde o ensino médio até o ensino superior, em especial os meus amigos, Davi, Gustavo, meu irmão Dhiego, Matheus, Nicolas, Gabriel Garcia, Caliman, Kaio, Luan, Gabriel Guimarães, Xilsu, Arthur, Renato, Ronaldo, às vezes o Polido, dentre outros. As minhas amigas, Lívia, Arissa, Marry, Nicole e Sofia, dentre outras. Às minhas “crias”, Anselmo e Suzy. Em especial, aos integrantes do Futuro que desde sempre preferíamos estar unidos, mesmo que sem falar uma palavra, mas sempre à disposição (espero que contem comigo mesmo após a Universidade). Somos o futuro além da UNESP, não no conceito de especiais, mas amigos, que vão além do espaço temporal do agora. Sempre fomos assim, a distância nunca foi um problema para comemorarmos um aniversário e nem para dizer uma palavra de conforto.

Além disso, agradeço a todos os membros do grupo seletivo denominado “Bostil” os quais estavam e estão todos os dias proferindo a verdade em busca do entretenimento, sendo eles: Gabex, Modscoleo4, Nalberto, Guguinha bostileiro, Fiscaliza, Fernando, Improve, Z, Toki, Davimedio e Pantene. Também agradeço a playlist de Phonk e a “Joe Sujera” que estavam comigo nos momentos que necessitavam de ânimo e agitação.

Novamente venho a agradecer, a minha mãe que sempre me apoiou e provavelmente sem ela não teria coragem para iniciar a universidade ou ter feito os vestibulares no momento de sua morte, mas que sempre vai estar em meu coração, ainda lembro fixamente de sua voz e rosto. Que Jesus te conforte e espero que esteja mandando “beijos mil” ao final dessa apresentação.

Obrigado a todos, independente de que momento você interferiu ou pensou em interferir na minha vida. Sou apenas grato. Que possamos um dia nos reunir e tomar um pouco de Coca.

Resumo

Este trabalho objetivou a criação de um mapeador objeto-grafo (OGM) para um banco de dados baseado em grafos, visando aprimorar a manipulação e recuperação de dados, especialmente em ambientes de *big data*. Em cenários de grande amplitude de dados, é crucial escolher soluções eficientes, uma vez que a complexidade desses ambientes pode impactar negativamente o desempenho.

A utilização de banco de dados baseados em grafos oferece vantagens significativas, mas também apresenta desafios, como a dificuldade na integração de ferramentas e alta complexidade operacional. Portanto, é introduzido uma camada de abstração na forma de OGMs. Esses mapeadores têm o papel de interpretar comandos, interagir com o banco de dados e mapear os dados para a aplicação.

O projeto concentrou-se na estruturação, implementação e avaliação desse mapeador, reconhecendo sua importância na simplificação da manipulação de dados em ambientes de *big data*. Para avaliar a eficiência do mapeador proposto, foram realizados testes exaustivos para capturar métricas e indicadores. Os resultados obtidos permitiram concluir que o mapeador demonstrou sua capacidade de otimizar a manipulação e recuperação de dados, proporcionando uma experiência simplificada para os desenvolvedores, além disso, os indicadores confirmaram a eficiência do mapeador validando sua capacidade de aprimorar a eficácia na manipulação de dados e fornecer uma interface intuitiva para o desenvolvimento de aplicações.

Palavras-chave: *Big data*; Banco de dados baseado em grafos; Mapeamento objeto-grafo; Abstração de dados; Otimização.

Abstract

This work aimed to create an Object-Graph Mapper (OGM) for a graph-based database, with the goal of enhancing data manipulation and retrieval, particularly in big data environments. In scenarios with vast amounts of data, choosing efficient solutions is crucial, as the complexity of these environments can negatively impact performance.

Graph-based databases offer significant advantages, but they also present challenges such as tool integration difficulties and high operational complexity. Therefore, we introduced an abstraction layer in the form of OGMs. These mappers have the role of interpreting commands, interacting with the database, and mapping the data to the application.

The project focused on structuring, implementing, and evaluating this mapper, recognizing its importance in simplifying data manipulation in big data environments. To assess the efficiency of the proposed mapper, exhaustive tests were conducted to capture metrics and indicators. The results obtained allowed us to conclude that the mapper demonstrated its ability to optimize data manipulation and retrieval, providing a streamlined experience for developers. Furthermore, the tests confirmed the mapper's efficiency, validating its capability to enhance data manipulation effectiveness and provide an intuitive interface for application development.

Keywords: Big data; Graph database; Object-Graph Mapper (OGM); Data abstraction; Optimization.

Listas de figuras

Figura 1 – Exemplo de um esquema do modelo relacional.	24
Figura 2 – Níveis de abstração em um SGDB.	26
Figura 3 – Esquema conceitual visual.	26
Figura 4 – Esquema conceitual em uma possível linguagem DDL.	27
Figura 5 – Exemplo da criação de uma entidade ¹ “Estudantes” com os seus respectivos campos.	28
Figura 6 – Exemplo de recuperação de dados usando uma DML sendo subconjunto da linguagem SQL.	29
Figura 7 – Exemplo de uma consulta usando SQL para identificar amigos em comum entre dois usuários em uma rede social fictícia.	30
Figura 8 – Exemplo de uma consulta que define uma variável	33
Figura 9 – Exemplo de uma consulta que procura pessoa que conhece uma pessoa . .	33
Figura 10 – Exemplo de atribuição de uma variável na cláusula “WITH”	33
Figura 11 – Exemplo de atribuição de uma variável na cláusula “MERGE”	33
Figura 12 – Três componentes presentes no termo “Big Data”	35
Figura 13 – Exemplo de uma sentença na linguagem C.	40
Figura 14 – Uma gramática para uma linguagem simples	41
Figura 15 – Derivação de um programa na linguagem descrita em Figura 14	41
Figura 16 – Exemplo de formato de nomes na linguagem C	44
Figura 17 – Exemplo de formato de nomes na linguagem Cloujure	45
Figura 18 – Exemplo de formato de nomes na linguagem Python	45
Figura 19 – Exemplo do uso da palavra “Integer” como palavra-chave e em seguida como identificador.	45
Figura 20 – Ilustração vetorizada de possíveis dispositivos	47
Figura 21 – Exemplo de um código escrito em TypeScript	53
Figura 22 – Distribuição dos produtos únicos nas categorias principais.	57
Figura 23 – Gráfico de mapa de árvore das categorias e subcategorias	57
Figura 24 – Gráfico de barras usando agrupamentos dos eventos por horário do dia . .	58
Figura 25 – Variação de tempo durante 300 buscas por ID.	59
Figura 26 – Todos os módulos presentes no OGM	60
Figura 27 – Enter Caption	61
Figura 28 – Diagrama de classe do módulo de interação com o banco de dados	62
Figura 29 – Definição da função que cria e executa uma transação	63
Figura 30 – Definição da função que executa as sentenças da linguagem de consulta .	64
Figura 31 – Estrutura da DDL criada.	66

Figura 32 – Exemplo de definição de um nó do tipo “Usuário” com um atributo “id” com incremento automático em um tipo inteiro	66
Figura 33 – Trecho da definição dos identificadores dos literais.	69
Figura 34 – Fluxograma simplificado do processo executado pelo analisador léxico.	69
Figura 35 – Fluxograma simplificado do processo executado pelo <i>parser</i>	70
Figura 36 – Árvore sintática construída	71
Figura 37 – Classe “ModelMap”	72
Figura 38 – Exemplo de como recuperar um modelo.	72
Figura 39 – Fluxograma do módulo de definição e estruturação do esquema	73
Figura 40 – Principais componentes do módulo de construção de consulta	74
Figura 41 – Fluxograma de execução simplificado sobre a execução do método “Create”	77
Figura 42 – Fluxograma da instanciação da aplicação	78
Figura 43 – Método que realiza o tratamento de erros comuns.	80
Figura 44 – Método que realiza o tratamento dos parâmetros de campos obtidos pela requisição.	81
Figura 45 – Definição do módulo dinâmico para o OGM desenvolvido.	83
Figura 46 – Definição do módulo para o PrismaJS.	84
Figura 47 – Definição da extensão do “PrismaClient”	84
Figura 48 – Estrutura da relação eventos entre produtos e a sessões de usuário.	86
Figura 49 – Estrutura do esquema para o OGM desenvolvido.	87
Figura 50 – Estrutura do esquema para a API desenvolvida usando Prisma e banco de dados relacional.	88
Figura 51 – Definição do serviço de categorias usando o <i>concern</i>	89
Figura 52 – Definição do serializador de categorias usando o <i>concern</i>	89
Figura 53 – Definição do controlador de categorias usando o <i>concern</i>	90
Figura 54 – Definição do <i>concern</i> de serviço na aplicação usando o OGM desenvolvido	91
Figura 55 – Definição do <i>concern</i> de serviço na aplicação usando o PrismaJS	93
Figura 56 – Atribuição do adaptador do Fastify a aplicação.	94
Figura 57 – Trecho das alterações realizadas no <i>concern</i> do controlador ao substituir o Express para o Fastify	95
Figura 58 – Contêineres executando no ambiente de produção.	97
Figura 59 – Consulta para realizar a importação do conjunto de dados ao Neo4j usando a linguagem Cypher	98
Figura 60 – Tempos das etapas de execução de uma consulta usando Prisma	99
Figura 61 – Requisição realizada no formato do comando Curl	100
Figura 62 – Três requisições a aplicação desenvolvida usando o OGM.	101
Figura 63 – Consultas que buscam 65 elementos das respectivas categorias: usuário, produto e categoria.	101
Figura 64 – Tempos das etapas de execução de uma consulta usando OGM.	103

Figura 65 – Tempos das etapas de execução de uma consulta usando Prisma	104
Figura 66 – Gráfico de gantt de um dos processos capturados do OGM dentre as 7500 consultas criadas	105
Figura 67 – Gráfico de gantt de um dos processos capturados do ORM dentre as 7500 consultas criadas	105
Figura 68 – Tempo de respostas da aplicação usando o OGM desenvolvido	106
Figura 69 – Proporção dos <i>status</i> das respostas da aplicação usando o OGM.	107
Figura 70 – Tempo de respostas da aplicação usando o Prisma	107
Figura 71 – Proporção dos <i>status</i> das respostas da aplicação usando o Prisma	108
Figura 72 – Informações sobre o banco de dados PostgreSQL usando “\I+”	108

Lista de quadros

Quadro 1 – Uma instância da relação Alunos.	24
Quadro 2 – Lexemas e <i>tokens</i> da sentença ignorando espaços em branco	40
Quadro 3 – Características de mapeadores encontrados	59
Quadro 4 – Informações sobre o banco de dados Neo4j usando “:sysinfo”	109

Lista de tabelas

Tabela 1 – Trecho do conjunto de dados	56
Tabela 2 – Trecho dos tempos capturados das consultas em OGM desenvolvido em milissegundos (ms).	102
Tabela 3 – Trecho dos tempos capturados das consultas usando o Prisma em milissegundos (ms)	104
Tabela 4 – Trecho das capturas do teste de requisições para aplicação que utiliza o OGM desenvolvido em milissegundos (ms)	106
Tabela 5 – Trecho das capturas do teste de requisições para aplicação que utiliza o ORM desenvolvido em milissegundos (ms)	106
Tabela 6 – Análise das métricas do mapeador relacional(3).	110
Tabela 7 – Análise das métricas do mapeador desenvolvido(2).	110
Tabela 8 – Métricas coletas das aplicações	112

Lista de abreviaturas e siglas

BI	<i>Business Intelligence</i>
CRUD	<i>Create, Read, Update, Delete</i>
DDL	<i>Data Definition Language</i>
DML	<i>Data Manipulation Language</i>
DTO	<i>Data Transfer Object</i>
ETL	<i>Extract, Transform, Load</i>
GPL	<i>General Public License</i>
OGM	<i>Object Graph Mapper</i>
ORM	<i>Object Relational Mapper</i>
POO	Programação Orientada a Objetos
REST	<i>Representational State Transfer</i>
SGDB	Sistemas de Gerenciamento de Banco de Dados
SQL	<i>Structured Query Language</i>
SSH	<i>Secure Socket Shell</i>

Sumário

1	INTRODUÇÃO	19
1.1	Problemática	20
1.2	Justificativa	20
1.3	Objetivos	21
1.3.1	Objetivo Geral	21
1.3.2	Objetivos Específicos	21
2	FUNDAMENTAÇÃO TEÓRICA	22
2.1	Banco de dados	22
2.1.1	Sistema de gerenciamento de banco de dados	22
2.1.1.1	Abstração de consultas: uma camada de software sobre os dados	22
2.1.2	Modelo de dados	23
2.1.2.1	Modelo relacional	23
2.1.2.2	Modelo baseado em grafo	24
2.1.3	Níveis de abstração em um SGDB	26
2.1.3.1	Esquema conceitual	26
2.1.3.2	Esquema físico	27
2.1.3.3	Esquema externo	27
2.1.4	Independência de dados	27
2.1.5	Linguagens do banco de dados	28
2.1.5.1	Linguagem SQL	29
2.1.5.1.1	Sintaxe	29
2.1.5.1.2	Exemplos de consultas	30
2.1.5.2	Cypher	31
2.1.5.2.1	Modelo de dados	31
2.1.5.2.2	Sintaxe e estrutura	32
2.2	Big data	34
2.2.1	Desafios associados	34
2.2.1.1	Variedade de dados	35
2.2.1.2	Volume de dados	35
2.2.1.3	Velocidade	36
2.2.1.4	Segurança dos dados	36
2.3	Mapeamento de banco de dados	36
2.3.1	Componentes	36
2.3.1.1	Conexão com o banco de dados	37

2.3.1.2	Mapeamento	37
2.3.1.3	Consulta de dados	37
2.3.1.4	Persistência de dados	37
2.3.1.5	Gerenciamento de transações	37
2.3.1.6	Otimização de consultas	37
2.3.1.7	Cache de dados	37
2.3.1.8	Validação de dados	37
2.4	Métricas de avaliação de desempenho	38
2.4.1	Métricas específicas para avaliar o desempenho de mapeadores	38
2.5	Linguagem	39
2.5.1	Conceitos envolvidos	39
2.5.1.1	Sintaxe	39
2.5.1.2	Métodos formais para descrever sintaxe	40
2.5.1.2.1	Gramáticas livres de contexto	40
2.5.1.2.2	Gramáticas e reconhecedores	42
2.5.1.3	Gramática de atributos	42
2.5.1.4	Análise léxica e sintática	42
2.5.1.4.1	Análise léxica	43
2.5.1.4.2	Análise sintática	43
2.5.2	Questões de projeto	43
2.5.2.1	Formato dos nomes	44
2.5.2.1.1	Palavras especiais	45
2.6	Modelo de projeto de software	46
2.6.1	Soluções	46
2.6.2	Multiplataforma	47
2.6.3	Desenvolvimento orientado a objetos	47
2.6.3.1	Conceitos	47
2.6.3.2	Princípios	48
2.6.3.2.1	S.O.L.I.D.	48
2.6.3.2.2	D.R.Y.	49
2.6.4	Desenvolvimento baseado em componentes	49
2.6.5	Desenvolvimento de software de código aberto	49
2.6.5.1	Princípios e valores	49
3	METODOLOGIA	51
3.1	Materiais	51
3.1.1	Tecnologias para o OGM	52
3.1.1.1	Neo4j	52
3.1.1.2	TypeScript	53
3.1.1.2.1	NodeJs	53

3.1.1.2.2	Mocha e Chai	53
3.1.1.2.3	Zod	54
3.1.2	Tecnologias para a API	54
3.1.2.1	Nest.js	54
3.1.2.2	PostgreSQL	55
3.1.2.3	PrismaJS	55
3.1.3	Conjunto de dados	55
3.1.3.1	<i>eCommerce behavior data from multi-category store</i>	55
3.1.3.1.1	Análise do conjunto de dados	56
3.2	Desenvolvimento do OGM	58
3.2.1	Módulo de configuração	61
3.2.2	Módulo de integração com o banco de dados	61
3.2.3	Módulo de definição e estruturação do esquema	64
3.2.3.1	Linguagem de definição	65
3.2.3.1.1	Analizador léxico	68
3.2.3.1.2	Analizador sintático e regras semânticas	70
3.2.3.2	Mapa de modelos	71
3.2.4	Módulo de construção de consulta	73
3.2.4.1	Classe de construção de consultas	75
3.2.5	Módulo de validação	77
3.2.6	Aplicação	78
3.3	Desenvolvimento da API	78
3.3.1	Tratamentos	79
3.3.1.1	Tratamento de erros	79
3.3.1.2	Tratamento de paginação	80
3.3.1.3	Tratamento de parâmetros	80
3.3.2	Rotas padrões	81
3.3.3	Criação do serviço do mapeador	81
3.3.3.1	Módulo do OGM	82
3.3.3.2	Módulo do ORM	84
3.3.4	Criação dos modelos	85
3.3.5	Otimizações	94
3.4	Hospedagem	95
3.4.1	“Dockerização” da aplicação	95
3.4.1.1	Criação da imagem	95
3.4.1.2	Gerenciamento da aplicação	96
3.4.1.2.1	Otimização	97
3.4.1.3	Importação do conjunto de dados	97
3.4.1.3.1	Neo4j	97

3.4.1.3.2	PostgreSQL	98
4	EXPERIMENTAÇÃO E ANÁLISE DOS RESULTADOS	100
4.1	Experimentos	100
4.1.1	Experimentos sobre as otimizações	100
4.1.2	Experimentos sobre o mapeador	102
4.1.3	Experimentos sobre a aplicação	105
4.1.4	Experimentos sobre o banco de dados	108
4.1.4.1	PostgreSQL	108
4.1.4.2	Neo4j	108
4.2	Análise dos resultados	109
4.2.1	Análise do mapeador desenvolvido	109
4.2.1.1	Análise de praticidade	111
4.2.2	Análise das aplicações	111
4.2.3	Análise do banco de dados	112
5	CONSIDERAÇÕES FINAIS	114
5.1	Trabalhos Futuros	114
	REFERÊNCIAS	115

1 Introdução

No que se diz respeito a banco de dados, é impossível negar a sua importância para qualquer aplicação no contexto atual, contudo, a linha do tempo é importante para explicar e justificar as tomadas de decisões na criação de novas tecnologias. A evolução dos bancos de dados é um tema de grande importância para a tecnologia da informação. Desde a criação do primeiro banco de dados relacional na década de 70, esses sistemas têm sido fundamentais para armazenar, organizar e gerenciar grandes volumes de dados de maneira eficiente, conforme Navathe (1992).

A partir da década de 2000, o crescimento exponencial dos dados e a evolução das tecnologias têm levado ao reaparecimento de tipos alternativos de bancos de dados, como os bancos de dados *NoSQL*. Esses bancos de dados são projetados para lidar com volumes massivos de dados em tempo real, com alta disponibilidade e escalabilidade, sem as restrições dos bancos de dados relacionais.

Dentre as alternativas destacam-se os baseados em grafos como uma solução adequada para armazenamento de grande massa de dados e gerenciamento complexo para dados que são relacionais. Ao contrário dos bancos de dados relacionais, que armazenam dados em tabelas e usam chaves estrangeiras para estabelecer relacionamentos entre tabelas, os bancos de dados de grafo são modelados em nós e relacionamentos, permitindo aos desenvolvedores modelarem relacionamentos complexos de forma mais intuitiva. Conforme Meng, Cai e Cui (2022) pode-se definir como um tipo de banco de dados *NoSQL* que utiliza a teoria dos grafos para armazenar e processar dados. No modelo de banco de dados baseado em grafos, os nós (vértices) representam as entidades e as arestas representam as conexões entre as entidades. Ressalta-se que, ao contrário dos bancos de dados relacionais, os bancos de dados baseados em grafos permitem que sejam criados relacionamentos complexos entre os dados e possibilitam que sejam feitas consultas eficientes, até mesmo aquelas que precisam, no modelo relacional, de múltiplas junções entre várias tabelas.

No entanto, trabalhar diretamente com um banco de dados de grafo pode ser desafiador para os desenvolvedores, ao envolver a manipulação de consultas e a lógica de acesso a dados de baixo nível. Portanto, para ampliar a eficiência e velocidade no desenvolvimento, torna-se primordial utilizar ferramentas que atuam no acesso e realizam a definição de dados, tais como as ferramentas chamadas *Object-Graph Mapper* (OGM) (DIETZE et al., 2016).

Com um OGM, os desenvolvedores podem trabalhar com objetos em sua aplicação em um nível mais alto de abstração, em vez de lidar diretamente com consultas e manipulação de dados no banco de dados. De acordo com Copeland e Maier (1984), o mapeamento objeto-relacional consiste em converter dados armazenados em um paradigma orientado a objetos,

em um sistema compatível com bancos de dados relacionais. No entanto, abstraindo para um OGM pode-se definir como mapeamento de objetos para um sistema compatível com grafos e tal objeto permite ao desenvolvedor interagir com os dados usando métodos e propriedades conforme o paradigma de orientação a objetos.

Atualmente, a construção de APIs tem se tornado uma tarefa cada vez mais comum no desenvolvimento de software, uma vez que permite a comunicação entre diferentes aplicações e plataformas. A utilização de OGM é uma prática comum nesse processo, já que permite a abstração da camada de acesso a dados, tornando o código mais limpo e organizado, além de simplificar a lógica de acesso ao banco de dados (ZHYKHARSKYI; GUBARYEVA, 2023).

1.1 Problemática

Assim como pontuado por Moniruzzaman e Hossain (2013), com o avanço da tecnologia, a quantidade de dados gerados pelas empresas vem aumentando exponencialmente, gerando uma crescente preocupação em relação ao armazenamento, processamento e análise desses dados. Os bancos de dados relacionais, que há muito tempo foram a solução preferida para armazenar, agora estão se tornando cada vez mais ultrapassados para lidar com abundância de dados. Além disso, como já foi mencionado, a modelagem relacional dos dados pode ser bastante limitante quando se trata de representar relacionamentos complexos entre os dados.

Diante desse contexto, os bancos de dados baseados em grafos têm se mostrado uma alternativa viável e promissora. Eles são especialmente eficientes em lidar com dados que possuem relacionamentos complexos e variedades de tipos de dados. No entanto, apesar de sua eficácia, a utilização de bancos de dados baseados em grafos ainda é um desafio para o desenvolvimento, devido à dificuldade de integrar as consultas e definição dos dados com segurança, uma vez que é permitido a entrada de qualquer tipo de dados a um nó, na maioria dos gerenciadores. Além disso, a dificuldade em integrar os bancos de dados baseados em grafos às ferramentas de alto nível usadas pelas empresas também é um fator que dificulta sua adoção. Isso ocorre porque essas ferramentas, como ferramentas de “*Business Intelligence*” (BI) e “*Extract, Transform, Load*” (ETL), são geralmente desenvolvidas para bancos de dados relacionais. Como resultado, as empresas acabam deixando de lado a possibilidade de utilizar bancos de dados baseados em grafos, mesmo quando eles seriam a solução ideal para seus problemas.

1.2 Justificativa

Nesse sentido, a criação e aplicação de um OGM para um banco de dados baseado em grafos surge como uma possível solução para esses desafios. Um OGM consegue mapear objetos em uma estrutura de dados de grafo e, assim, permitir o acesso aos dados de maneira

mais intuitiva e eficiente, que por este objetivo atribui todas as vantagens do paradigma de programação orientada a objetos, contribuindo com o desenvolvimento e o manuseio dos dados, tornando o processo mais fluido e eficiente. Além disso, a criação de um OGM oferece uma solução para conectar outras ferramentas disponíveis nas linguagens de alto nível comuns nas empresas, ampliando ainda mais as possibilidades de integração e simplificando a complexidade na interação com bancos de dados baseados em grafos.

1.3 Objetivos

1.3.1 Objetivo Geral

O objetivo geral deste trabalho é desenvolver e aplicar um OGM para um banco de dados baseado em grafos, visando aprimorar a manipulação e recuperação de dados no intuito de criar uma interface intuitiva e eficiente para criações de aplicações que conectem aos banco de dados.

1.3.2 Objetivos Específicos

- Realizar uma revisão bibliográfica sobre OGMs e bancos de dados baseados em grafos, identificando as principais características e desafios envolvidos na sua utilização;
- Analisar e selecionar um banco de dados baseado em grafos para ser utilizado na implementação do OGM;
- Projetar e implementar um OGM que permita a integração entre a aplicação e o banco de dados, simplificando o acesso e manipulação dos dados;
- Avaliar o desempenho do OGM desenvolvido em comparação com outras soluções existentes no mercado, utilizando métricas como tempo de resposta, consumo de recursos e escalabilidade; e
- Aplicar o OGM desenvolvido em um projeto real, verificando sua eficácia na manipulação e recuperação de dados em um contexto prático, validando sua praticidade.

2 Fundamentação Teórica

Este capítulo apresenta os conceitos e fundamentos teóricos necessários ao desenvolvimento deste trabalho.

2.1 Banco de dados

De acordo com Sadalage e Fowler (2013), os bancos de dados relacionais tornaram-se uma parte tão inerente à nossa cultura computacional que é comum não lhes dar a devida importância. Para compreender melhor o movimento NoSQL ao longo do tempo, é vital compreender a estrutura e as características dos bancos de dados relacionais e, ao mesmo tempo, rever seus conceitos e benefícios.

Em um contexto mais amplo, um banco de dados é uma estrutura de armazenamento que permite a organização, armazenamento e recuperação eficiente de informações. No entanto, um banco de dados contemporâneo vai além desse propósito básico. Ele lida com a concorrência, garantindo que várias operações de leitura e gravação possam ocorrer simultaneamente, mantendo a consistência dos dados. Além disso, é projetado para se integrar perfeitamente a um ecossistema de software abrangente. Isso implica que ele não opera isoladamente, mas consegue trocar dados com outros sistemas e aplicativos, possibilitando a criação de soluções tecnológicas mais ricas e flexíveis.

Desta forma, a essência de um banco de dados vai muito além do simples armazenamento de dados, abrangendo a capacidade de lidar com a concorrência e a integração eficaz em um ecossistema de software diversificado e dinâmico. Isso é essencial para atender às crescentes demandas das aplicações e sistemas modernos, e enfatiza a importância de reavaliar o papel fundamental dos bancos de dados relacionais em nossa cultura computacional.

2.1.1 Sistema de gerenciamento de banco de dados

Um sistema de gerenciamento de banco de dados (SGDB) é uma peça fundamental da infraestrutura de tecnologia da informação e sua necessidade surge da complexidade e quantidade de informações geradas e armazenadas. Para compreender plenamente a necessidade desses sistemas, é instrutivo considerar a visão apresentada por Ramakrishnan e Gehrke (2008).

2.1.1.1 Abstração de consultas: uma camada de software sobre os dados

Ramakrishnan e Gehrke (2008) argumentam que um dos papéis críticos dos SGDBs é fornecer uma camada de abstração entre os dados brutos e os aplicativos que os utilizam.

Essa abstração permite que os desenvolvedores e usuários finais interajam com os dados de maneira eficiente e intuitiva, sem a necessidade de compreender as complexidades subjacentes do armazenamento e da recuperação de dados. Ramakrishnan e Gehrke (2008) elencam as seguintes necessidades-chave de um SGDB:

- Independência de dados: os programas aplicativos não são expostos aos detalhes de representação e armazenamento de dados;
- Acesso eficiente aos dados: um SGDB utiliza uma variedade de técnicas para armazenar e recuperar dados eficientemente;
- Integridade e segurança dos dados: por se responsabilizar pelos dados, o SGDB também pode verificar se está íntegro, colocando restrições;
- Administração de dados: permitir organizar a representação de dados para realizar as sintonizações finas do armazenamento dos dados para garantir uma eficiente recuperação;
- Acesso concorrente e recuperação de falhas: Um SGDB permite múltiplos acesso aos dados simultaneamente, gerenciando bloqueios e executando transações¹ para evitar conflitos e garantir a consistência dos dados. Também sendo projetados para garantir a recuperação em falhas, possuindo então recursos de *backup* e restauração de dados.

2.1.2 Modelo de dados

De acordo com Ramakrishnan e Gehrke (2008), um modelo de dados é “uma coleção de construtores de alto nível para descrição dos dados que ocultam vários detalhes de baixo nível do armazenamento”, isto é, permite que o usuário defina os dados a serem armazenados.

2.1.2.1 Modelo relacional

Uma descrição de dados em termos de modelo de dados é chamada esquema (RAMAKRISHNAN; GEHRKE, 2008). No modelo relacional, o esquema de uma relação tem: seu nome, nome de cada campo (atributo, ou coluna), e o tipo de cada campo. A Figura 1 mostra o esquema Alunos, no modelo relacional, que armazena dados sobre alunos.

¹ Uma transação é uma execução qualquer de um programa de usuário em um SGBD. (A execução de um mesmo programa diversas vezes gerará diversas transações.) Esta é a unidade básica de alteração reconhecida pelo SGBD (RAMAKRISHNAN; GEHRKE, 2008).

Figura 1 – Exemplo de um esquema do modelo relacional.

Alunos	
CPF	string
nome	string
idade	number
média	decimal(5,2)

Fonte: Elaborada pelo autor.

O esquema Alunos informa que cada registro na relação Alunos tem 4 campos, sendo os nomes e tipos de campos conforme indicados na definição. Um exemplo de instância da relação Alunos pode ser visto no Quadro 1.

Quadro 1 – Uma instância da relação Alunos.

CPF	Nome	Idade	Média
839.711.330-45	Nicolas Gomes Barbosa	26	9,20
284.089.470-01	Isadora do Carmo Santos	21	10,00
752.789.180-71	Sandra Lucia Rodrigues	49	10,00
992.110.770-44	Sra. McCallister	120	2,24

Fonte: Elaborada pelo autor.

Cada linha na relação Alunos é um registro que descreve um aluno. Ramakrishnan e Gehrke (2008) observam que “é possível tornar a descrição de um conjunto de alunos mais precisa especificando as restrições de integridade, que são condições que os registros de uma relação devem satisfazer”.

2.1.2.2 Modelo baseado em grafo

Conforme definido por Angles e Gutierrez (2008), um modelo baseado em grafo é um tipo de modelo de dados caracterizado por:

1. Estrutura de dados:

Os dados e esquemas são representados por meio de grafos, ou por meio de estruturas de dados que generalizam a noção de um grafo, como hipergrafos ou hiper-nós. Esses grafos podem ser direcionados e também conter rótulos associados a seus elementos;

2. Manipulação de dados:

A manipulação de dados, conforme Gyssens, Paredaens e Gucht (1990), refere-se à forma como os dados são processados e modificados. Ela é realizada por meio de operações

que se baseiam em características dos grafos, como caminhos, vizinhanças, subgrafos, conectividade e estatísticas de grafos.

Portanto, ao trabalhar com dados em um modelo de grafo, as ações executadas envolvem a análise e a transformação desses dados com base na estrutura do grafo subjacente. Essas operações podem incluir a busca por caminhos nos subgrafos, identificação de subgrafos relevantes, verificação da conectividade entre nós. Devido a essa abordagem, na realização de consultas e operações em dados organizados, o modelo irá executar com maior eficiência;

3. Restrição de integridade:

Nesse contexto, existe uma forte preocupação com a separação entre esquema e instâncias, devido aos princípios de *design* de banco de dados e à necessidade de garantir a integridade, flexibilidade e recuperação dos dados. A separação entre esquema e instância é fundamental. Dado que ao separar claramente o esquema das instâncias, é possível definir regras de integridade que garantam que os dados armazenados estejam em conformidade com o esquema. Contudo, a separação proporciona flexibilidade ao permitir alterações no esquema sem impactar as instâncias existentes, facilitando o controle granular sobre os dados.

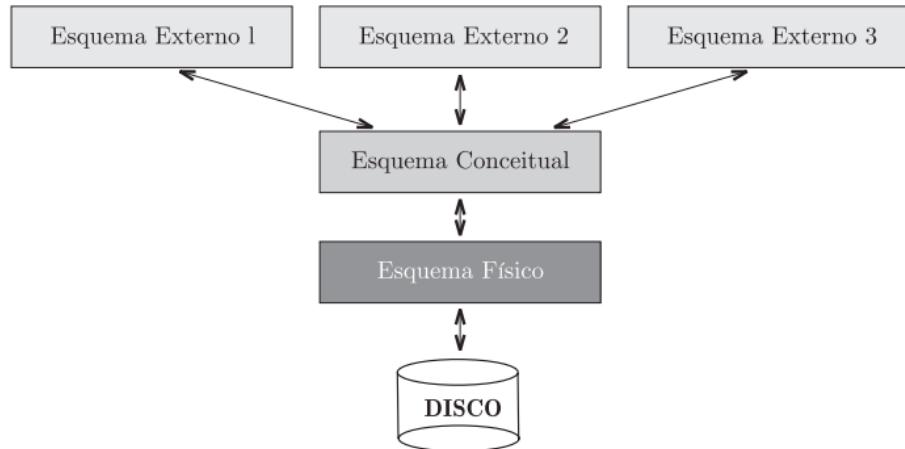
Banco de dados baseados em grafos são aplicados em áreas que a informação sobre a interconectividade ou topologia é muito importante como dito por Angles e Gutierrez (2008, p. 5), proporcionando algumas vantagens:

1. Modelagem natural: como dito por Paredaens, Peelman e Tanca (1995), as estruturas de grafos são visíveis para o usuário e possibilitam uma abordagem intuitiva para lidar com dados de aplicativos, tais como dados hipertexto ou geográficos. Os grafos têm a vantagem de poder armazenar todas as informações relacionadas a uma entidade em um único nó e apresentar informações correlacionadas por meio de arcos conectados a esse nó;
2. Consultas mais simples: as Consultas podem se referir diretamente a estrutura de grafo. Além de que, para os grafos, existem operações de grafo específicas na álgebra da linguagem de consulta, como a busca de caminhos mais curtos e a determinação de subgrafos específicos, possibilitando que os usuários expressem consultas em um nível mais elevado de abstração. Em certa medida, isso contrasta com a manipulação de grafos em bancos de dados dedutivos, onde as regras muitas vezes precisam ser bastante complexas assim define Güting (1994).

2.1.3 Níveis de abstração em um SGDB

De acordo com Ramakrishnan e Gehrke (2008), os dados são definidos em três níveis de abstração: o conceitual, o físico e o externo, conforme ilustrado na Figura 2.

Figura 2 – Níveis de abstração em um SGDB.



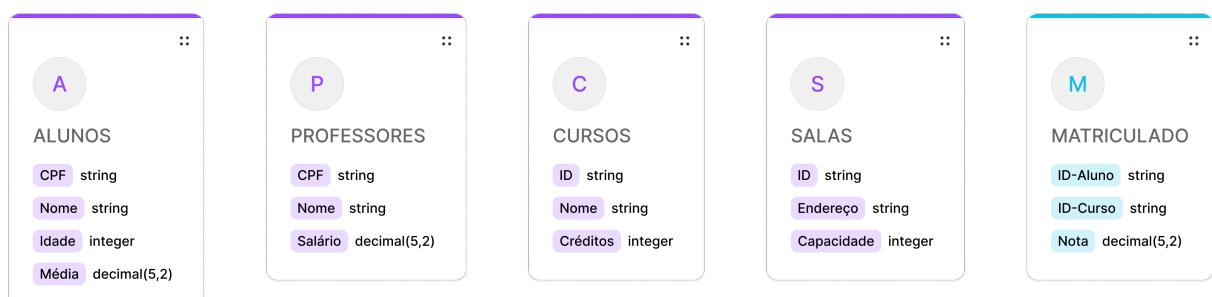
Fonte: Ramakrishnan e Gehrke (2008, p. 10).

Para definir os esquemas externos e conceituais é usado uma linguagem de definição de dados (DDL, *data definition language*).

2.1.3.1 Esquema conceitual

Consoante com Ramakrishnan e Gehrke (2008), o esquema conceitual, muitas vezes referido como esquema lógico, é uma representação que descreve os dados armazenados em um SGDB. Em um modelo entidade-relacionamento o esquema conceitual é utilizado para descrever as relações contidas no banco de dados. Essas relações abrangem informações sobre diversas entidades, em que cada entidade pode ser representada por meio de registros em uma relação específica, conforme demonstrado em Figura 3 e Figura 4.

Figura 3 – Esquema conceitual visual.



Fonte: Elaborada pelo autor.

Figura 4 – Esquema conceitual em uma possível linguagem DDL.

```
Alunos(CPF: string; Nome: string; Idade: integer; Média: decimal(5,2))
Professores(CPF: string; Nome: string; Idade: integer; Salário: decimal(5,2))
Cursos(ID: string; Nome: string; Créditos: integer)
Salas(ID: string; Endereço: string; Capacidade: integer)
Matriculado(ID-Aluno: string; ID-Curso: string; Nota: decimal(5,2))
```

Fonte: Elaborada pelo autor.

No processo de criação do esquema conceitual, a seleção das relações e a definição dos campos para cada relação podem não ser óbvias e diretas. Esse processo, envolve a produção de um esquema conceitual eficaz e adequado, sendo denominado projeto conceitual de banco de dados, e que conforme o modelo de dados usado pelo SGDB deve ter ajustes e planejamento adequado para maior eficiência, assim define Ramakrishnan e Gehrke (2008, p. 11).

2.1.3.2 Esquema físico

De acordo com Elmasri (2019, p. 34)

“O nível interno tem um esquema interno, que descreve a estrutura do armazenamento físico do banco de dados. O esquema interno usa um modelo de dados físico e descreve os detalhes completos do armazenamento de dados e os caminhos de acesso para o banco de dados.”

Em complemento, neste nível é decidido quais organizações de arquivos utilizar para armazenar as relações e criar estruturas de dados auxiliares para acelerar as operações de recuperação, conhecidas como índices de acordo com Ramakrishnan e Gehrke (2008, p. 11).

2.1.3.3 Esquema externo

Conforme Elmasri (2019, p. 34)

“O nível externo ou de visão inclui uma série de esquemas externos ou visões do usuário. Cada esquema externo descreve a parte do banco de dados que um grupo de usuários em particular está interessado e oculta o resto do banco de dados do grupo de usuários.”

Isto é, uma camada intermediária que atua como uma interface entre o esquema conceitual e os usuários ou aplicativos. O uso de esquemas externos ajuda a manter a segurança, privacidade, integridade e eficiência do banco de dados.

2.1.4 Independência de dados

Em Elmasri (2019, p. 35) são definidos dois tipos de independência de dados:

- Independência lógica de dados: capacidade de alterar o esquema conceitual sem ter de alterar os esquemas externos; e
- Independência física de dados: capacidade de alterar o esquema interno sem ter de alterar o esquema conceitual, consequentemente sem alterar os esquemas externos.

2.1.5 Linguagens do banco de dados

Para especificar os esquemas conceituais e internos para o banco de dados como definidos anteriormente é necessário usar a linguagem DDL (definition data language). Sendo assim, como apontado em Elmasri (2019, p. 36)

“O SGBD terá um compilador da DDL cuja função é processar instruções da DDL a fim de identificar as descrições dos construtores de esquema e armazenar a descrição de esquema no catálogo do SGBD”

A Figura 5 aponta um exemplo de uma DDL sendo subconjunto da linguagem SQL (Structured Query Language).

Figura 5 – Exemplo da criação de uma entidade² “Estudantes” com os seus respectivos campos.

```
CREATE TABLE Estudantes
(
    CodEstudante int NOT NULL,
    Nome varchar(255) NOT NULL,
    Sobrenome varchar(255) NOT NULL,
);
```

Fonte: Elaborada pelo autor.

Contudo, quando o banco é populado, os usuários precisam manipular os dados, executando operações como recuperação, inserção, exclusão ou modificação dos dados, portanto, O SGDB oferece uma linguagem chamada linguagem de manipulação de dados (DML, *data manipulation language*) para exercer essa função. A Figura 6 mostra um exemplo de comando DML para recuperar os dados dos estudantes com nome “João”.

² No modelo Entidade-Relacionamento, uma entidade representa um objeto ou conceito do mundo real que se deseja armazenar informações.

Figura 6 – Exemplo de recuperação de dados usando uma DML sendo subconjunto da linguagem SQL.

```
SELECT *
FROM Estudantes
WHERE Nome = 'João'
```

Fonte: Elaborada pelo autor.

2.1.5.1 Linguagem SQL

SQL é uma linguagem amplamente utilizada para consultas e manipulação de dados em banco de dados relacionais, conforme é dito em subseção 2.1.2.1, o modelo de dados se baseia em tabelas, colunas e linhas para representar informações.

2.1.5.1.1 Sintaxe

Conforme Melton e Simon (2001), a linguagem é composta por diversos comandos que realizam operações sobre um banco de dados. Os principais comandos são:

- SELECT: Usado para recuperar dados, especificando quais colunas devem ser retornadas;
- INSERT: Usado para adicionar novos registros a tabela, especificando tabela destino e os valores a serem adicionados;
- UPDATE: Usado para modificar dados preexistentes em uma tabela, especificando as colunas a serem atualizadas; e
- DELETE: Usado para remover registros de uma tabela, especificando tabela destino.

Além dos comandos, a linguagem SQL também contém cláusulas para refinar e completar as sentenças. As principais cláusulas são:

- FROM: Especifica a tabela da qual os dados devem ser recuperados;
- WHERE: Permite filtrar os resultados da consulta usando condições; e
- ORDER BY: Classifica os resultados em uma ordem específica.

Em conjunto com essa estrutura, a linguagem inclui operadores que permitem realizações comparações e cálculos em expressões. Os principais operadores são:

- Operadores de comparação: Estes operadores, como “=”, “>”, “<”, permitem comparar valores;

- Operadores lógicos: Estes operadores são usados para criar condições lógicas mais complexas, sendo eles: “AND”, “OR” e “NOT”; e
- JOINS: São operadores utilizados para combinar dados de duas ou mais tabelas, permitindo relacionar registros de tabelas diferentes com base em chaves em comum, criando um conjunto de resultados mais abrangente, sendo eles: “INNER JOIN”, “LEFT JOIN”, “RIGHT JOIN”, “FULL JOIN”, “CROSS JOIN” e “SELF JOIN”;

Conforme salientado por Elmasri (2019), os operadores JOINS são essenciais para lidar com os dados em que estão distribuídos em várias tabelas relacionadas. Eles permitem criar consultas mais complexas ao usar a extração de informações que abrangem mais de uma tabela.

As consultas podem ser aplicadas a uma variedade de cenários de gerenciamento de dados. No entanto, nem todos os cenários são iguais em termos de complexidade. Cenários que envolvem dados interconectados e altamente relacionais, como redes sociais, frequentemente apresentam desafios adicionais que requerem consultas mais elaboradas.

2.1.5.1.2 Exemplos de consultas

As redes sociais, como o Facebook, LinkedIn e Tumblr, são exemplos clássicos de cenários que envolvem um excesso de dados altamente interconectados. Nessas redes, os usuários criam amizades, seguem mutualmente, compartilham postagens, curtidas e comentários. Cada ação de um usuário pode gerar inúmeras interações e relacionamentos com outros usuários, resultado em um grafo complexo de conexões.

Esses cenários de redes sociais podem ser particularmente desafiadores quando se trata de realizar consultas eficazes para extrair informações significativas interconectadas

Um exemplo de consulta é a identificação de amigos em comum entre dois usuários, apresentada na Figura 7.

Figura 7 – Exemplo de uma consulta usando SQL para identificar amigos em comum entre dois usuários em uma rede social fictícia.

```
SELECT U1.nome AS amigo1, U2.nome AS amigo2
FROM Usuarios AS U1
JOIN Amizades AS A1 ON U1.id = A1.id_usuario
JOIN Amizades AS A2 ON A1.id_amigo = A2.id_amigo
JOIN Usuarios AS U2 ON A2.id_usuario = U2.id
WHERE U1.nome <> U2.nome;
```

Fonte: Elaborada pelo autor.

Nesta consulta, a complexidade reside na necessidade de várias junções entre tabelas, em que os componentes da consulta significam:

1. SELECT: Seleciona os nomes de amigos em comum, presentes em “U1” e “U2” que são variáveis que redirecionam para a tabela de Usuários;
2. JOIN: As junções são usadas para conectar os dados das tabelas “A1” e “A2”, que são variáveis que redirecionam para a tabela de Amizades, e “U1” e “U2”, que são variáveis que redirecionam para a tabela de Usuários
Essas variáveis apontam para a mesma tabela, no entanto, a conjuntos diferentes, que respeitam as condições presentes na cláusula “ON”;
3. WHERE: A cláusula filtra as amizades consigo mesmo, isto é, não irá realizar agrupamentos de um usuário consigo mesmo.

2.1.5.2 Cypher

Conforme Francis et al. (2018)

“Cypher is a declarative query language for property graphs. Cypher provides capabilities for both querying and modifying data, as well as specifying schema definitions”

Em resumo, Cypher é uma linguagem de consulta declarativa que possui os subconjuntos DDL e DML.

2.1.5.2.1 Modelo de dados

Segundo a análise de Libkin, Martens e Vrgoč (2016), é importante destacar que o modelo de dados empregado pelo Neo4j, usado pelo Cypher, corresponde ao paradigma de grafos mais amplamente adotado na indústria, o qual tem prevalecido de forma notável no meio acadêmico, considerado o estado da arte.

Tal como discutido em subseção 2.1.2.2, onde se exploram as características genéricas de um modelo de dados fundamentado em grafos, observa-se que o modelo adotado pelo Neo4j atribui aos nós do grafo a função de representar entidades, tais como pessoas ou contas bancárias, enquanto os relacionamentos (também denominados vértices) simbolizam as conexões ou vínculos que se estabelecem entre essas entidades. É importante ressaltar que é possível associar um número arbitrário de atributos, frequentemente chamados de propriedades, sob a forma de tuplas compostas por chave-valor, tanto aos nós quanto aos relacionamentos, possibilitando, desse modo, a modelagem e a consulta de dados de elevada complexidade, afirma Francis et al. (2018).

2.1.5.2.2 Sintaxe e estrutura

Conforme Francis et al. (2018), a linguagem é caracterizada por uma abordagem linear, cada cláusula em uma consulta atua como uma função que recebe uma tabela como entrada e gera uma tabela que pode expandir o número de campos ou adicionar novas tuplas. A consulta na totalidade é, então, a composição dessas funções. Essa ordem linear das cláusulas é compreendida de maneira puramente declarativa, tendo a liberdade de reordenar a execução das cláusulas, desde que isso não altere a semântica da consulta. Assim, ao contrário da declaração da projeção no início da consulta, como faz o SQL com o “SELECT”, em cypher, a projeção ocorre ao final da consulta, através da cláusula “RETURN”.

O fluxo linear das consultas em Cypher se estende também a composição das consultas. Usando a cláusula “WITH”, a consulta continua com a tabela projetada da parte da consulta anterior ao “WITH” como a tabela de base para a parte após essa cláusula, propagando os dados, permitindo agregações. Além disso, Cypher oferece suporte a filtragem com base nos campos projetados.

A estrutura do Cypher é construída por meio de várias cláusulas que descrevem a ação a ser executada semelhante ao SQL, no entanto, descreve os dados visualmente, semelhante a “arte ASCII”, isto é, usa-se uma notação visual para representar a estrutura de nós, relacionamentos e propriedades. As principais cláusulas incluem:

- MATCH: Usada para especificar os padrões a serem correspondidos no grafo;
- RETURN: Define quais dados devem ser retornados como resultado da consulta;
- WHERE: Permite filtrar os resultados com base em condições;
- CREATE: Utilizada para criar nós e relacionamentos no grafo;
- DELETE: Remove nós e relacionamentos do grafo;
- SET: Atualiza as propriedades dos nós e relacionamentos; e
- WITH: Divide a consulta em partes, permitindo propagar os resultados da primeira para a segunda;

Constituindo a estrutura para efetuar a projeção são necessárias variáveis para identificar elementos no grafo, assim permitindo vincular os resultados da consulta. Possuindo diversas formas de se declarar variáveis, sendo listadas a seguir:

- Atribuição direta:

primeira forma de declarar variáveis consiste na atribuição direta, onde um elemento do grafo (seja um nó, um relacionamento ou uma propriedade) é associado a uma variável

por meio de um operador de atribuição “=”. A Figura 8 mostra uma consulta que atribui o nó correspondente a uma variável denominada “Davi”:

Figura 8 – Exemplo de uma consulta que define uma variável

```
MATCH (n:Pessoa {nome: 'Davi'})
SET Davi = n
```

Fonte: Elaborada pelo autor.

- Atribuição em padrões:

Outra forma comum de declarar variáveis é no contexto de padrões de grafos especificados na cláusula “MATCH”. Nesse cenário, as variáveis são frequentemente utilizadas para representar nós e relacionamentos encaixados nos padrões definidos.

Figura 9 – Exemplo de uma consulta que procura pessoa que conhece uma pessoa

```
MATCH (n:Pessoa)-[r:CONHECE]-(m:Pessoa)
RETURN n, r, m
```

Fonte: Elaborada pelo autor.

- Uso de cláusula “WITH”:

Permite a declaração de variáveis na cláusula “WITH”, a qual é utilizada para criar variáveis intermediárias, posteriormente passadas entre diferentes partes da consulta; e

Figura 10 – Exemplo de atribuição de uma variável na cláusula “WITH”

```
MATCH (n:Pessoa)
WITH n AS pessoa
RETURN pessoa.name
```

Fonte: Elaborada pelo autor.

- Variáveis em cláusulas “MERGE”:

Cláusula que permite a correspondência de padrões e criação de elementos no grafo, assim como o “MATCH”.

Figura 11 – Exemplo de atribuição de uma variável na cláusula “MERGE”

```
MERGE (n:Pessoa {nome: 'Arissa'})
ON CREATE SET n.idade = 23
RETURN n AS pessoa
```

Em suma, as variáveis em Cypher desempenham um papel crucial na flexibilidade e na capacidade de manipulação de dados. A variedade de formas de declaração de variáveis possibilita uma abordagem versátil na formulação de consultas, atendendo às necessidades específicas do usuário no contexto da análise e recuperação de informações em grafos.

2.2 Big data

De acordo com Sagiroglu e Sinanc (2013)

Big data is a term for massive data sets having large, more varied and complex structure with the difficulties of storing, analyzing and visualizing for further processes or results

Isto é, se refere a uma quantidade massiva de dados com as mais variadas e complexas estruturas de dados, possuindo a dificuldade de armazenar analisar e visualizar. Dados estes sendo gerados por transações online, e-mails, vídeos, áudios, imagens, arquivos de log, publicações, pesquisas, redes sociais, sensores, dentre outros eventos e dispositivos.

2.2.1 Desafios associados

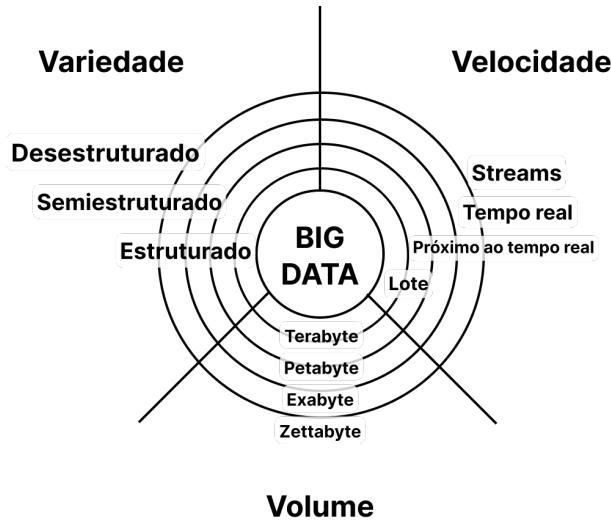
Conforme apresentado por Guide (2013)

5 exabytes (10^{18} bytes) of data were created by human until 2003. Today this amount of information is created in two days. In 2012, digital world of data was expanded to 2.72 zettabytes (10^{21} bytes). It is predicted to double every two years, reaching about 8 zettabytes of data by 2015

Segundo Singh e Singh (2012), dado que a IBM indica que a cada dia 2,5 exabytes são criados, o armazenamento dessa quantidade de dados crescentes e massivos se tornam uma grande dificuldade.

De acordo com Sagiroglu e Sinanc (2013), Big Data requer um avanço revolucionário em três componentes: variedade, velocidade e volume, conforme ilustra a Figura 12.

Figura 12 – Três componentes presentes no termo “Big Data”



Fonte: Adaptado de (SAGIROGLU; SINANC, 2013).

2.2.1.1 Variedade de dados

Conforme Sagiroglu e Sinanc (2013), os dados podem vir de uma grande variedade de fontes e essa variedade de fontes pode resultar em três tipos principais de dados:

- Dados estruturados: Dados com um formato definido e rígido sendo armazenados em banco de dados que suportam subseção 2.1.2.1;
- Dados semiestruturados: Dados que têm algum formato, mas não tão rígidos e imutáveis quanto os estruturados; e
- Dados não estruturados: Dados sem formato algum definido.

A variedade de dados grandes pode tornar a análise desafiadora, uma vez que dados estruturados podem ser facilmente analisados usando ferramentas tradicionais. No entanto, os dados semiestruturados e não estruturados requerem ferramentas e técnicas de análise mais avançadas.

2.2.1.2 Volume de dados

Assim como apontado em Sagiroglu e Sinanc (2013), e discutido em subseção 2.2.1, o tamanho dos dados agora é maior que terabytes e petabytes, superando a projeção das ferramentas e técnicas tradicionais de armazenamento.

O principal desafio é ter capacidade de armazenamento sem a necessidade de um *Data Center* o que graças aos serviços em nuvem e a possibilidade de escalar horizontalmente em seus serviços é permitido empresarialmente. Contudo, o problema ainda persiste.

2.2.1.3 Velocidade

De acordo com Sagiroglu e Sinanc (2013) velocidade é obrigatória não só no contexto de *big data*, mas em todos os processos, e ao ter muitos dados é necessário maximizar este valor.

2.2.1.4 Segurança dos dados

Além dos desafios relacionados à variedade, volume e velocidade, a segurança é uma preocupação primordial na análise de *Big Data*. Com o aumento na quantidade de informações, o controle e a proteção dos dados se tornam ainda mais críticos. A privacidade e a integridade dos dados devem ser mantidas, e as organizações precisam implementar estratégias ainda mais robustas de segurança de dados para evitar vazamentos e violações.

2.3 Mapeamento de banco de dados

De acordo com Torres et al. (2017) e Juneau e Juneau (2013), pode-se definir o conceito de mapeamento de banco de dados como um componente fundamental no desenvolvimento de sistemas que operam com bancos de dados. Os mapeadores consistem em uma tecnologia de integração de dados que desempenha um papel crucial na tradução entre as representações de dados utilizadas pelos bancos de dados e aquelas empregadas na programação orientada a objetos.

O principal objetivo do mapeador é servir como uma ponte entre o SGDB e as aplicações de *software* que adotam a programação orientada a objetos. Como resultado, o mapeador permite que os desenvolvedores trabalhem com os dados armazenados no banco de dados usando as mesmas estruturas e conceitos que utilizaram para qualquer outra operação com objetos em seus aplicativos. Esse alinhamento simplifica o desenvolvimento, uma vez que os desenvolvedores podem aplicar os princípios de orientação a objetos sem a necessidade de recorrer a técnicas especiais ou a necessidade de adquirir um conhecimento sólido de linguagens de consulta.

2.3.1 Componentes

De maneira geral, os mapeadores operam como uma camada de abstração entre o software da aplicação e o banco de dados subjacente. Essa tradução transparente entre as estruturas de dados dos objetos em memória e as tabelas do banco de dados, bem como a administração das operações de consulta e persistência dos dados, ocorre em diversos níveis por meio de mensagens entre os módulos, conforme destacado Torres et al. (2017) e Juneau e Juneau (2013).

2.3.1.1 Conexão com o banco de dados

A primeira função desempenhada por um mapeador é estabelecer a conexão com o banco de dados. Esse processo envolve a configuração das informações de conexão, como endereço, porta, credenciais de acesso e outros parâmetros, providos da aplicação.

2.3.1.2 Mapeamento

O mapeamento é uma etapa crítica em que objetos e classes da aplicação são mapeados para a estrutura do banco de dados e vice-versa. Essa tradução é essencial para garantir a correspondência correta entre os objetos na memória da aplicação e os banco de dados.

2.3.1.3 Consulta de dados

Uma das funcionalidades fundamentais de um mapeador é permitir que os desenvolvedores realizem consultas no banco de dados usando uma sintaxe orientada a objetos. Essa abordagem simplifica a recuperação de dados e torna as consultas mais legíveis.

2.3.1.4 Persistência de dados

O mapeador cuida da inserção, atualização e exclusão de registros no banco de dados, refletindo as alterações feitas nos objetos da aplicação, garantindo a integridade dos dados.

2.3.1.5 Gerenciamento de transações

Os mapeadores oferecem suporte ao gerenciamento de transações, garantindo que as operações no banco de dados sejam atômicas. Transações garantem a consistência dos dados e a integridade do banco de dados.

2.3.1.6 Otimização de consultas

Um mapeador pode otimizar as consultas geradas, garantindo que sejam eficientes.

2.3.1.7 Cache de dados

Alguns mapeadores oferecem suporte ao cache de dados, o que pode melhorar significativamente o desempenho, permitindo que os dados sejam armazenados em cache para acesso rápido.

2.3.1.8 Validação de dados

Os mapeadores podem oferecer recursos de validação de dados para garantir que os dados inseridos atendam aos requisitos e estejam íntegros.

Projetados para solucionar a dicotomia existente entre os SGDBs e os princípios da programação orientada a objetos mapeadores, desempenham um papel fundamental na convergência eficiente dessas abordagens. A questão principal reside na disparidade entre as estruturas de dados empregadas por ambos os sistemas, tornando essencial que a integridade e a tradução adequada sejam prioridades no processo.

2.4 Métricas de avaliação de desempenho

A gestão eficiente de dados desempenha um papel central no cenário da tecnologia da informação, onde a capacidade de armazenar, recuperar e manipular informações de maneira eficaz é essencial para o funcionamento de aplicações, como discutido em subseção 2.1.1 e conceitualizado em seção 2.2. À medida que a quantidade e a complexidade dos dados crescem, a avaliação do desempenho dos sistemas de banco de dados torna-se uma tarefa crítica, visando garantir que os sistemas atendam às demandas de processamento e disponibilidade, de acordo com Piattini et al. (2001).

Existem várias métricas e indicadores que podem ser usadas para avaliar o desempenho de um banco de dados, de acordo com Piattini et al. (2001), a saber:

- **Tempo de resposta:** Tempo que o banco de dados leva para atender uma consulta ou operação;
- **Throughput:** Indicador da quantidade de dados que o banco de dados pode processar em um determinado período;
- **Latência:** Atraso entre o envio de uma solicitação e o recebimento de uma resposta do banco de dados. Baixa latência é essencial para sistemas de tempo real e aplicativos sensíveis;
- **Utilização de recursos:** Monitorar a utilização da Unidade central de processamento (do inglês, CPU) e o uso de memória para identificar gargalos e problemas no gerenciador;
- **Planos de execução de consultas:** Analisar os planos de execução de consultas pode revelar consultas ineficientes ou não otimizadas.

2.4.1 Métricas específicas para avaliar o desempenho de mapeadores

Para avaliar o desempenho de mapeadores em sistemas de banco de dados, algumas métricas podem ser mais críticas dependendo dos objetivos do projeto. No entanto, algumas métricas gerais para a avaliação incluem:

- **Tempo de resposta:** Medir o tempo que o mapeador leva para traduzir consultas ou operações da camada de objetos para a linguagem de consulta e, em seguida, para executar

essas operações no banco de dados. Um tempo de resposta rápido é fundamental para manter a responsividade da aplicação;

- Eficiência de consulta: Analisar a qualidade das consultas geradas pelo mapeador; e
- Tratamento de erros e exceções: Avaliar como o mapeador lida com erros e exceções, como problemas de conexão ou consultas malformadas.

2.5 Linguagem

De acordo com Bruce (1994), a formação de linguagem de programação é um processo complexo que envolve a definição de conceitos-chave, a estruturação da sintaxe e da semântica, bem como a implementação de um compilador ou interpretador eficaz para traduzir o código-fonte em instruções executáveis.

2.5.1 Conceitos envolvidos

Conforme Sebesta (2018) pode-se definir os conceitos:

- Sintaxe: Estruturação das regras gramaticais que determinam como as instruções e expressões devem ser escritas;
- Semântica: Determina o significado das instruções e expressões;
- Tipagem: Define com os tipos de dados são tratados. A tipagem pode ser estática, em que os tipos são verificados em tempo de compilação, ou dinâmica, em que os tipos são verificados em tempo de execução;

2.5.1.1 Sintaxe

De acordo com (SEBESTA, 2018)

“Uma linguagem, seja natural (como a língua inglesa) ou artificial (como Java), é um conjunto de cadeias de caracteres formadas a partir de um alfabeto. As cadeias de uma linguagem são chamadas de sentenças”

As regras sintáticas de uma linguagem especificam quais cadeias de caracteres formadas a partir do alfabeto estão presentes e compõem a linguagem. E os lexemas³ em que sua descrição é definida por uma especificação léxica, no qual, incluem literais, operadores, e palavras especiais. Sendo divididos em grupos, como:

³ Conforme Sebesta (2018), os quais são as unidades sintáticas do mais baixo nível, representam a forma básica e indivisível de uma palavra ou símbolo que um analisador léxico reconhece e processa.

- Identificadores;
- Operadores;
- Palavras reservadas;
- Constantes.

Essa divisão em lexemas desempenha um papel fundamental na análise e processamento da linguagem por permitir que os compiladores e intérpretes comprendam e processam o código-fonte. Cada grupo de lexemas é representado por um nome, ou símbolo. Desta forma, um *token* de uma linguagem é uma categoria de seus lexemas. Considere a seguinte sentença em C e seus respectivos lexemas e tokens.

Figura 13 – Exemplo de uma sentença na linguagem C.

```
total = valor_item * quantidade_item + 5.25;
```

Fonte: Elaborada pelo autor.

Quadro 2 – Lexemas e *tokens* da sentença ignorando espaços em branco

Lexemas	Tokens
total	Identificador
=	Operador de igualdade
valor_item	Identificador
*	Operador de multiplicação
quantidade_item	Identificador
+	Operador de soma
5.25	Literal numérico
;	Ponto e vírgula

Fonte: Elaborado pelo autor.

2.5.1.2 Métodos formais para descrever sintaxe

Para definição e criação de linguagem são necessários mecanismos formais para geração de linguagem, geralmente chamadas de gramáticas⁴, segundo Sebesta (2018) que são usadas para descrever a sintaxe da linguagem.

2.5.1.2.1 Gramáticas livres de contexto

Assim como discorre Sebesta (2018), Chomsky descreveu quatro classes de linguagens: gramática com estrutura de frase ou irrestrita, gramáticas sensíveis ao contexto, gramáticas

⁴ Definido por Sebesta (2018), como um dispositivo de geração para definir linguagens.

livres de contexto e gramáticas regulares. Em que a definição dos reconhecedores definiram a forma dos *tokens* podendo ser descrita por expressões regulares.

Além disso, os axiomas da linguagem são geradas por meio de uma sequência de aplicação de regras, partindo de um não terminal especial da gramática chamado de símbolo inicial, enquanto a geração de uma sentença é chamada de derivação, assim é dito por Sebesta (2018).

Figura 14 – Uma gramática para uma linguagem simples

```

<program> → begin <statement_list> end
<statement_list> → <statement>
                  | <statement> ; <statement_list>
<statement> → <var> = <expression>
<var> → A | B | C
<expression> → <var> + <var>
                | <var> - <var>
                | <var>
  
```

Fonte: Adaptado de (SEBESTA, 2018, p. 142)

A linguagem escrita pela gramática da Figura 14 tem a seguinte derivação de um programa na linguagem:

Figura 15 – Derivação de um programa na linguagem descrita em Figura 14

```

<program> → begin <statement_list> end
            → begin <statement> ; <statement_list> end
            → begin <var> = <expression> ; <statement_list> end
            → begin A = <expression> ; <statement_list> end
            → begin A = <var> + <var> ; <statement_list> end
            → begin A = B + <var> ; <statement_list> end
            → begin A = B + C ; <statement_list> end
            → begin A = B + C ; <statement> end
            → begin A = B + C ; <var> = <expression> end
            → begin A = B + C ; B = <expression> end
            → begin A = B + C ; B = <var> end
            → begin A = B + C ; B = C end
  
```

Fonte: Adaptado de (SEBESTA, 2018, p. 142)

Desta forma, cada uma das cadeias na derivação incluindo “*<program>*”, é chamada de forma sentencial, conforme Sebesta (2018).

O SEBESTA disserta que um dos recursos mais atrativos das gramáticas é que elas descrevem a estrutura hierárquica estática das sentenças das linguagens as quais definem. Em

que, essas estruturas são chamadas de árvores de análise sintática, estruturadas de forma que cada nó interno é rotulado com um símbolo não terminal, e em cada folha, é rotulada com um símbolo terminal, o qual, cada sub-árvore desta árvore descreve uma instância de uma abstração da sentença. Assim, essas árvores permitem examinar a linguagem de forma sintática, resultando na possibilidade de determinar se a gramática é ambígua.

Conforme Sebesta (2018), a ambiguidade sintática é um problema, pois os compiladores baseiam a semântica nas estruturas de sua forma sintática, isto é, o significado da estrutura não pode ser determinado unicamente, podendo trazer resultados inesperados para linguagens de programação.

2.5.1.2.2 Gramáticas e reconhecedores

De acordo com Sebesta (2018), “Dada uma gramática livre de contexto, um reconhecedor para a linguagem gerada pela gramática pode ser algorítmicamente construído”

2.5.1.3 Gramática de atributos

Segundo Sebesta (2018) as linguagens têm características estruturais que podem ser difíceis como também impossíveis descrever apenas com a gramática, como a compatibilidade de tipos. Em linguagens de programação como Java, regras como a impossibilidade de atribuir um valor de ponto flutuante a uma variável do tipo inteiro precisam de regras adicionais. Outra condição comum mencionado por Sebesta (2018) é a regra que exige que todas as variáveis sejam declaradas antes de serem referenciadas, sendo incapaz de expressar por meio da gramática. Tais desafios configuram as regras de linguagem denominadas regras semânticas.

A semântica estática de uma linguagem, lida com as formas permitidas dos programas, em vez, de seu significado durante a execução. Para abordar essas questões, foram desenvolvidos mecanismos, quais são as gramáticas de atributos, concebidas por Knuth (1968), como explicado por Sebesta (2018). As gramáticas de atributos são uma abordagem formal usada para descrever tanto a sintaxe quanto a semântica estática de programas, desempenhando um papel na especificação e verificação das regras semânticas.

A análise necessária para verificar as especificações de semântica estática é realizada em tempo de compilação, e as gramáticas de atributos fornecem a estrutura para essa análise.

2.5.1.4 Análise léxica e sintática

Existem três abordagens conforme lista Sebesta (2018): compilação, interpretação pura e implementação híbrida. Em que, a compilação é usada para implementar linguagens de programação para grandes aplicações, enquanto, sistemas de interpretação pura é normalmente usada para sistemas menores, nos quais a eficiência de execução não é crítica.

Ainda segundo o autor, analisadores sintáticos, são baseados em uma descrição formal da sintaxe, isto é, na gramática utilizada. Em suma, a maioria dos compiladores separa a tarefa de analisar a sintaxe em duas partes distintas, as análises léxica e sintática.

2.5.1.4.1 Análise léxica

É descrito um analisador léxico como um casador de padrões, que busca uma subcadeia de uma cadeia de caracteres que se associe a um padrão de caracteres. Sebesta (2018) descreve que, o analisador coleta caracteres e os agrupa, conforme sua estrutura, em que, os agrupamentos lógicos são chamados de lexemas e os códigos internos dessa categoria são chamados de *tokens*.

Os lexemas são reconhecidos por meio da associação aceita de um padrão de caracteres, os *tokens* são constantes nomeadas após identificação da categoria dos lexemas. A execução da análise léxica geralmente inclui deixar comentários e espaços em branco fora dos lexemas, por não serem relevantes para a semântica do programa, além disso, os analisadores detectam erros sintáticos em *tokens*.

Portanto, o analisador léxico é responsável pela construção da tabela de símbolos, armazenando informações acerca dos nomes provados pelo usuário e assim atributos como: tipo, parâmetros e nome.

2.5.1.4.2 Análise sintática

De acordo com Sebesta (2018), a análise sintática, também chamada de *parsing* é responsável por construir a árvore de análise sintática ao desempenhar dois objetivos: verificar o programa de entrada se está sintaticamente correto e produzir uma mensagem de erro caso seja encontrado. Os analisadores são divididos conforme a direção na qual é construído as árvores, sendo descendentes, o qual a árvore é construída a partir da raiz em direção às folhas, e ascendentes, a partir das folhas em direção à raiz.

Segundo Sebesta (2018), “Todos os algoritmos usados pelos analisadores sintáticos dos compiladores comerciais têm complexidade $O(n)$ ”, isto é, são lineares com o tamanho da cadeia a ser analisada sintaticamente.

2.5.2 Questões de projeto

Conforme Sebesta (2018) o projeto de uma linguagem de programação é um equilíbrio delicado entre a criação de uma linguagem que seja poderosa, eficiente e fácil de usar.

As questões de projeto permeiam diversos tópicos:

- Formato dos nomes;

- Palavras especiais;
- Vinculação de atributos a variáveis;
- Vinculação de tipos;
- Vinculações de armazenamento e tempo de vida;
- Definição de escopo; e
- Ordem de declaração.

2.5.2.1 Formato dos nomes

Segundo o Sebesta (2018), “um nome é uma cadeia de caracteres usada para identificar alguma entidade em um programa”, podendo definir limites como tamanho limite, e condições para identificação, como, por exemplo, se os nomes serão sensíveis à capitalização. Outra possibilidade é definir se as palavras especiais da linguagem serão consideradas palavras reservadas ou palavras-chave.

Essas limitações são impostas devido construção da tabela de símbolos durante a compilação. Além disso, as linguagens podem impor formato caracteres no formato ou recomendar um padrão, como na maioria das linguagens como: uma letra seguida de uma cadeia de letras, dígitos e sublinhados. Assim como destaca Sebesta (2018), nas linguagens baseadas em C⁵, o formato “CamelCase”⁶ é utilizado como padrão da linguagem, em que, consiste que todas as palavras de nome contendo múltiplas palavras têm a primeira letra maiúscula, exceto pela primeira palavra. Conforme é apresentado nas Figuras 16, 17 e 18, diversos formatos em múltiplas linguagens, é possível notar que o uso de sublinhados e de capitalização mista nos nomes é uma questão de estilo.

Figura 16 – Exemplo de formato de nomes na linguagem C

```
// Exemplo na linguagem "C"
somaticoTotal = 0
```

Fonte: Elaborada pelo autor.

⁵ Uma linguagem de programação de alto nível criada nos anos 1970 para desenvolvimento de sistemas operacionais.

⁶ Chamada de “camelo” por conta que as palavras escritas tem uma notação de letras maiúsculas, que se assemelham com as corcovas de um camelo.

Figura 17 – Exemplo de formato de nomes na linguagem Clojure

```
;; Exemplo na linguagem "Clojure"
(def somatorio_total = 0)
```

Fonte: Elaborada pelo autor.

Figura 18 – Exemplo de formato de nomes na linguagem Python

```
# Exemplo na linguagem "Python"
somatorio_total = 0
```

Fonte: Elaborada pelo autor.

2.5.2.1.1 Palavras especiais

Conforme Sebesta (2018), palavras especiais são usadas para legibilidade das ações a serem realizadas, separando as partes sintáticas das sentenças e programas, destacando que, na maioria das linguagens, são classificadas como palavras reservadas, mas há exceções classificando como palavras-chave.

Além disso, algumas linguagens incluem nomes pré-definidos, que estão entre as palavras especiais e possivelmente entre os nomes definidos pelo usuário. Desta forma, alguns podem ser redefinidos pelo usuário e outros não, conforme cada linguagem.

Consoante com Sebesta (2018), palavra-chave é uma palavra de uma linguagem de programação especial de acordo com alguns contextos e condições. Como, por exemplo, na linguagem Fortran, a palavra “Integer”, quando encontrada no início de uma sentença seguida de um identificador, é considerada palavra-chave que indica a tipagem estática, conforme ilustrado na Figura 19.

Figura 19 – Exemplo do uso da palavra “Integer” como palavra-chave e em seguida como identificador.

```
Integer Apple
Integer = 4
```

Fonte: Adaptado de (SEBESTA, 2018)

Uma palavra reservada é definida como uma palavra especial que não pode ser usada como identificador, conforme Sebesta (2018). Como uma escolha de projeto de linguagem,

estas palavras são preferíveis no sentido de evitar confusão, contudo, ao incluir uma quantidade grande palavras reservadas, pode dificultar a definição de nomes do usuário.

2.6 Modelo de projeto de software

Conforme Brooks (2018) projetos devem ser planejados e mesmo sendo voláteis é preciso ser rapidamente executados devido aos ciclos de vida, portanto, o planejamento deve ser bem feito. Desta forma, as melhores práticas conforme Pressman e Maxim (2021) e práticas comuns no desenvolvimento de aplicações utilizando o modelo de grafos foram elencados.

2.6.1 Soluções

A escolha da linguagem de programação e ecossistema desempenha um papel fundamental no desenvolvimento de software. Entre os aspectos mais significativos a serem considerados na seleção de uma linguagem está na tipagem, que pode ser estática ou dinâmica, como discuto em subseção 2.5.1.3.

Ou seja, durante o desenvolvimento, ao adotar o uso de uma linguagem de tipagem estática, é possível obter vantagens como:

- Detecção antecipada de erros;
- Melhor documentação e compreensão do código;
- Melhor desempenho;
- Facilidade na refatoração; e
- Ferramentas de desenvolvimento.

Isto é, com essa abordagem é possível extrair vantagens durante o desenvolvimento como o uso do Intellisense (MICROSOFT, 2023b), que oferece autocompletar, análise sintática para identificar operadores, funções, variáveis. Desta forma, aprimorando a produtividade do desenvolvedor, ao fornecer sugestões de código em tempo real, documentação de funções e classes, além de evitar erros de digitação, reduzindo o tempo necessário para concluir tarefas de programação.

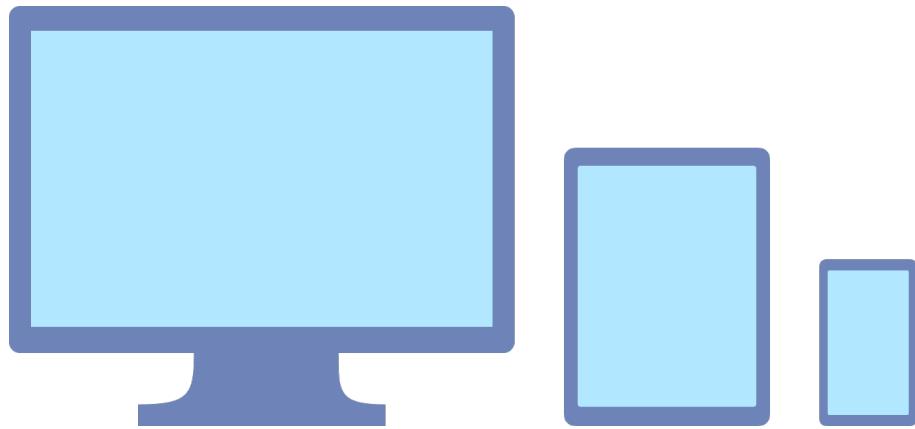
Além disso, outra vantagem notável de utilizar uma linguagem de tipagem estática é a facilidade de substituição de módulos, aproveitando as interfaces predefinidas, conforme subseção 2.5.1.3 e implementado como princípio em subseção 2.6.3.2.1. Através da definição clara e estática dos tipos de dados, a gramática de atributos da linguagem fornece um contexto seguro para a substituição de módulos. Isso significa que, quando se deseja modificar ou atualizar um componente do sistema, ele pode ser facilmente substituído por outro módulo que

siga a mesma interface, sem a preocupação de incompatibilidades de tipos ou funcionalidades. Isso simplifica a manutenção do código, tornando-o mais flexível e adaptável a mudanças futuras, o que é uma vantagem significativa em projetos de software de longa duração, segundo Silva et al. (2001). Portanto, a tipagem estática e as interfaces bem definidas contribuem para um desenvolvimento mais robusto e sustentável.

2.6.2 Multiplataforma

Considerando a definição de *cross-platform*, “able to be used with different types of computer systems” (CROSS-PLATFORM., 2023), a criação de um projeto de software tem como um público alvo, podendo inferir e definir qual a plataforma de ação do software, mas está ficando cada vez mais comum encontrar projetos multiplataforma em que é possível utilizar utilizando dispositivos móveis, grandes eletrônicos, computadores pessoais ou até mesmo eletrodomésticos, assim como ilustrado na Figura 20.

Figura 20 – Ilustração vetorializada de possíveis dispositivos



Fonte: Elaborada pelo autor.

2.6.3 Desenvolvimento orientado a objetos

De acordo com Cox (1986) e Pressman e Maxim (2021) a abordagem de programação orientada a objetos (POO) é um paradigma amplamente utilizado na construção de software moderno. Se baseando na representação de entidades reais e suas interações como objetos. O desenvolvimento utilizando tal paradigma oferece vantagens como eficácia em modelar sistemas complexos, tornando o código mais modular, reutilizável e reativo.

2.6.3.1 Conceitos

O cerne da POO é a modelagem de entidade e comportamentos da realidade, usando a representação de objetos. Em POO, um objeto é uma instância específica de uma classe, a qual é uma descrição abstrata contendo atributos, também chamados de propriedades, e

métodos, também chamados funções, que os objetos criados a partir dela possuirão, de acordo com Cox (1986). Além disso, incluí outros conceitos básicos como:

- Encapsulamento: Envolve o agrupamento de dados e funcionalidades relevantes, permitindo um controle preciso do acesso e em associação com a troca de mensagens, é um pilar da POO;
- Herança: Permite que novas classes sejam derivadas de classes existentes; e
- Polimorfismo: Permite que objetos de diferentes classes sejam tratados de maneira uniforme, desde que eles respondam a mensagens da mesma forma, criando a possibilidade de realizar composição entre classes.

2.6.3.2 Princípios

No entanto, o sucesso da POO não se resume apenas à aplicação de conceitos básicos, mas também à incorporação de princípios de projeto, que promovem a criação de sistemas eficazes e fáceis de manter. Entre esses princípios, é destacado o acrônimo S.O.L.I.D e o princípio D.R.Y. (do inglês, *Don't Repeat Yourself*).

2.6.3.2.1 S.O.L.I.D.

Conforme Martin (2009) o acrônimo, S.O.L.I.D. representa cinco princípios fundamentais da programação orientada a objetos, em que, cada letra em S.O.L.I.D. corresponde a um princípio específico:

1. S - princípio da responsabilidade única: Esse princípio afirma que uma classe deve ter apenas uma razão para mudar. Em outras palavras, uma classe deve ter uma única responsabilidade ou tarefa;
2. O - princípio do aberto/fechado: Enfatiza que as classes devem estar abertas para extensão, mas fechadas para modificação. Isto é, uma classe pode estender o seu comportamento sem precisar alterar seu código-fonte;
3. L - princípio da substituição de Liskov: Declara que os objetos de uma classe derivada devem poder ser substituídos por objetos da classe base sem afetar a assertividade do programa. Isso garante que a herança seja usada de maneira consistente e que as subclasses estejam verdadeiramente especializando a classe;
4. I - princípio da segregação de interfaces: Enfatiza de que as interfaces de uma classe devem ser específicas para os que usam; e
5. D - princípio da inversão de dependência: Define que as classes de alto nível não devem depender das classes de baixo nível, mas ambas devem depender de abstrações.

2.6.3.2.2 D.R.Y.

O princípio D.R.Y. é uma diretriz fundamental na programação em geral, que destaca a importância de evitar a duplicação de código, enfatizando que a manutenibilidade mais difícil.

2.6.4 Desenvolvimento baseado em componentes

Conforme salientado em Pressman e Maxim (2021) é uma abordagem na engenharia de software que destaca ser eficaz para a criação de sistemas complexos, que se define como, a construção de módulos independentes e reutilizáveis, sendo montados atomicamente que podem se comunicar mediante mensagens estruturadas.

Ainda segundo o autor, um componente de software é uma unidade autônoma e autocontida que encapsula uma funcionalidade específica, devendo ser projetado para ser independente. Proporcionando, ciclos de vidas isolados, reduzindo a duplicação e aumentando a eficiência do desenvolvimento.

Desta forma, em Pressman e Maxim (2021) destaca que para o desenvolvimento baseado em componentes seja eficaz, é importante que os componentes sejam altamente coesos e fracamente acoplados. A alta coesão implica que um componente deve realizar uma tarefa claramente definida e relacionada a um objetivo específico, enquanto, o baixo acoplamento garante que os componentes não dependam fortemente uns dos outros. Além disso, a devida aplicabilidade deve favorecer a escalabilidade, à medida que as necessidades crescem, novos componentes podem ser adicionados ou substituídos, se adaptando a mudança de requisitos.

2.6.5 Desenvolvimento de software de código aberto

De acordo com Wu e Lin (2001) o desenvolvimento de software de código aberto é um movimento que revolucionou a maneira como a tecnologia é criada, compartilhada e desenvolvida.

2.6.5.1 Princípios e valores

O desenvolvimento de código aberto é fundamentado em vários princípios e valores essenciais:

- Transparência: Código-fonte é acessível ao público;
- Colaboração: Projetos de código aberto sempre que uma pessoa faz uma alteração, ela pode compartilhar e colaborar com o projeto;
- Licenças abertas: Uso de licenças de código aberto define os termos para utilização, modificação e distribuição do software. As licenças, como a *General Public License*

(GPL), garantem que o código permaneça aberto e seja redistribuído sob os mesmos termos;

- Compartilhamento e reutilização: Promove o compartilhamento de soluções, sendo possível reutilizar componentes; e
- Comunidade: Muitos projetos têm uma comunidade engajada e especialistas que contribuem ativamente para o desenvolvimento e suporte contínuo do software.

O desenvolvimento de software de código aberto se tornou parte essencial do panorama tecnológico moderno, devido à inovação contínua e colaboração ativas, conforme se afirma em Hauge, Ayala e Conradi (2010).

3 Metodologia

Este capítulo aborda o processo de construção do OGM e sua implantação em um projeto, incluindo detalhes sobre os materiais utilizados, a modelagem adotada, a arquitetura do mapeador e o desenvolvimento da ferramenta. Para fornecer uma compreensão abrangente, cada módulo é descrito de forma independente, seguido pela apresentação e explicação conjunta dos fluxogramas relativos a esses módulos.

No desenvolvimento do OGM, os conceitos fundamentais delineados em Pressman e Maxim (2021) foram incorporados como diretrizes essenciais. Um desses conceitos foi a modularidade que, conforme descrito em Pressman e Maxim (2021), refere-se à capacidade de dividir um sistema em componentes independentes, conhecidos como módulos, que podem ser desenvolvidos, testados e mantidos separadamente, facilitando, dessa forma, a compreensão, a manutenção e a escalabilidade do sistema.

Ao adotar o princípio da modularidade, o OGM foi projetado para permitir a construção e o aprimoramento de cada módulo de maneira independente, promovendo a flexibilidade e a adaptabilidade do sistema à medida que novos requisitos e funcionalidades são incorporados.

3.1 Materiais

A escolha do software e do hardware utilizados para a construção e a implementação do OGM, focou na simulação de um ambiente real de hospedagem de uma aplicação.

- ***Software***

- Desenvolvimento
 - * Sistema Operacional: Pop_OS 22.04; e
 - * Ambiente de Desenvolvimento Integrado: Microsoft Visual Studio Code.
 - Servidor
 - * Sistema Operacional: Ubuntu 22.04.3 LTS.
 - Contêiner
 - * Sistema Operacional: Alpine Linux 3.18; e
 - * Javascript Runtime: Node 20.8.1.

- ***Hardware***

- Desenvolvimento

- * Notebook Lenovo ideiapad gaming 3i: AMD Ryzen 7 5800H @ 3,2 GHz com uma placa de vídeo NVIDIA GeForce GTX 1650 com 4 GB de RAM GDDR6, equipado com 20 GB de RAM DDR4, contendo dois SSDs com 512 GB M.2 NVMe e 256 GB M.2 NVMe de armazenamento; e
- Servidor
 - * Computador: Intel Core i5-10400 CPU @ 2.90GHz equipado 32 GB de RAM DDR4 com 5 GB de Swap, contendo 4 dispositivos de armazenamento: 2 SSDs M.2 NVMe 1 TB e 512 GB, 2 HD 7200rpm 4 TB e 1 TB.

3.1.1 Tecnologias para o OGM

Para a construção do OGM foram cuidadosamente selecionadas uma variedade de tecnologias a fim de atender às necessidades do projeto. Tais tecnologias são descritas a seguir.

3.1.1.1 Neo4j

Como um dos componentes-chave que merece destaque é o SGDB que implementa o modelo de grafo escolhido, que deve representar a maior parcela das aplicações que utilizam esse modelo. Desta forma, a decisão de empregar o Neo4j como banco de dados central foi motivada pelo fato de que o Neo4j implementa o modelo de grafo amplamente reconhecido e utilizado na indústria conforme Libkin, Martens e Vrgoč (2016).

Uma das grandes vantagens do Neo4j é a suíte de ferramentas incorporando um ecossistema que em conjunto é extremamente eficiente, as utilizadas frequentemente no desenvolvimento e no momento de análise foram:

- Neo4j Desktop: Ferramenta essencial que oferece uma interface gráfica que permite criar, configurar e gerenciar diversas instâncias do Neo4j como também diferentes projetos. Além de oferecer uma maneira conveniente de instalar servidores Neo4j, permite o acesso a outras ferramentas e extensões;
- Neo4j Bloom: Ferramenta de visualização interativa que permite explorar e visualizar os dados armazenados, projetado para permitir executar a ciência de dados, isto é, extrair informações dos dados de grafo;
- Neo4j Data Importer: Ferramenta que simplifica a tarefa de importar dados de fontes externas para uma instância de banco de dados, possibilitando mapear e migrar os dados de forma simples. Contudo, por ser uma ferramenta simples, existem outras maneiras de importar dados complexos e de grande escala; e
- Neo4j Browser: Interface de consulta interativa, permitindo que usuários executem consultas e visualizem os resultados.

Figura 21 – Exemplo de um código escrito em TypeScript

```
function soma(x: number, y: number): number {
    return x + y;
}

const resultado = soma(1, 2);
console.log(resultado); // 3
```

Fonte: Elaborada pelo autor.

3.1.1.2 Typescript

Conforme Microsoft (2023a), TypeScript é uma linguagem de programação de código aberto desenvolvida pela Microsoft, sendo um superconjunto sintático estrito de JavaScript, adicionando tipagem estática opcional à linguagem, como exemplificado na Figura 21.

Ao escolher a linguagem de programação e o ambiente a qual a ferramenta seria desenvolvida foi observado que não existia um OGM atualmente mantido para Javascript¹ uma vez que o Neode teve sua última atualização em 21 de fevereiro de 2023 conforme Cowley (2017), e nenhum construído usando Typescript.

Desta forma, os projetos que optarão por utilizar o OGM desenvolvido podem tirar o proveito do conceito de multiplataforma conforme definido em subseção 2.6.2. Além disso, a tipagem estática oferecida pelo Typescript, os usuários podem tirar proveito de ferramentas de autocompletas, documentação das funções e classes, além de identificar erros em tempo de desenvolvimento, o que é particularmente vantajoso em projetos de grande escala. Essa escolha de tecnologia visa atender à demanda por um OGM moderno e eficaz.

3.1.1.2.1 NodeJs

Foi decidido que o desenvolvimento do OGM seria utilizando o *runtime*² Node.js devido à sua ampla adoção e grande comunidade de desenvolvedores e pacotes disponíveis. No entanto, é importante ressaltar que a escolha do Node.js como runtime não impede a utilização de outros *runtimes* Javascript, como Bun e o Deno. Portanto, para os desenvolvedores poderem utilizar o OGM em qualquer ambiente de execução Javascript foi decidido utilizar a especificação de geração de código final “CommonJS“.

3.1.1.2.2 Mocha e Chai

Para a criação de suítes de teste, a metodologia de desenvolvimento orientado a testes (TDD) desempenha um papel fundamental no desenvolvimento de módulos e componentes

¹ Conforme Mozilla (2023), é uma linguagem de programação de *script* orientada a objetos e multiplataforma usada para tornar as páginas da Web interativas.

² Refere-se ao ambiente de execução no qual um programa é executado.

de software. Durante o processo de desenvolvimento, a utilização das ferramentas Mocha e Chai permite criar e executar testes automatizados.

O Mocha é um framework de testes para Javascript amplamente adotado, oferecendo uma estrutura flexível e extensível para organização e execução de testes unitários e de integração. Por sua vez, o Chai é uma biblioteca de afirmações, isto é, introduz um conjunto de comandos para testar assertividade. Em conjunto, essas ferramentas permitem que os desenvolvedores definam e verifiquem comportamentos esperados de maneira semântica e intuitiva.

Sendo adotado a metodologia TDD no desenvolvimento, com a implementação dessas ferramentas é possível criar um software mais robusto e confiável, favorecendo a detecção de erros e a manutenção dos comportamentos esperados do OGM.

3.1.1.2.3 Zod

Para a validação de esquemas foi escolhida a biblioteca Zod, devido ser uma biblioteca de validação Typescript de alto desempenho, que oferece uma abordagem fortemente tipada para definir e verificar esquemas de dados. Essa escolha baseia-se em critérios como tipagem estática, desempenho e extensibilidade.

3.1.2 Tecnologias para a API

Para construção da API foram cuidadosamente selecionadas uma variedade de tecnologias a fim de atender às necessidades do projeto. Tais tecnologias são descritas a seguir.

3.1.2.1 Nest.js

É um framework de código aberto, construído com Typescript cujo objetivo é facilitar o desenvolvimento de APIs, usando o paradigma de programação orientada a objetos, em que fornece uma estrutura de diretórios e arquivos predefinidos para organizar o seu projeto de forma lógica. Além disso, o framework inclui uma série de módulos e pacotes contendo características implementadas diminuindo o tempo de desenvolvimento.

O framework se baseia nos princípios e conceitos de orientação a objetos, conforme descritos na subseção 2.6.3, em que é construído usando arquitetura orientada a serviços para dividir a aplicação em pequenos serviços independentes, em conjunto com o sistema de roteamento e uma arquitetura de testes.

No contexto do NestJS, existem dois tipos principais de módulos no NestJS: módulos dinâmicos e módulos comuns

- Módulos comuns: São usados para agrupar controladores, serviços e provedores, usando o decorador³ “@Module” para definir como um módulo; e

³ É uma função que recebe uma classe ou função como parâmetro.

- Módulos dinâmicos: Permitem a flexibilidade na organização do código, ao poder registrar estes módulos, usado para extensões que podem ser adicionados ou removidos durante a execução da aplicação. Para isto é necessário definir o método “forRoot” para configurar as dependências do módulo.

3.1.2.2 PostgreSQL

É um SGDB relacional de código aberto, conhecido por sua confiabilidade e capacidade de estender suas funcionalidades. Além disso, o PostgreSQL é altamente compatível com ferramentas do ecossistema relacional.

3.1.2.3 PrismaJS

Tecnologia escolhida para representar ORM a ser utilizado em conjunto com o banco de dados PostgreSQL previamente mencionado, também sendo uma ferramenta de código aberto, que oferece uma camada de abstração capaz de interagir com o banco de dados.

É uma ferramenta que suporta vários banco de dados relacionais, dentre eles o PostgreSQL que foi selecionado na subseção 3.1.2.2, que em conjunto com suas interfaces de tipagem e servidor para geração de módulos para o clientes, sua integração com a aplicação é eficaz. Além disso, o PrismaJS oferece migrações de banco de dados a partir de um único arquivo, tornando uma escolha presente em quase todos os projetos conforme Prisma (2023a).

3.1.3 Conjunto de dados

Um conjunto de dados é uma coleção organizada de informações ou observações relacionadas a um tópico, ou problema específico (NEOWAY, 2023). Sendo fundamentais para a análise de dados, pesquisa e tomada de decisões, em que, podem conter uma variedade de tipos de dados como texto, números, imagens, áudio e outros formatos.

A princípio, considerou-se o uso de um conjunto de dados que continha informações sobre acidentes catalogados pelo INSS. Este conjunto de dados possuía um tamanho de 12 GB e incluía campos como localização do acidente, estado empregatício, nomes das pessoas envolvidas, descrição do acidente e a data em que ocorreu. No entanto, houve desafios com esse conjunto de dados, uma vez que nem todos os campos estavam preenchidos e o volume de dados não ultrapassava dois milhões de registros, para simular um contexto de *big data*.

3.1.3.1 *eCommerce behavior data from multi-category store*

É um conjunto de dados disponível na plataforma Kaggle⁴ composto por 285 milhões de eventos de usuários de uma plataforma de vendas multimarcas. A plataforma não deixa

⁴ Disponível em <https://www.kaggle.com/datasets/mkechinov/ecommerce-behavior-data-from-multi-category-store?select=2019-Oct.csv>, acesso em 29 de out. de 2023

disponível todos os arquivos para *download*, contudo o autor Kechinov (2023), disponibiliza o conjunto todo, que no formato comprimido equivale a 16,45 GB, contudo quando descomprimido os arquivos equivalem a 65,35 GB.

O conjunto de dados apresenta os campos: “event_time”, “event_type”, “product_id”, “category_id”, “category_code”, “brand”, “price”, “user_id” e “user_session”. Representados na Tabela 1.

Tabela 1 – Trecho do conjunto de dados

event_time	event_type	product_id	category_id	category_code	brand	price	user_id	user_session
2019-10-01 00:00:00 UTC	view	44600062	2,10380745959539E+018		shiseido	35.79	541312140	72d76fde-8bb3-4e00-8c23-a032dfed738c
2019-10-01 00:00:00 UTC	view	3900821	2,05301355232677E+018	appliances.environment.water_heater	aqua	33.20	554748717	9333dfbd-b87a-4708-9857-6336556b0fc

Fonte: Elaborado pelo autor.

A escolha deste conjunto de dados foi baseada em seu volume de dados, integridade e análises que poderiam ser obtidas no contexto de ciência de dados, tornado-o uma opção mais adequada para análises e experimentações do projeto.

No conjunto, cada atributo desempenha um papel específico e fornece informações cruciais. O campo “event_time” registra o horário em que o evento ocorreu no formato UTC, permitindo rastrear o tempo dos eventos, “event_type” descreve o tipo de evento que ocorreu, identificando as diferentes ações que os usuários realizaram, que incluem: “remove_from_cart”, “purchase”, “view” e “cart”. O “product_id” serve como um identificador para os produtos em questão. O “category_id” é responsável por associar cada produto à sua categoria correspondente, enquanto “category_code” oferece uma taxonomia detalhada das categorias dos produtos. A informação sobre a “brand” indica a marca dos produtos em letras minúsculas. O atributo “price” revela o preço dos produtos, fornecendo dados essenciais para análises financeiras. O “user_id” é um identificador para cada usuário, permitindo o acompanhamento das atividades de cada um. Por fim, “user_session” representa o identificador de uma sessão temporária de um usuário, auxiliando na compreensão do comportamento dos usuários durante sessões específicas.

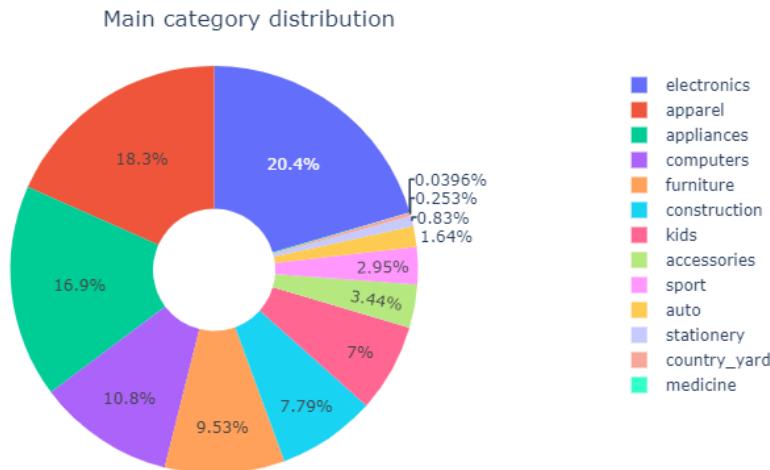
3.1.3.1.1 Análise do conjunto de dados

A análise do conjunto de dados colabora para criação e modelagem do modelo lógico do banco de dados. Desta forma, foram feitas algumas análises usando dois dos seis arquivos do conjunto de dados em conjunto com as análises de lu (2022).

Para compreender a distribuição dos produtos no conjunto de dados, foi realizada uma análise que considera a quantidade para as principais categorias. Vale ressaltar que o conjunto de dados apresenta uma estrutura taxonômica de categorias, em que cada categoria é representada por um ou mais termos separados por ponto. Essa estrutura permite uma categorização mais granular dos produtos em categorias e subcategorias. No entanto, 56,14%

dos produtos não têm uma categoria definida. Desta forma, a Figura 22 ilustra a distribuição de produtos únicos sobre as categorias, demonstrando haver uma discrepância nos produtos, que são interagidos além da distribuição de produtos.

Figura 22 – Distribuição dos produtos únicos nas categorias principais.



Fonte: (LU, 2022).

Para distribuir as categorias e subcategorias em uma visualização que permite identificar os conjuntos e suas proporções no conjunto de dados é necessário o gráfico de mapa de árvore, permitindo identificar às áreas de maior e menor concentração de produtos. Desta forma a Figura 23 colabora a visualização das categorias tão quanto suas subcategorias.

Figura 23 – Gráfico de mapa de árvore das categorias e subcategorias

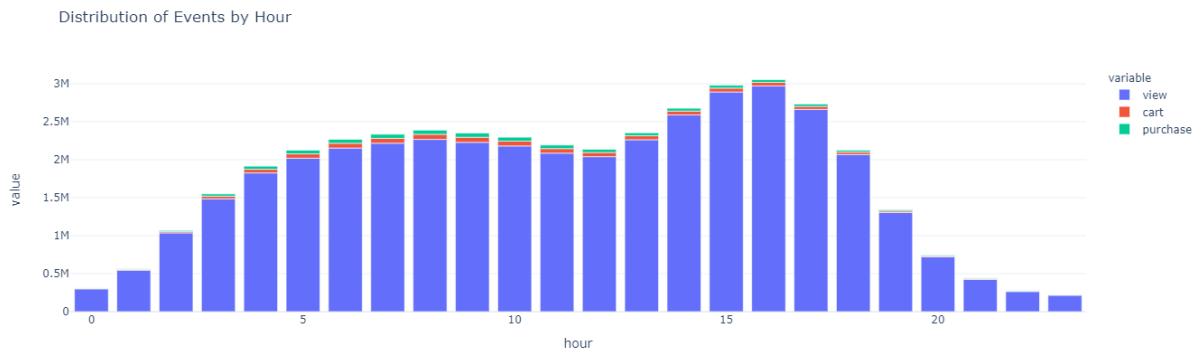


Fonte: (LU, 2022).

Além das análises nominais, investigar a distribuição de eventos ao longo do dia gera um gráfico que exibe a quantidade de eventos registrados em diferentes horas do dia, podendo

otimizar estratégias de marketing como também planejar expansão horizontal nos serviços para suportar o tráfego, assim como demonstra a Figura 24.

Figura 24 – Gráfico de barras usando agrupamentos dos eventos por horário do dia



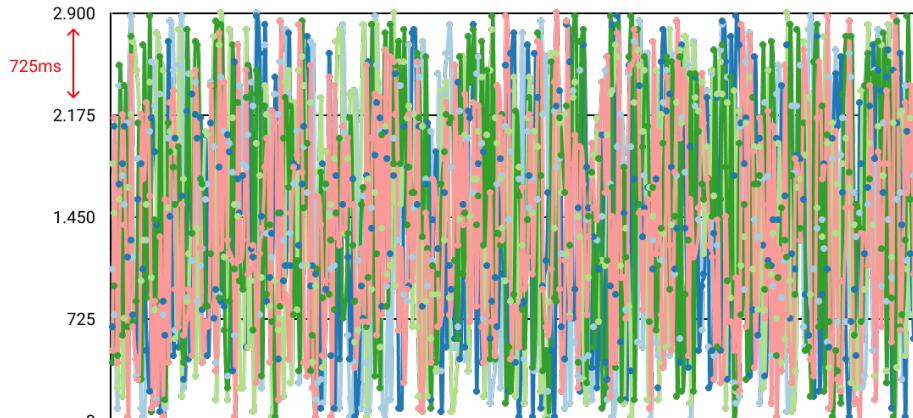
Fonte: (LU, 2022).

3.2 Desenvolvimento do OGM

Conforme conceitualizado em seção 2.6, segundo Pressman e Maxim (2021) e a análise de Brooks (2018), o desenvolvimento do OGM envolveu uma fase de planejamento detalhada, anterior ao desenvolvimento, que incluiu a avaliação de diversas opções de OGM e ORM (do inglês, *object relational mapping*) para identificar características críticas nas implementações. Posteriormente, foram criados diagramas que delinearam as expectativas em relação a um OGM eficaz, bem como fluxogramas iniciais que detalharam os processos e módulos.

A fase de pesquisa para o Quadro 3 foi conduzida, considerando critérios cruciais para a escolha das tecnologias adequadas, envolvendo a avaliação de diversos pacotes de OGM e ORM, com foco na análise de campos como linguagem de programação, suporte a diferentes bancos de dados, capacidade de criação de esquemas, desempenho, disponibilidade de ferramentas auxiliares e, não menos importante, a popularidade desses pacotes na comunidade de desenvolvimento de software. O aspecto de desempenho foi avaliado por meio da execução de 300 consultas de busca, atualização e inserção, usando o cálculo da média do tempo de execução para cada tipo de consulta e após isso foi separado em três níveis diferentes: baixo, regular e alto. Para identificar qual nível corresponde, foi calculada a diferença entre cada um dos tempos de execução e retirada a média para cada tipo de consulta, traçando linhas a cada porção correspondente a média da diferença do tempo de execução, conforme mostra a Figura 25.

Figura 25 – Variação de tempo durante 300 buscas por ID.



Fonte: Elaborado pelo autor.

É possível verificar a alta variação na Figura 25 entre as ferramentas durante as 300 buscas executadas, contudo algumas ferramentas apresentaram médias abaixo de 1.450ms, que as classificaram como alto desempenho, enquanto também houve ferramentas apresentando médias entre 2.175ms e 1.450ms, sendo rotuladas como regulares, e uma média demasiadamente acima de 2.175ms classificada como baixa, em Quadro 3.

Quadro 3 – Características de mapeadores encontrados

Característica	Sequelize	TypeORM	Hibernate	Neode	Neo4jOGM
Linguagem	TypeScript	TypeScript	Java	JavaScript	Java
Banco de dados	Relacional	Relacional	Relacional	Neo4j	Neo4j
Criação de esquema	Sim	Sim	Sim	Não, DDL é externa	Sim
Desempenho	Regular	Baixa	Alta	Regular	Alta
Ferramentas	Sim	Sim	Sim	Manual	Sim
Popularidade	Alta	Alta	Alta	Baixa	Alta

Fonte: Elaborado pelo autor.

Analizando o mercado de ferramentas, tanto no contexto de ORM quanto OGM, são identificadas características em comum que são cruciais para a popularidade e eficiência dessas soluções. Uma dessas características é a capacidade de definição de esquema, além do que, algumas ferramentas também implementam o conceito de migrações, facilitando a evolução do esquema ao longo do tempo de maneira controlada e rastreável. Além disso, a integração simples com um ecossistema mais amplo, incluindo outros *frameworks*⁵ e bibliotecas, é uma característica essencial para garantir que as ferramentas de OGM e ORM possam ser incorporadas em projetos. Essas características nortearam as expectativas no desenvolvimento da aplicação, colaborando na listagem de características a ser implementadas:

- Definição de esquema;

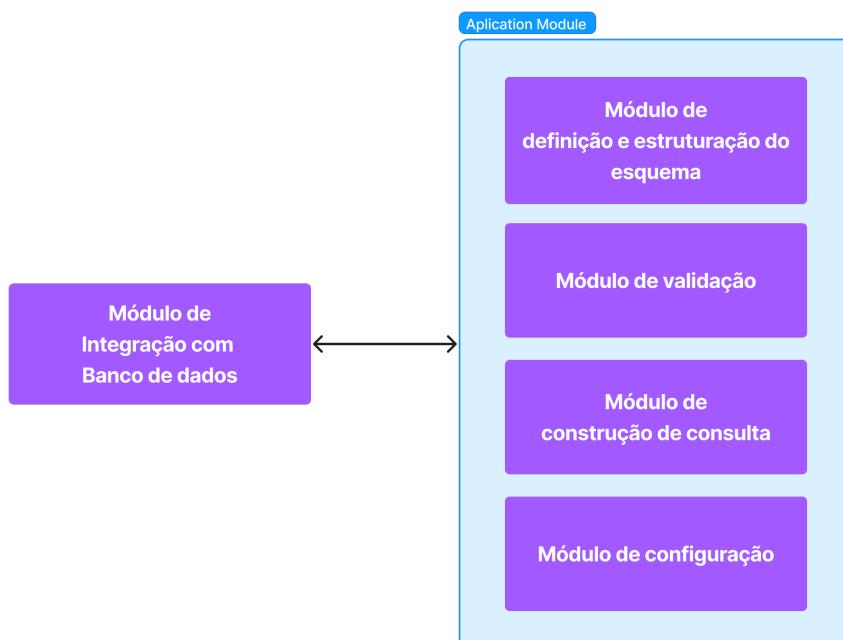
⁵ Um framework é a estrutura de software que fornece uma base para o desenvolvimento de aplicações, forçando uma estrutura para o desenvolvimento, conforme Gackenheimer (2015).

- Validação dos dados;
- Possibilidade de injeção de dependência;
- Abstração de consultas, bem definida;
- Tratamento de erros; e
- Configuração do ambiente dinamicamente.

Considerando a seção 2.6 e as análises dos OGM e ORM existentes, foi identificada a necessidade de incorporar a característica de validação de dados ao OGM desenvolvido. Notavelmente, nenhum dos OGMs encontrados durante a pesquisa demonstrou a capacidade de realizar uma validação durante a execução do programa ou permitir a injeção de dependência para esse processo da validação. Em contraste, no contexto dos ORMs, a comunidade de desenvolvedores elencou frequentemente a validação de dados como uma tarefa desnecessária a ser integrada, devido à facilidade de implementá-la nos clientes que utilizam os dados dos objetos mapeados. Portanto, a integração da validação de dados no OGM surgiu como uma decisão estratégica, contribuindo para a robustez e a confiabilidade do OGM desenvolvido, ao garantir a integridade e consistência nos dados manipulados em seu ambiente.

Continuando o processo de planejamento, conforme delineado na subseção 2.6.4, foram elencados os principais módulos que seriam desenvolvidos para atender às expectativas estabelecidas. Cada um desses módulos desempenha um papel específico e crítico no funcionamento do OGM desenvolvido.

Figura 26 – Todos os módulos presentes no OGM



Fonte: Elaborada pelo autor.

É possível verificar na Figura 26, que a aplicação foi separada em duas seções, os módulos de interação externos, como o módulo de interação com o banco de dados, e a seção de módulos de integração com a aplicação, que incluem: módulo de configuração, módulo de construção de consulta, módulo de validação e módulo de definição e estruturação do esquema.

3.2.1 Módulo de configuração

O módulo de configuração é responsável por facilitar a configuração dinâmica do sistema e a leitura de variáveis de ambiente essenciais para o funcionamento correto de todos os módulos. Este módulo permite que informações, como credenciais de conexão e configurações do projeto, sejam adquiridas de maneira flexível e segura, garantindo a propagação dessas configurações a todos os outros módulos do OGM.

Para o desenvolvimento deste módulo foi usada a leitura direta de variáveis de ambiente no ambiente de execução, acessando a variável processo representada na Figura 27.

Figura 27 – Enter Caption

```
const connectionString =
  process.env.NEO4J_CONNECTION_STRING ??
    `${process.env.NEO4J_PROTOCOL}://${process.env.NEO4J_HOST}:${
      process.env.NEO4J_PORT
    }/${process.env.NEO4J_DATABASE ?? "neo4j"}`,
  username = process.env.NEO4J_USERNAME,
  password = process.env.NEO4J_PASSWORD;
```

Fonte: Elaborada pelo autor.

3.2.2 Módulo de integração com o banco de dados

O módulo de integração com o banco de dados representa um dos pilares fundamentais da aplicação desenvolvida, sendo responsável por estabelecer e gerenciar a conexão com o banco de dados subjacente, bem como por realizar operações de leitura e escrita necessárias. Esse módulo proporciona uma interface flexível que simplifica a interação entre o sistema e o banco de dados, em que a interface do módulo, que consiste em apenas uma classe, que oferece funcionalidades essenciais, como a criação de transações, assim possibilitando a execução de operações atômicas, bem como a criação de sessões em modo de leitura e escrita, além de permitir inicializar e encerrar a conexão com o banco de dados.

A classe expressa na Figura 28 utiliza o *driver* mantido pela própria Neo4j, que oferece suporte para a interação com banco de dados. Além disso, a classe adquire informações das variáveis de ambiente por meio do módulo de configuração (3.2.1), permitindo a configuração flexível e dinâmica das opções de conexão e autenticação, adaptando-se às necessidades de diferentes projetos e ambientes. A capacidade de ajustar essas variáveis de ambiente oferece uma camada adicional de personalização e adaptável a uma variedade de cenários, como configuração de múltiplas instâncias.

Figura 28 – Diagrama de classe do módulo de interação com o banco de dados



Fonte: Elaborada pelo autor.

Na Figura 28 é possível visualizar a classe com seus métodos públicos e privados, além de seus atributos, em que se pode destacar os métodos “transaction” e “cypher”. Os quais são responsáveis por executar uma sequência de comandos com possibilidades de definir modos de isolamento, e executar consultas, respectivamente.

As funções implementadas neste módulo foram projetadas para atingir o menor nível de complexidade, além de seguir os princípios e conceitos preconizados por Pressman e Maxim (2021). Usando a abordagem orientada a testes, na qual o comportamento esperado foi definido antes do desenvolvimento.

Figura 29 – Definição da função que cria e executa uma transação

```
/**  
 * Create a transaction.  
 */  
transaction(): Transaction & { success: () => Promise<void> } {  
    const session = this.#driver.session();  
  
    const transaction: Transaction & { success: () => Promise<void> } =  
        session.beginTransaction();  
  
    transaction.success = async () => {  
        await transaction.commit();  
        session.close();  
    };  
  
    return transaction;  
}
```

Fonte: Elaborada pelo autor.

É possível observar na implementação do método “transaction” na Figura 29, o qual ocorre a criação de uma sessão a partir do *driver* de acesso ao banco de dados, na sequência é anexado um fluxo de controle padronizado chamada “etapa de finalização”, que ao finalizar a transação efetivará automaticamente as mudanças usando a cláusula “COMMIT” e finaliza a conexão com o banco de dados para a sessão.

Figura 30 – Definição da função que executa as sentenças da linguagem de consulta

```

/**
 * Run a cypher query
 */
cypher(
  query: string,
  params: Record<string, any> = {},
  session: false | Session = false
) {
  const driver = session ? session : this.readSession();

  return driver
    .run(query, params)
    .then((response) => {
      if (!session) driver.close();

      return response;
    })
    .catch((error) => {
      if (!session) driver.close();

      error.query = query;
      error.params = params;

      throw error;
    });
}

```

Fonte: Elaborada pelo autor.

O método “cypher” implementado na Figura 30 executará uma consulta Cypher em que recebe uma cadeia de caracteres com a possibilidade de haver argumentos e variáveis usando o marcador \$ precedendo o nome da variável no conteúdo anexado ao parâmetro “query”, desta forma o método verifica se existe uma sessão presente se não cria uma nova sessão, na sequência utilizando o *driver* coloca para executar passando os valores presentes nas variáveis do parâmetro “query”. Tirando proveito da estrutura de “Promise” o método implementa que ao finalizar com sucesso a consulta, finaliza-se a sessão retornando a resposta, contudo, se acontecer algum erro, é enviado uma exceção com o erro ocorrido, além de finalizar a sessão.

3.2.3 Módulo de definição e estruturação do esquema

O módulo de definição e estruturação do esquema representa um dos pilares fundamentais do OGM desenvolvido, por desempenhar o papel crítico na gestão de esquemas de dados e na estruturação de entidades no contexto do modelo de dados baseado em grafo, em que envolve identificar os atributos e tipos definidos no modelo, com o propósito de aprimorar

a consistência de dados em tais modelos. A implementação deste módulo envolveu um planejamento adicional, uma vez que a definição e a estruturação de esquemas são tarefas de extrema importância para garantir a integridade e a coerência dos dados no sistema. Conforme destacado na pesquisa no Quadro 3, a funcionalidade de definição e estruturação do esquema foi identificada com uma das características essenciais em mapeadores, sido abordada com particular atenção.

Para a estruturação deste módulo, foi realizada uma análise das opções de ORM e OGM que ofereciam suporte à funcionalidade de definição de esquema, visualizando as opções de código aberto e suas implementações. Esse processo visava identificar a implementação mais adequada que seria conveniente para os desenvolvedores. Após a análise, ficou claro que a implementação de um arquivo de esquema se destacava com a solução mais abordada.

3.2.3.1 Linguagem de definição

A abordagem do arquivo de esquema consiste em permitir que os desenvolvedores definam de maneira simples as entidades, representadas como modelos de nós e relacionamentos no modelo de dados baseado em grafo, bem como os tipos de dados associados a seus atributos, por meio de uma linguagem semanticamente aceita pelo mapeador. Isso é alcançado por meio de um arquivo específico, ou múltiplos arquivos, que permitem a declaração de entidades, relacionamentos e atributos de forma estruturada (PRISMA, 2023b).

Nesta etapa, foi desenvolvida uma linguagem de definição intermediária ao banco de dados, visando proporcionar uma forma legível de descrever a estrutura do esquema do banco de dados, abstraindo a complexidade inerente à configuração de um banco de dados de grafo. Além de simplificar o processo de desenvolvimento, pode servir como documentação do projeto assim como funciona o modelo lógico de um banco de dados relacional.

Conforme a seção 2.5, a criação da linguagem envolveu uma série de considerações que visavam simplificar o processo de configuração e torná-lo o mais intuitivo e claro possível. Desta forma, a linguagem foi projetada com a intenção de seguir o padrão amplamente utilizado nos ORM pesquisados, o que facilitaria a transição de desenvolvedores familiarizados com os sistemas de mapeamento relacional para o modelo de mapeamento baseado em grafo. Como resultado, cada modelo, representando um tipo de nó ou relacionamento, é definido claramente, seguindo o padrão na Figura 31.

Figura 31 – Estrutura da DDL criada.

(a) Exemplo da estrutura de definição de um nó.

```
Node Usuário {
    nome:      String
    salário:   Decimal
}
```

(b) Exemplo da estrutura de definição de um relacionamento.

```
Relationship Event {
    time:        Date
    type:        Enum { "view", "cart", "remove_from_cart", "purchase" }
    user_session: String
    user:         User
    product:     Product
}
```

Fonte: Elaborada pelo autor.

Essa abordagem permite que os desenvolvedores definam entidades intuitivamente, especificando atributos, tipos de dados e, quando aplicável, operadores e parâmetros adicionais. Por exemplo, para definir um nó do tipo “Usuário” com um atributo “id” que é um identificador com um incremento automático do tipo inteiro, utiliza-se a seguinte definição, apresentado na Figura 32.

Figura 32 – Exemplo de definição de um nó do tipo “Usuário” com um atributo “id” com incremento automático em um tipo inteiro

```
Node Usuário {
    id: Int @identifier(auto: true)
}
```

A sintaxe foi cuidadosamente planejada para oferecer expressividade e praticidade aos desenvolvedores, permitindo a definição flexível de atributos em tipos de nós e relacionamentos no modelo de dados baseado em grafo. Essa abordagem simplificada permite que os desenvolvedores especifiquem atributos de acordo com suas necessidades, considerando a obrigatoriedade, a opcionalidade e a multiplicidade de cada atributo. Para alcançar essa flexibilidade, foram adotados sufixos específicos para representar os diferentes estados dos atributos, conforme descrito a seguir:

- “?” Indica um atributo com valor opcional; e
- “[]” Indica um atributo com múltiplos valores.

Além disso, a linguagem teve implementação ao suporte a diversos tipos de dados, incluindo:

- Booleano;
- UUID (identificador único universal);
- Enumeradores;
- Cadeia de caracteres;
- Inteiro;
- Número com ponto flutuante;
- Datas (intervalo de tempo, data e tempo de duração);
- Ponto; e
- Coordenada.

A linguagem também inclui operadores específicos que enriquecem as definições de atributos, como:

- “Unique”: Garante a unicidade de um atributo, evitando a duplicação de valores sobre o mesmo modelo de dados, isto é, quando um nó ou relacionamento com uma etiqueta (identificador do tipo do modelo), todos os nós, ou relacionamentos que tenham esse conjunto de etiquetas, tem valores distintos neste atributo, contudo, um valor opcional pode ser único, mas só é validado após atribuição; e
- “Identifier”: Operador define o atributo como identificador dos nós no nível de esquema.

Embora o banco de dados forneça identificadores internos para os objetos, para os nós, o operador “identificador” permite que os desenvolvedores especifiquem um atributo exclusivo que pode ser usado para identificar um modelo de nó de maneira eficaz durante buscas e consultas, independente dos identificadores internos do banco de dados. Isso proporciona um maior controle sobre a identificação dos nós. Dessa forma, a linguagem projetada, permite que os desenvolvedores personalizem as definições de dados de acordo com as necessidades específicas do projeto.

A implementação do tipo de dados enumerador (enum) foi conduzida com base nos conceitos adotados em linguagens de programação modernas, principalmente em que incorporam as chamadas funções lambda. Com o intuito de oferecer uma maneira simples e expressiva de identificar os valores suportados pelos enums, especificados pelos usuários, introduzindo um

formato que se baseia em um conjunto de chaves “{}” para delimitar os valores permitidos, os quais são separados por vírgulas. A linguagem aceita a especificação de valores enum com suporte para dois tipos de dados: cadeia de caracteres (*string*) e inteiros.

Segundo a seção 2.5 a implementação do interpretador para essa linguagem requereu a criação do processo de análise léxica e sintática, isto é, o *lexer* e *parser* capazes de entender a linguagem e transformá-la em objetos de definição para o mapeador.

3.2.3.1.1 Analisador Léxico

Desempenhando o papel de interpretar no processamento do arquivo de esquema. Sua principal responsabilidade é percorrer o arquivo e atribuir *tokens* aos elementos identificados.

O lexer contém expressões regulares e definições em estruturas de dados que o ajudam a reconhecer e classificar os elementos presentes na linguagem. Essas definições incluem identificar palavras reservadas, literais, identificadores, separadores, operadores, espaços em branco e comentários. Considerando os destaques na subseção 2.5.2.1.1 as palavras reservadas incorporadas incluem:

- Tipos de dados possíveis: “boolean”, “UUID”, “enum”, “string”, “integer”, “decimal”, “datetime”, “date”, “timestamp”, “localtime”, “point” e “local”;
- Palavras-chave como “Node”, “Relation” usadas para definir os tipos de nós e relacionamentos;
- Nome de funções usadas como operadores na linguagem; e
- Nomes especiais como “in”, “out” e “both”, usados para especificar a direção de relacionamentos.

Figura 33 – Trecho da definição dos identificadores dos literais.

```

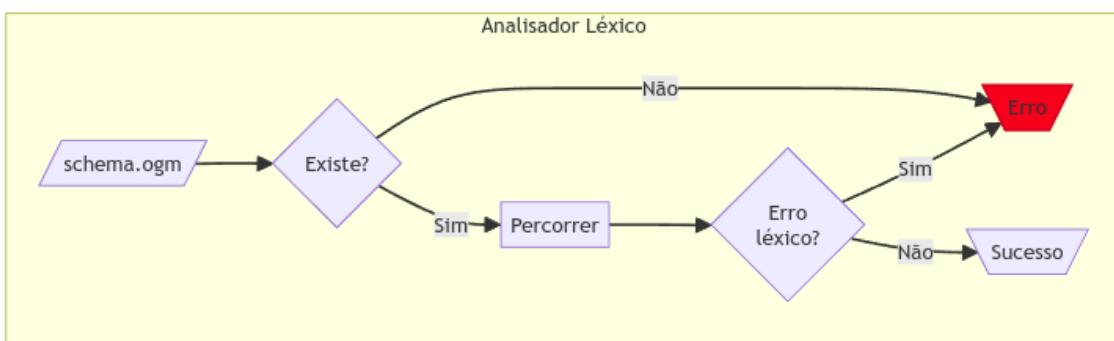
constants: [
    // UUIDv4
    createToken({
        name: "UUIDLiteral",
        pattern: /\w{8}-\w{4}-\w{4}-\w{4}-\w{12}/,
    }),
    createToken({ name: "BooleanLiteral", pattern: /true|false/i }),
    createToken({ name: "StringLiteral", pattern: /'(.*?)?(?=')/ },
    createToken({ name: "StringLiteral", pattern: /"(.*?)(?=")/ },
    createToken({ name: "IntegerLiteral", pattern: /\d+/ },
    createToken({ name: "DecimalLiteral", pattern: /\d+\.\d+/ },
    createToken({ name: "DateLiteral", pattern: /\d{4}-\d{2}-\d{2}/ },
    createToken({
        name: "DateTimeLiteral",
        pattern: /\d{4}-\d{2}-\d{2}\T\d{2}:\d{2}:\d{2}/,
    }),
    createToken({ name: "TimeLiteral", pattern: /\d{2}:\d{2}:\d{2}/ },
    createToken({
        name: "LocationLiteral",
        pattern: /\d{1,3}°\d{1,2}'\d{1,2}"[NS],\s\d{1,3}°\d{1,2}'\d{1,2}"[EW]/,
    }),
],

```

Fonte: Elaborada pelo autor.

O analisador léxico é projetado para percorrer o arquivo de esquema e, com base nas definições estabelecidas, atribuir tokens a cada elemento identificado. Esses tokens são posteriormente utilizados no processo de análise sintática e semântica para interpretar e validar as definições de esquema. Assim como mostra o fluxograma apresentado na Figura 34.

Figura 34 – Fluxograma simplificado do processo executado pelo analisador léxico.

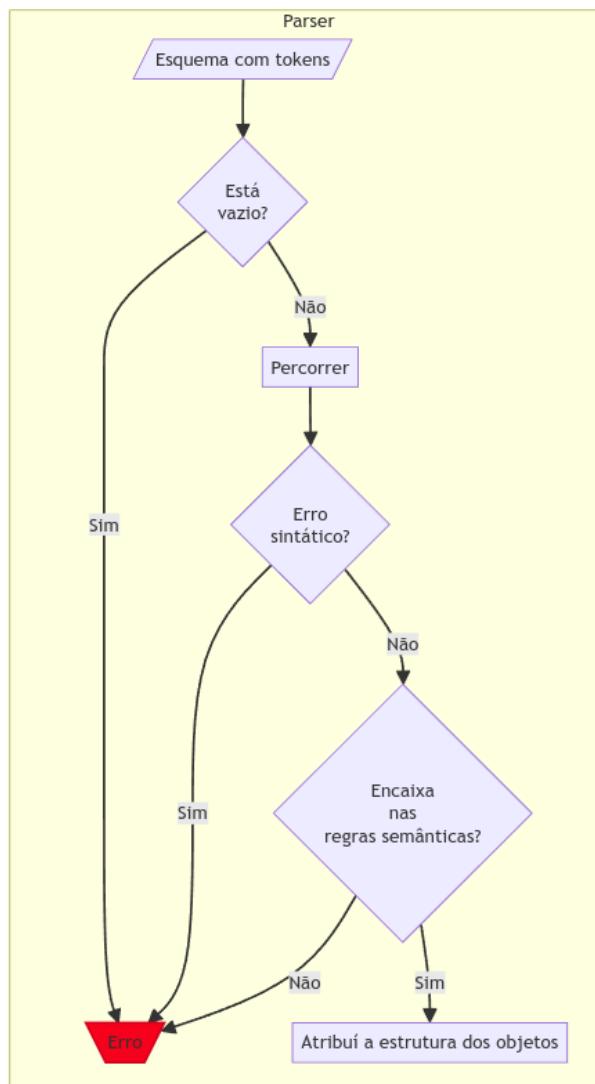


Fonte: Elaborado pelo autor.

3.2.3.1.2 Analisador sintático e regras semânticas

Responsável por interpretar e validar as regras do arquivo de esquema definidas na linguagem. Esse componente representado pela classe “Parser” realiza o fluxograma representado na Figura 35, que descreve a análise sintática como concomitante a criação de objetos correspondentes a todos os modelos de nó e relacionamento, bem como seus atributos, incluindo seus tipos de dados.

Figura 35 – Fluxograma simplificado do processo executado pelo parser.



Fonte: Elaborado pelo autor.

A abordagem de desenvolvimento adotado pelo analisador segue um modelo *top-down* que em partes entra em conformidade com Frost, Hafiz e Callaghan (2007), em que as regras sintáticas definidas na linguagem são correspondidas por funções específicas, que extraem as informações relevantes conforme são identificadas como aceitas pela estrutura. Essas funções são estruturadas para refletir as regras da linguagem de forma organizada e coerente, permitindo que a análise seja conduzida de maneira sistêmica.

Durante o processo da análise sintática, o parser percorre uma estrutura de dados similar ao arquivo de esquema, só que com os *tokens* já inseridos e aplica as funções que descreve a estrutura correta, construindo objetos que representam os modelos de nó e relacionamento especificados pelos desenvolvedores. As informações coletadas incluem os nomes dos nós e relacionamentos, bem como a definição de seus atributos, incluindo os tipos de dados associados a cada um deles, conforme ilustrado na Figura 36.

Figura 36 – Árvore sintática construída

```

→ parser.schema
  ↘ {nodes: Map(10), relations: Map(1)}
    ↘ nodes: Map(10) {size: 10, User => {identifier: ..., ...}, Pilot => {identifier: ..., ...}, Fly_Attendant => ..., Ticket => ..., Company => ..., ...}
      ↘ 0: {"User" => Object}
        key: 'User'
        ↘ value: {identifier: 'User', properties: {...}}
          identifier: 'User'
            ↘ properties: {id: ..., cpf: ..., email: ..., username: ..., password: ..., ...}
              > address: {type: 'string', primaryKey: false, required: true, options: ...}
              > birth_at: {type: 'date', primaryKey: false, required: true, options: ...}
              > cpf: {type: 'string', primaryKey: false, required: true, options: ...}
              > created_at: {type: 'date', primaryKey: false, required: true, options: ...}
              > email: {type: 'string', primaryKey: false, required: true, options: ...}
              > id: {type: 'UUID', primaryKey: true, required: true, options: ...}
              > name: {type: 'string', primaryKey: false, required: true, options: ...}
              > password: {type: 'string', primaryKey: false, required: true, options: ...}
              > rg: {type: 'string', primaryKey: false, options: ...}
              > sex: {type: 'enum', values: Array(3), primaryKey: false, required: true, options: ...}
              > tickets: {type: 'relation', node: 'Ticket', primaryKey: false, multiple: true, required: true, ...}
              > updated_at: {type: 'date', primaryKey: false, required: true, options: ...}
              > username: {type: 'string', primaryKey: false, required: true, options: ...}
            >
  >

```

Fonte: Elaborada pelo autor.

É possível observar na Figura 36 que as regras sintáticas criadas segmentam a árvore, no qual a cada filho apresenta uma nova regra interna ou terminal, com o valor correspondente.

3.2.3.2 Mapa de modelos

A classe responsável pelo mapa de modelos é uma extensão personalizada de um objeto comum da linguagem, enriquecida com funcionalidades presentes na estrutura de dados “Map” conforme na Figura 37. Essa classe é projetada para operar com os modelos obtidos a partir do esquema, permitindo que os desenvolvedores recuperem os modelos das entidades mapeadas, usando os identificadores de cada nó ou relacionamento.

Figura 37 – Classe “ModelMap”

ModelMap
-#app -#models
+has(name) +get(name) +set(name, model) +keys() +delete(name) +forEach(callback) +clear() +schema() +attributes()

Uma vez os modelos terem sido mapeados e incorporados ao mapa, o desenvolvedor pode recuperar o modelo conforme a Figura 38.

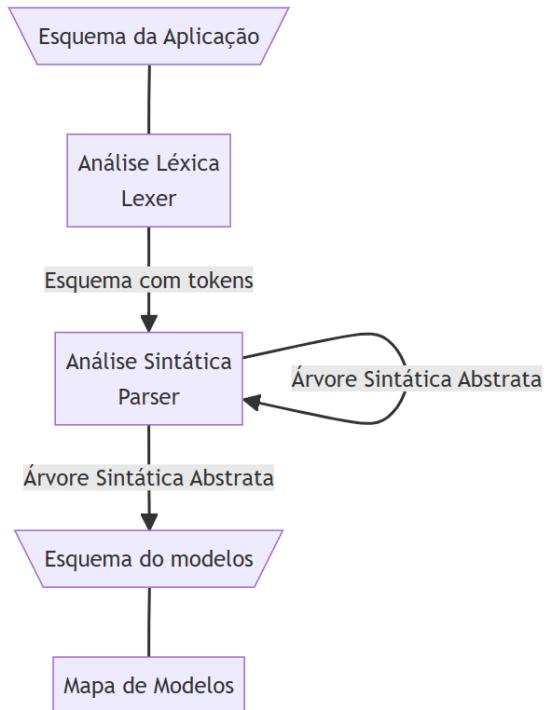
Figura 38 – Exemplo de como recuperar um modelo.

```
import OGM from 'ogm'  
  
OGM.retrieveModel<T>('User');
```

Fonte: Elaborado pelo autor.

Desta forma o fluxograma deste módulo pode ser expresso como na Figura 39.

Figura 39 – Fluxograma do módulo de definição e estruturação do esquema



Fonte: Elaborado pelo autor.

Portanto, ao receber o arquivo do esquema da aplicação como descrito na subseção 3.2.3.1.1, o “Lexer” é responsável por atribuir *tokens* ao arquivo da aplicação, na sequência os *tokens* são interpretados pelo “Parser” utilizando os métodos descritos em subseção 3.2.3.1.2 elaborando a construção da árvore sintática abstrata recursivamente assim obtendo o esquema dos modelos.

3.2.4 Módulo de construção de consulta

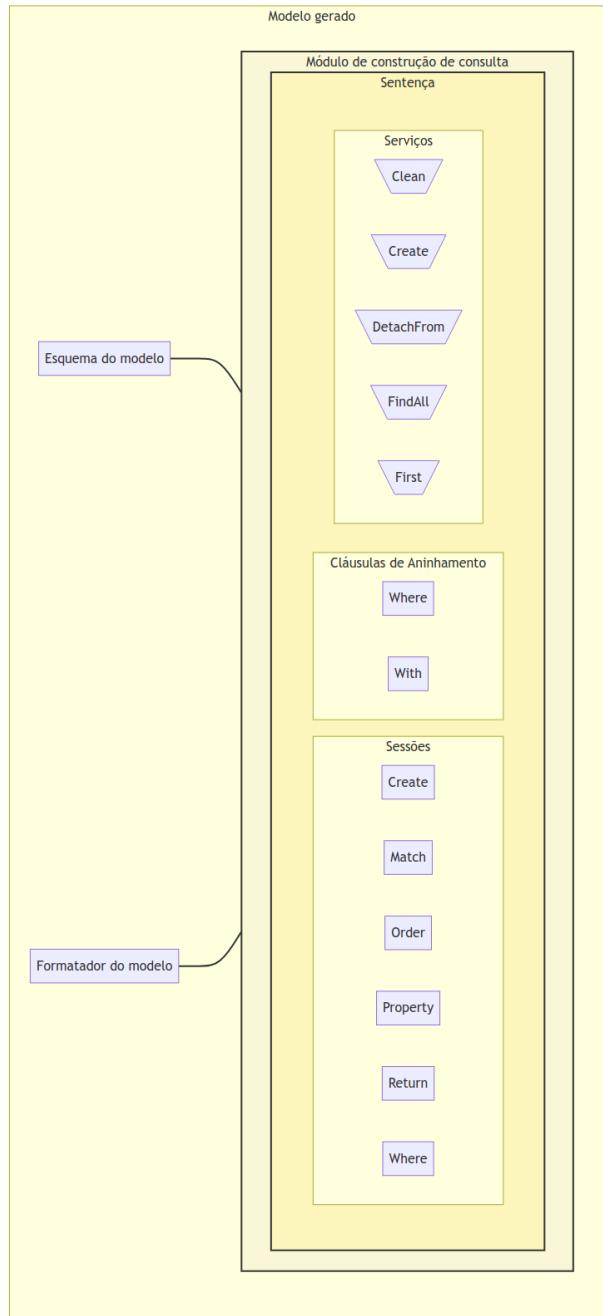
Como parte fundamental do sistema, projetado para permitir a criação de consultas eficazes e personalizadas para o banco de dados. Esse módulo foi construído com base em referências de mercado, como o Knex.js⁶ e o PrismaJS, os quais são amplamente reconhecidos como ferramentas de referência para a construção de consultas em sistemas de gerenciamento de banco de dados relacional, contudo, a abordagem adotada foi adaptada às necessidades do sistema, que opera um modelo de dados baseado em grafo e utiliza a linguagem de consulta Cypher.

O processo de construção de consultas é projetado como um processo recursivo, no qual uma consulta começa vazia e, à medida que o usuário executa operações e expressa as necessidades, cláusulas são adicionadas à consulta.

⁶ É um construtor de consultas otimizadas para PostgreSQL, CockroachDB, MSSQL, MySQL, MariaDB, SQLite3, Oracle e Amazon Redshift, disponível em <https://knexjs.org>.

Além disso, o módulo de construção de consulta é projetado para ser altamente adaptável e extensível, possibilitando a incorporação de novas cláusulas e operações à medida que o sistema e a linguagem evoluí, proporcionando uma base sólida para aprimoramento e alterações necessárias. Os componentes do módulo podem ser visualizados pela Figura 40.

Figura 40 – Principais componentes do módulo de construção de consulta



Fonte: Elaborada pelo autor.

Na Figura 40 é possível visualizar que o esquema e o formatador o modelo gerado são partilhados com o módulo de geração de consulta, e há presença de três componentes na abstração de sentença, que incluem: serviços, cláusulas de aninhamento e sessões. Com exceção do componente de serviços, todos os componentes internos de sentença, são classes

que possuem sua forma como abstração de funções que recebem a sentença como parâmetro, podendo se interferir umas as outras conforme são adicionadas.

3.2.4.1 Classe de construção de consultas

A classe de construção de consultas, com o nome de “QueryBuilder”, é a responsável por permitir a criação e manipulação de consultas de forma estruturada, esse processo é facilitado pelo uso de uma estrutura de pilha, na qual as operações de consulta são tratadas como funções empilhadas, permitindo que as sentenças criadas interajam.

As funções principais presentes em “QueryBuilder” abrangem as principais sentenças de consulta da linguagem Cypher e são organizadas de acordo com suas respectivas funcionalidades. As principais funções de classe incluem:

- “Create”: Responsável por criar elementos no banco de dados, como nós e relacionamentos. Essa função permite especificar as propriedades dos elementos criados;
- “Match”: Busca elementos no banco de dados com base nos critérios específicos. É uma parte fundamental das consultas, ao definir os elementos a serem manipulados;
- “Order”: Ordena os resultados da consulta segundo os critérios específicos, como ordem alfabética ou numérica;
- “Return”: Especifica quais elementos ou propriedades devem ser retornados como resultado da consulta; e
- “Where”: Define condições que os elementos pesquisados devem atender para ser inclusos nos resultados da consulta.

Além das funções relacionadas às sentenças de consulta, a classe também implementa funções recorrentes, como:

- “Detach from”: Desvincula um nó ou relacionamento de outros elementos no banco de dados;
- “Relate to”: Define relacionamentos entre nós ou elementos do banco de dados; e
- “Update”: Permite a atualização de propriedades de nós e relacionamentos no banco de dados.

No entanto, a classe “QueryBuilder” não é destinada a ser usada diretamente pelo usuário, a menos que o usuário deseja acessá-la por meio de uma função disponível na classe principal do OGM. Para que os modelos de nós e relacionamentos tenham uma interface ainda mais amigável, o sistema implementa a classe “Queryabble”, que tem acesso a uma instância

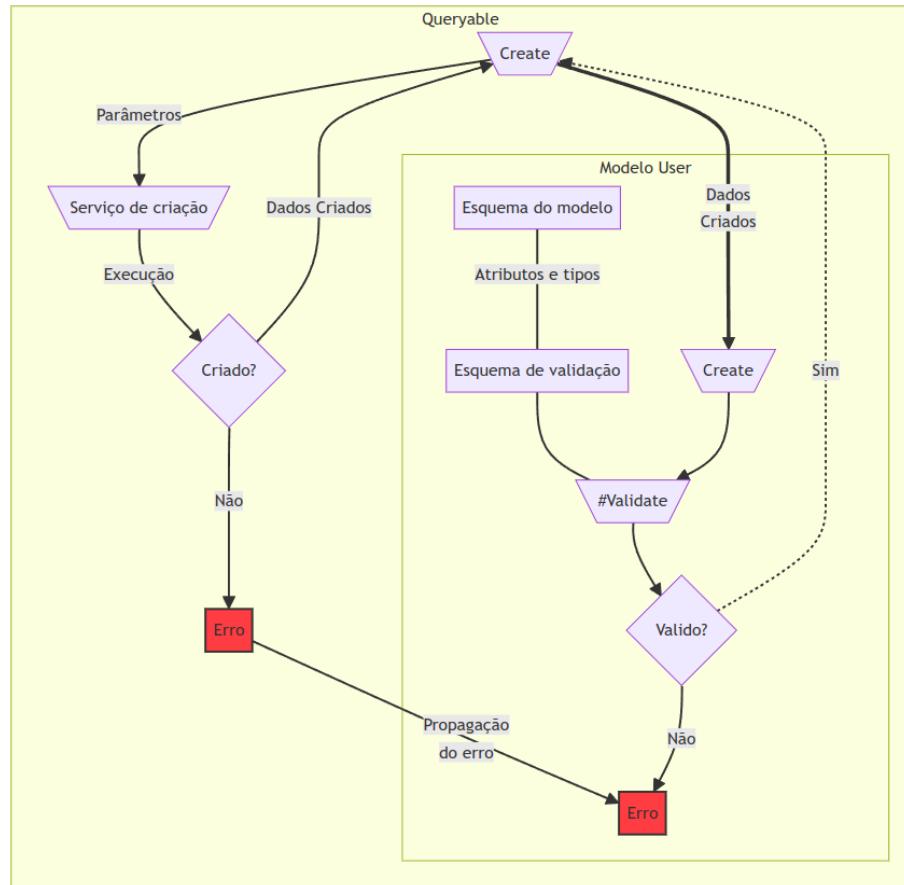
do “QueryBuilder” a qual é implementado métodos que permitem que os desenvolvedores realizem as operações:

- “Create”;
- “Delete”;
- “DeleteAll”;
- “All”;
- “Find”;
- “FindByID”; e
- “First”.

A classe “Queryable” está presente nos módulos gerados para o cliente, o que significa que o modelo dos nós e relacionamentos, destinados a serem recuperados e utilizados pelos desenvolvedores conforme em subseção 3.2.3.2, inclui a implementação dos métodos de “Queryable”, uma vez que a classe de modelo é projetada para que ela seja uma extensão da classe “Queryable”.

A função da programação orientada a objetos foi importante para o desenvolvimento desse trabalho, pois além de diminuir a complexidade de desenvolvimento, atribui conceitos de programação dinâmica usando a pilha de chamadas. Ao usar um modelo gerado pela aplicação e solicitar a criação de uma instância no banco de dados, a geração de consultas é executada conforme ilustrado na Figura 41.

Figura 41 – Fluxograma de execução simplificado sobre a execução do método “Create”



Em que se destaca que o modelo “User” contendo uma função “Create” a qual é chamada inicialmente, em que é executado a etapa de validação conforme explicado na subseção 3.2.5, que por meio da instância do “esquema de validação” é possível validar se os dados são íntegros perante o esquema, se válido, vai prosseguir com o processo executado na classe “Queryable” em que sentenças e cláusulas são adicionadas conforme os parâmetros, utilizando o serviço de criação presente na Figura 40. Ademais, qualquer erro durante a execução é propagado para o desenvolvedor realizar os tratamentos.

3.2.5 Módulo de validação

O módulo de validação tem um papel importante em garantir a integridade e conformidade dos dados manipulados pelo sistema, sendo projetado para realizar a validação de dados de acordo com as especificações definidas para o modelo durante a execução de alguma instrução.

Desta forma, a validação de dados é realizada por meio de uma função privada na classe de modelo chamada “validate”, tendo sido, incorporada a esta classe para que pudesse ser executada ao realizar uma instrução controlada pelo construtor de consultas. A função captura o esquema associado ao modelo e aplica a validação com base nessas especificações,

gerando, se não já definido, um esquema de validação, isto é, uma instância de um validador, que por padrão é da biblioteca Zod.

Além disso, o módulo de validação oferece métodos sobrecarregados que possibilitam aos desenvolvedores controlar o processo de validação, incluindo a capacidade de pular a etapa ou fornecer parâmetros para a validação.

3.2.6 Aplicação

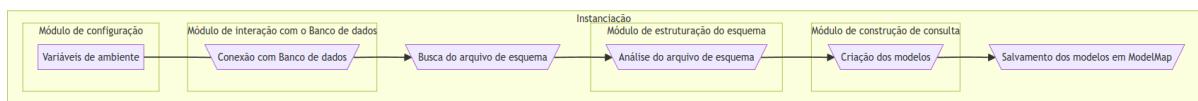
Ludicamente é o coração do sistema, onde todos os módulos independentes se unem para criar uma aplicação funcional, a classe central chamada “OGM” é responsável na coordenação das ações, e gestão de inicialização da aplicação na totalidade.

No construtor da classe OGM, uma série de ações fundamentais são executadas para garantir que a aplicação esteja pronta:

1. Criar instância do mapa de modelos, para ter a estrutura para suportar os modelos de nós e relacionamentos posteriormente criados; e
2. Localizar o arquivo de esquema da aplicação, percorrendo os arquivos e pastas recursivamente da esquerda para direita, com exceções de pastas como diretório de instalação de pacotes comuns no ambiente Javascript.

A interação entre os módulos na aplicação é coordenada conforme o fluxograma exposto na Figura 42. Esse fluxograma estabelece a ordem das operações e a comunicação entre os diferentes componentes.

Figura 42 – Fluxograma da instanciação da aplicação



Ademais, a classe OGM oferece a capacidade de inicialização da aplicação por meio de variáveis direitas invés de variáveis de ambiente. Ao optar por esta abordagem, o módulo de configuração é inicializado antes que os processos enumerados pelo construtor.

3.3 Desenvolvimento da API

Nesta seção, é explorada a metodologia adotada para o desenvolvimento da API que incorpora o OGM desenvolvido. A criação de uma API é uma escolha estratégica, uma vez

que as APIs são frequentemente associadas a ORM e OGM para servir como a interface de comunicação com sistemas e aplicações.

A criação desta API não apenas demonstra a aplicação prática do OGM, mas também permite uma análise realista do desempenho e da usabilidade do sistema em um contexto do mundo real. Para garantir que a API seja desenvolvida de maneira mais eficaz e apropriada possível, foi realizado uma pesquisa de mercado, identificando as tecnologias e ferramentas, mais aplicadas na criação de APIs, garantindo que o projeto simule e esteja nos padrões da indústria.

Para o desenvolvimento da API, foi essencial o planejamento inicial para integrar o OGM da maneira mais eficaz possível. Esse planejamento envolveu a ideia de um “concern”, que usando a definição de Rails (2023) pode ser definido como um módulo incluído em uma classe para adicionar uma funcionalidade. Sendo criado *concerns* para implementar a estrutura básica de um controlador, serviço e serializador, que recebe o serviço do OGM como entrada. Com base nos parâmetros fornecidos, são definidos métodos e rotas padrões, como, “all”, “update” (que inclui “PUT” e “PATCH”) e “create”. Essa abordagem permitiu uma integração eficiente e consistente, garantindo que as operações CRUD (Create, Read, Update, Delete) fossem tratadas padronizadamente.

Desta forma foi possível criar comportamentos abstratos aplicáveis em todos os modelos e rotas, conforme descrito nas próximas subseções.

3.3.1 Tratamentos

O *concern* para o controlador realiza alguns processos para garantir a integridade dos dados e seguir as especificações da JSON-API (API, 2022).

3.3.1.1 Tratamento de erros

O tratamento de erros é fundamental para capturar e gerenciar exceções e problemas inesperados que podem surgir durante a execução da requisição. Isso garante que, em vez de falhas catastróficas que poderia interromper a operação e retornar um erro sem semântica, este processo, garante que a aplicação possa responder adequadamente os erros, fornecendo mensagens de erros úteis, tomando medidas corretivas, aumentando a confiabilidade e segurança da aplicação.

Figura 43 – Método que realiza o tratamento de erros comuns.

```
#handleError(error: OGSError): ServiceException {
    if (error instanceof EntityError) {
        return {
            code: HttpStatus.FAILED_DEPENDENCY,
            cause: { ...error, message: error.message },
        };
    } else if (error instanceof TransactionError) {
        return {
            code: HttpStatus.INTERNAL_SERVER_ERROR,
            cause: { ...error, message: error.message },
        };
    } else if (error instanceof ModelError) {
        return {
            code: HttpStatus.BAD_REQUEST,
            cause: { ...error, message: error.message },
        };
    } else if (error instanceof ValidationError) {
        return {
            code: HttpStatus.BAD_REQUEST,
            cause: { ...error, message: error.message },
        };
    } else if (error instanceof SchemaError) {
        return {
            code: HttpStatus.BAD_REQUEST,
            cause: { ...error, message: error.message },
        };
    }

    return { code: HttpStatus.INTERNAL_SERVER_ERROR, cause: error };
}
```

Fonte: Elaborado pelo autor.

3.3.1.2 Tratamento de paginação

A paginação é uma técnica para lidar com conjuntos grandes de dados, como resultados de listagens, dividindo-os em partes menores, aprimorando o carregamento dos dados, evitando sobrecarga dos serviços e reduz o tempo de resposta.

3.3.1.3 Tratamento de parâmetros

O tratamento de parâmetros diz respeito à capacidade de filtrar e selecionar dados com base nos parâmetros fornecidos na requisição. Isso oferece uma maior flexibilidade de personalizar os resultados, em busca de acessar informações específicas.

Figura 44 – Método que realiza o tratamento dos parâmetros de campos obtidos pela requisição.

```
#handleAttributes(query: Record<string, any>): void {
  const { include } = query;

  // get fields[<model>] from query and set { projection: { <model>: 0 } }
  // TODO: implement <model>: 1 for projection if has included but not fields
  this.serializationOptions.projection = Object.entries(query)
    .filter(([key]) => key.startsWith('fields['))
    .reduce((projection, [key, fields]) => {
      const model = key.replace('fields[', '').replace(']', '');

      this.serializationFields = fields.split(',') as (keyof Schema &
        string)[];
      return { ...projection, [model]: 0 };
    }, {});
}

// each '.' in include is a depth level
if (include) {
  this.serializationOptions.depth = include.split('.').length;
}
```

Fonte: Elaborado pelo autor.

3.3.2 Rotas padrões

Cada modelo de dados definido na aplicação é normalmente acompanhado de um conjunto de rotas padrões que permite realizar operações CRUD, nos dados correspondentes. Essas rotas são projetadas para serem semanticamente claras e seguem convenções aceitas em APIs REST (Representational State Transfer), em que, cada método HTTP é associado a essas rotas que chamam o serviço do modelo de dados correspondente para realizar a conexão com o banco de dados.

- Listagem;
- Buscar por identificador;
- Criar novo dado; e
- Remover um dado.

Cada uma dessas rotas chama o serviço correspondendo, para interagir com o banco de dados utilizando um tratamento de erros para detectar se houve um erro durante a execução do serviço e assim retornando uma situação (*status*) coerente.

3.3.3 Criação do serviço do mapeador

Para alcançar o acesso granular e flexível aos recursos do mapeadores, é necessário criar um módulo dinâmico do mapeador para que esse serviço fosse disponível em toda aplicação. Desta forma é possível que o mapeador se inicialize como um serviço do projeto.

3.3.3.1 Módulo do OGM

Para o OGM foi necessário criar uma classe e definir como um módulo dinâmico, utilizando o decorador “@Module” e usando os métodos “forRoot” e “forRootAsync” para definir as dependências do módulo que incluem: o próprio OGM e o arquivo de configuração do módulo.

Figura 45 – Definição do módulo dinâmico para o OGM desenvolvido.

```

import { DynamicModule, Module } from '@nestjs/common';

import { OGMService } from './ogm.service';
import { NEO4J_CONFIG, NEO4J_DRIVER } from './ogm.constants';
import { Neo4jConfig } from './ogm.interface';

import { OGM } from 'ogm-neo4j/app';
import { ConfigModule } from '@nestjs/config';

@Module({})
export class OGMModule {
    static forRoot(config: Neo4jConfig): DynamicModule {
        return {
            module: OGMModule,
            global: true,
            providers: [
                {
                    provide: NEO4J_CONFIG,
                    useValue: config,
                },
                {
                    provide: NEO4J_DRIVER,
                    inject: [NEO4J_CONFIG],
                    useFactory: async ({
                        connectionString,
                        password,
                        username,
                        config,
                    }: Neo4jConfig) =>
                        await OGM.build(connectionString, username, password, config),
                },
                OGMService,
            ],
            exports: [OGMService],
        };
    }

    static forRootAsync(configProvider): DynamicModule {
        return {
            module: OGMModule,
            global: true,
            imports: [ConfigModule],
            providers: [
                {
                    provide: NEO4J_CONFIG,
                    ...configProvider,
                },
                {
                    provide: NEO4J_DRIVER,
                    inject: [NEO4J_CONFIG],
                    useFactory: async ({
                        connectionString,
                        password,
                        username,
                        database: _database,
                        config,
                    }: Neo4jConfig) =>
                        await OGM.build(connectionString, username, password, config),
                },
                OGMService,
            ],
            exports: [OGMService],
        };
    }
}

```

Fonte: Elaborado pelo autor.

3.3.3.2 Módulo do ORM

Para o PrismaJS como o ORM, foi necessário criar um módulo atrelado com um serviço, uma vez que cada módulo do cliente é instância independente, contudo é necessário criar um serviço para estender o cliente do prisma para compreender os estágios de vida da aplicação, como inicialização e finalização segura da aplicação, conforme apresentado nas Figura 46 e Figura 47.

Figura 46 – Definição do módulo para o PrismaJS.

```
import { Module } from '@nestjs/common';
import { DatabaseService } from './database.service';

@Module({
  providers: [DatabaseService],
  exports: [DatabaseService],
})
export class DatabaseModule {}
```

Fonte: Elaborada pelo autor.

Figura 47 – Definição da extensão do “PrismaClient”

```
import { Injectable, OnModuleDestroy, OnModuleInit } from '@nestjs/common';
import { ConfigService } from '@nestjs/config';
import { PrismaClient } from '@prisma/client';

@Injectable()
export class DatabaseService
  extends PrismaClient
  implements OnModuleInit, OnModuleDestroy
{
  constructor(private configService: ConfigService) {
    /**
     * Get the database url from environmental variables and pass it in.
     */
    super({
      datasources: {
        db: {
          url: configService.get('DATABASE.CONNECTION_STRING'),
        },
      },
    });
  }

  async onModuleInit() {
    await this.$connect();
  }

  async onModuleDestroy() {
    await this.$disconnect();
  }
}
```

Fonte: Elaborada pelo autor.

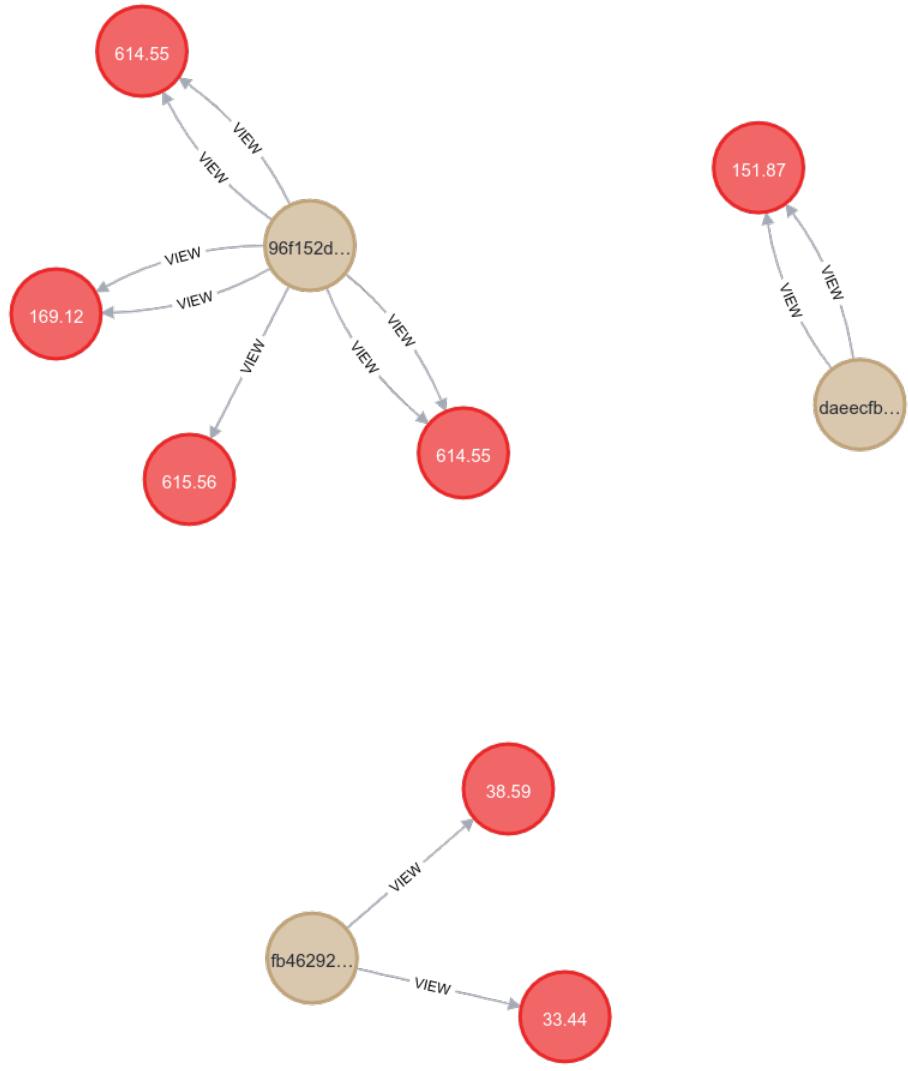
É possível notar que na Figura 47 é definido tratativas para os eventos que acontecem no módulo, sendo eles: inicialização e finalização. Desta forma, ao inicializar o módulo o serviço irá se conectar com o banco de dados e ao finalizar ou ao abortar a conexão será fechada.

3.3.4 Criação dos modelos

A criação dos modelos de dados para este projeto foi uma tarefa relativamente simples, graças à natureza do conjunto de dados mencionado na subseção 3.1.3.1. Esse conjunto de dados está em um formato lógico de um modelo de dados relacional, o que significava que não era necessário realizar um processo extenso de normalização para adaptá-lo a um banco de dados relacional em sua terceira forma normal.

No entanto, a criação dos modelos de dados de grafo exigiu algumas adaptações. O principal ajuste feito foi a transformação da relação entre produtos e sessões de usuário, sendo um evento materializado na forma de uma relação entre dois nós, assim explicado na subseção 3.4.1.3.1. Portanto, o evento que originalmente é associado apenas a produtos, passou a ser uma relação entre os nós de produtos e usuários. A Figura 49 representa o esquema do banco de dados baseado em grafos.

Figura 48 – Estrutura da relação eventos entre produtos e a sessões de usuário.



Fonte: Elaborado pelo autor.

Com base na Figura 48 fica claro que a representação de dados fica mais explicativa, se comparada com a visualização tabular em modelos relacionais. Na qual, na imagem é possível visualizar que em cada sessão, representado pelo nó na cor marrom executou a ação de visualizar nos produtos, representados em vermelho.

Figura 49 – Estrutura do esquema para o OGM desenvolvido.

```

Node User {
    id:      Int          @identifier(auto: true)
}

Node Product {
    id:      Int          @identifier(auto: true)
    brand:  String?
    price:  Decimal
}

Node Category {
    id:      String       @identifier
    code:   String?
}

Relationship Event {
    time:     Date
    type:     Enum { "view", "cart", "remove_from_cart", "purchase" }
    user_session: String
    user:     User
    product:  Product
}

```

Fonte: Elaborada pelo autor.

Seguindo o processo descrito na subseção 3.2.3 e com a base de dados selecionada narrada na subseção 3.1.3.1 foi criado o esquema apresentado na Figura 49, que em destaque fica o tipo do atributo identificador do nó de categoria, que devido à representação desse dado no conjunto de dados, foi necessário adaptá-lo para uma cadeia de caracteres.

Essas adaptações foram necessárias para criar um modelo de dados de grafos coerentes que refletissem a interação complexa entre produtos e usuários nos dados do conjunto. Com essas mudanças, a estrutura do banco de dados de grafo tornou-se mais representativa conforme a Figura 48.

Enquanto para o esquema de dados usado para definir a estrutura do banco de dados, usando o Prisma (3.1.2.3):

Figura 50 – Estrutura do esquema para a API desenvolvida usando Prisma e banco de dados relacional.

```

generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_POSTGRES_URL")
}

model User {
  id      Int      @id @default(autoincrement())
  events Event[]
}

model Product {
  id        Int      @id @default(autoincrement())
  brand     String?
  price     Decimal
  events    Event[]
  category_id String
  category   Category @relation(fields: [category_id], references: [id])
}

model Category {
  id      String    @id
  code    String?
  Product Product[]
}

enum EventKind {
  cart
  view
  purchase
  remove_from_cart
}

model Event {
  id      Int      @id @default(autoincrement())
  time   DateTime @db.Timestamptz()
  kind    EventKind
  user_session String
  user_id   Int
  product_id Int
  user     User     @relation(fields: [user_id], references: [id])
  product  Product  @relation(fields: [product_id], references: [id])
}

```

Fonte: Elaborada pelo autor.

Em que é possível notar a similaridade entre as duas figuras 50 e 49, contudo, é notável o tratamento usando o operador “@relation” para definir relações e os atributos que compõem essa relação, o que condiz com o modelo relacional descrito na subseção 2.1.2.1. No entanto, para definir o atributo “time” em “Event”, o qual no conjunto de dados é usado a representação numérica de uma data, foi necessário ter um conhecimento complexo da linguagem e do banco de dados utilizado, por ser necessário a utilização do operador “@db.Timestamptz”.

Após a definição dos modelos de dados, foi necessário criar componentes adicionais

para cada modelo de dados, a fim de estabelecer a estrutura e funcionalidades do NestJS. Para ambos os modelos, relacional e de grafo, a criação de controladores, interfaces, serializadores e serviços.

No caso do modelo de dados relacional, além dos controladores, interface e serviços, a necessidade de um DTO (*Data Transfer Object*) se faz presente. Esse DTO, que representa um objeto de transferência de dados, desempenha um papel importante na validação dos dados e na definição da estrutura dos objetos que serão transferidos entre as camadas da aplicação. Além disso, a criação de um arquivo relacionado a entidade para mapear o modelo de dados para a documentação do Swagger⁷. Juntos, o controlador, o DTO, a entidade e o serviço garantem a manipulação segura e eficaz dos dados na aplicação.

Para o modelo de dado baseado em grafo, com o uso dos *concerns* foi possível definir os controladores, serviços e serializadores de maneira simples conforme exibe as imagens a seguir:

Figura 51 – Definição do serviço de categorias usando o *concern*.

```
import { Injectable } from '@nestjs/common';

import { BaseModelService } from 'core/concerns/model';
import { OGMService } from 'core/database/ogm-neo4j/ogm.service';

import type { CategoryModel } from './category.interface';

@Injectable()
export class CategoriesService extends BaseModelService<CategoryModel> {
    constructor(private readonly ogmService: OGMService) {
        super(ogmService, 'Category');
    }
}
```

Fonte: Elaborada pelo autor.

Figura 52 – Definição do serializador de categorias usando o *concern*.

```
import { BaseSerializer } from 'core/concerns/model';

import { Category } from './category.interface';
import { ConfigService } from '@nestjs/config';

export class CategorySerializer extends BaseSerializer<Category> {
    constructor(private readonly configService: ConfigService) {
        super('categories', configService);
    }
}
```

Fonte: Elaborada pelo autor.

⁷ Trata-se de uma aplicação de código aberto que auxilia desenvolvedores nos processos de definir, documentar e consumir APIs REST, conforme Gr1d (2022).

Figura 53 – Definição do controlador de categorias usando o *concern*.

```

import { Controller } from '@nestjs/common';
import { ConfigService } from '@nestjs/config';

import { BaseController } from 'core/concerns/model/base.controller';

import { Category } from './category.interface';
import { CategoriesService } from './category.service';
import { CategorySerializer } from './category.serializer';

@Controller('categories')
export class CategoriesController extends BaseController<Category, 'id'> {
    constructor(
        private readonly categoryService: CategoriesService,
        private readonly configService: ConfigService,
    ) {
        super(categoryService, new CategorySerializer(configService));
    }
}

```

Fonte: Elaborada pelo autor.

Para garantir a consistência entre as duas aplicações e seguir um padrão bem definido, a criação de *concerns*, para o modelo de dados relacional também foi empreendida. No entanto, essa tarefa revelou-se um pouco mais desafiadora em comparação com o modelo de dados de grafo, como evidenciado nas imagens da Figura 54 e Figura 55. Um dos principais motivos para essa complexidade reside na natureza dinâmica do Prisma, o qual não permite os nomes das entidades de forma estática nos tipos. Isso implica que não é possível inferir os tipos de maneira direta nos *concerns*, o que pode resultar na perda da vantagem intrínseca ao uso do Typescript, conforme discutido em subseção 2.5.1.3. No entanto, apesar dessas complexidades, as implementações são comparáveis e intercambiáveis.

Figura 54 – Definição do *concern* de serviço na aplicação usando o OGM desenvolvido

```

import { BadGatewayException, HttpStatus } from '@nestjs/common';
import { Model } from 'ogm-neo4j/models';

import { OGMService } from 'core/database/ogm-neo4j/ogm.service';
import {
    AllResponse,
    CreateResponse,
    DeleteResponse,
    GetResponse,
    IBaseModelService,
} from './ibase.service';
import { jsonData } from './base.controller';

export interface ServiceException {
    code: HttpStatus;
    cause: jsonData;
}

export class BaseModelService<T extends Model<any, any>>
    implements IBaseModelService<T>
{
    readonly name: string;
    readonly #entityRepository: T;

    constructor(service: OGMService, name: string) {
        this.name = name;
        this.#entityRepository = service.app.retrieveModel<T>(this.name);
    }

    create(entity: Parameters<T['create']>['0']) {
        try {
            return this.#entityRepository.create(entity) as CreateResponse<T>;
        } catch (error) {
            throw new BadGatewayException(error);
        }
    }

    getAll(
        properties?: Parameters<T['all']>['0'],
        options?: Parameters<T['all']>['1'],
    ) {
        try {
            return this.#entityRepository.all(properties, options) as AllResponse<T>;
        } catch (error) {
            throw new BadGatewayException(error);
        }
    }

    get(id: Parameters<T['find']>['0']) {
        try {
            return this.#entityRepository.find(id) as GetResponse<T>;
        } catch (error) {
            throw new BadGatewayException(error);
        }
    }

    delete(id: Parameters<T['delete']>['0']) {
        try {
            return this.#entityRepository.delete(id) as DeleteResponse<T>;
        } catch (error) {
            throw new BadGatewayException(error);
        }
    }
}

```

Fonte: Elaborada pelo autor.

É possível notar na Figura 54 que ao receber como parâmetro de interface a interface do modelo, toda a classe “BaseModelService” irá garantir que a tipagem estática seja propagada,

isto é, ao inserir a interface do modelo os parâmetros e retornos das funções do serviço estarão com a inferência dos tipos alinhados e corretos, facilitando o desenvolvimento, como descrito na subseção 2.5.1.3.

Figura 55 – Definição do *concern* de serviço na aplicação usando o PrismaJS

```

import { HttpStatus } from '@nestjs/common';
import { DatabaseService } from '@core/database/database.service';

import { IBaseModelService } from './IBaseModel.service';
import { JsonData } from './baseModel.controller';
import { Prisma, PrismaClient } from '@prisma/client';
import { ZodSchema } from 'zod';

export interface ServiceException {
  code: HttpStatus;
  cause: JsonData;
}

export class BaseModelService<
  T extends Lowercase<Prisma.ModelName>,
  EntitySchema extends Record<string, any>,
  I extends keyof EntitySchema,
> implements IBaseModelService<EntitySchema, I>
{
  readonly name: keyof DatabaseService;
  #entityRepository: PrismaClient[T];
  #schema: ZodSchema<EntitySchema>;

  constructor(
    service: DatabaseService,
    name: keyof DatabaseService,
    schema: ZodSchema<EntitySchema>,
  ) {
    this.name = name;
    this.#entityRepository = service[name] as PrismaClient[T];
    this.#schema = schema;
  }

  getAll(): Promise<EntitySchema[]> {
    // @ts-expect-error - There is no way to infer the type of repository
    return this.#entityRepository.findMany();
  }

  get(id: EntitySchema[I]): Promise<EntitySchema> {
    // @ts-expect-error - There is no way to infer the type of repository
    return this.#entityRepository.findUnique({ where: { id } });
  }

  create(data: Partial<EntitySchema>): Promise<EntitySchema> {
    this.#schema.parse(data);
    // @ts-expect-error - There is no way to infer the type of repository
    return this.#entityRepository.create({ data });
  }

  delete(id: EntitySchema[I]): Promise<EntitySchema> {
    // @ts-expect-error - There is no way to infer the type of repository
    return this.#entityRepository.delete({ where: { id } });
  }

  update(
    id: EntitySchema[I],
    data: Partial<EntitySchema>,
  ): Promise<EntitySchema> {
    // @ts-expect-error - There is no way to infer the type of repository
    return this.#entityRepository.update({ where: { id }, data });
  }
}

```

Fonte: Elaborada pelo autor.

No entanto, na Figura 55 foi necessário utilizar o operador “@ts-expect-error” para o interpretador não verificar a tipagem e nem propagar, desta forma é necessário que o desenvolvedor tenha que garantir que os retornos das funções estejam alinhados, o que no ciclo de desenvolvimento de software conforme Silva et al. (2001) é uma preocupação adicional.

3.3.5 Otimizações

Após a execução bem-sucedida dos testes da aplicação, uma etapa crítica do processo de desenvolvimento foi a realização de estudos destinados a identificar oportunidades para otimizar os tempos de resposta da aplicação. Uma das possibilidades destacadas, inclusiva pela própria documentação do Nest.js foi a substituição do provedor HTTP padrão por uma alternativa, o Fastify.

Fastify e Express são dois *frameworks* de desenvolvimento de aplicações web, ambos fornecendo provedores HTTP para atender às requisições HTTP. No entanto, Fastify é uma opção mais recente, e tem ganhado destaque por sua reputação de ser mais veloz, conforme uma análise conduzida por Lawson (2022), demonstrando um desempenho superior em comparação com o Express. Essa melhoria de desempenho pode ser crucial, especialmente em aplicações onde a latência e a escalabilidade são críticas, como no contexto deste projeto.

Para realizar a alteração foi simples, instalando o novo provedor e alterando o arquivo de inicialização do projeto atribuindo o novo provedor, assim como ilustra na Figura 56. Contudo, a interface dos comandos para criação de uma resposta foi alterada, então foi necessário alterar o *concern* do controlador, conforme as alterações feitas na Figura 57.

Figura 56 – Atribuição do adaptador do Fastify a aplicação.

```
import { NestFactory } from '@nestjs/core';
import {
  FastifyAdapter,
  NestFastifyApplication,
} from '@nestjs/platform-fastify';
import { AppModule } from './app.module';
import { Logger, INestApplication } from '@nestjs/common';
import { ConfigService } from '@nestjs/config';

type Application = INestApplication & NestFastifyApplication;

async function bootstrap() {
  const app = await NestFactory.create<Application>(
    AppModule,
    new FastifyAdapter(),
  );
  const configService = app.get(ConfigService);

  await app.listen(configService.get('APP_PORT') ?? 3000, '0.0.0.0');

  Logger.log(`Server running on ${await app.getUrl()}`);
}

bootstrap();
```

Fonte: Elaborada pelo autor.

Figura 57 – Trecho das alterações realizadas no *concern* do controlador ao substituir o Express para o Fastify

```

export class BaseModelController<
  T extends Record<string, any>,
  I extends keyof T,
> {
  constructor(private readonly service: IBaseModelService<T, I>) {}

  #handleError(error: any): ServiceException {
    return { code: HttpStatus.INTERNAL_SERVER_ERROR, cause: error };
  }

  @Get()
  async findAll(@Res() response: Response): Promise<JsonData> {
    return this.service
      .getAll()
      .then((entities) =>
        response
          + .header('Content-Type', 'application/json')
          .status(HttpStatus.OK)
          - .json(JSON.stringify(Object.fromEntries(entities))),
          + .send(entities),
      )
      .catch((exception) => {
        const { code, cause: why } = this.#handleError(exception);

        return response.status(code).json({ why });
      });
  }
}

```

Fonte: Elaborada pelo autor.

3.4 Hospedagem

A hospedagem de uma aplicação é uma etapa crucial no ciclo de desenvolvimento, que envolve disponibilizar a aplicação para acesso dos usuários e garantir a funcionalidade e segurança. Nesta seção, será discutido o processo de hospedagem da aplicação e do OGM desenvolvido.

3.4.1 “Dockerização” da aplicação

A hospedagem da aplicação começou com a criação de um Dockerfile, um arquivo de configuração que descreve como a aplicação deve ser empacotada em um contêiner Docker. O Dockerfile foi projetado com a incorporação de conceitos avançados, como estágios separados, a fim de aprimorar a segurança e a eficiência do processo de construção do contêiner.

3.4.1.1 Criação da imagem

1. Estágio de instalação das dependências do sistema: Neste estágio as dependências do sistema necessárias para aplicação foram instaladas. Isso incluiu a instalação do Git para instalar o pacote do OGM desenvolvido do servidor do Github;

2. Estágio de instalação das dependências da aplicação: No segundo estágio, as dependências da aplicação foram instaladas, envolvendo o gerenciador de pacotes do ambiente de execução, que foi escolhido o NPM, para recuperar todas as bibliotecas e módulos necessários para a execução da aplicação; e
3. Estágio de execução da aplicação: O último estágio envolve a execução da inicialização da aplicação, sendo usado por boas práticas a definição no arquivo “package.json”.

3.4.1.2 Gerenciamento da aplicação

Para facilitar o gerenciamento dos contêineres e separando em ambientes de desenvolvimento e produção, foram criados dois arquivos “Docker Compose”, um para o ambiente de produção e outro para a execução local.

- “docker-compose.yml” (produção): Descreve a configuração necessária para implantar a aplicação no ambiente de produção, capturando o arquivo de variáveis de ambiente do Portainer (“stack.env”), incluindo configuração das imagens do banco de dados e os volumes, que são diretórios especiais que não são voláteis; e
- “docker-compose.local.yml”: Enquanto neste arquivo o foco é a execução simultânea da aplicação e o do banco de dados, usando um arquivo local como entrada das variáveis de ambiente. No entanto, também existem os volumes para os banco de dados e a execução da aplicação é usando o modo de desenvolvimento, para permitir *hot-reload* enquanto edita os arquivos da aplicação.

Para o ambiente de produção, devido ao grande volume de dados presente no conjunto de dados (subseção 3.1.3.1), a otimização dos recursos e minimizações dos custos relacionados à movimentação e o uso de rede tornaram-se imperativos. Para atingir esse objetivo, optou-se por criar um volume do tipo “bind” (ligação) no ambiente Docker. Esse volume foi vinculado a uma pasta do usuário que contém o extenso conjunto de dados, permitindo que a aplicação acesse esses dados. No entanto, essa solução não veio sem desafios, problemas de permissões e grupos de usuários surgiram, o que afetou a capacidade de acesso ao volume, identificando esse problema foi possível resolver criando um chamado para o operador do sistema, para executar a restauração adequada das permissões.

Além disso, a infraestrutura necessária para o funcionamento da aplicação envolveu a solicitação de liberação de quatro portas para permitir o acesso dos usuários. Cada uma dessas portas foi designada para um propósito, sendo eles: banco de dados de grafo, banco de dados relacional, a aplicação que utiliza o ORM e aplicação que desenvolvida com o OGM. Após a liberação das quatro portas, a próxima etapa foi configurar o proxy reverso para direcionar o tráfego de entrada e saída.

Figura 58 – Contêineres executando no ambiente de produção.

Name	State	Image	Created	IP Address	Gpus	Published Ports	Ownership
kszinhu.ecommerce-analysis.ap...	running	ghcr.io/kszinhu/neo4j-bin:dbms-neo4j	2023-10-18 21:36:26	...	none	4242:4242	private
kszinhu.ecommerce-analysis.ap...	running	ghcr.io/kszinhu/neo4j-bin:dbms-postgres	2023-10-18 21:36:19	...	none	5682:5682	private
kszinhu.ecommerce-analysis.d...	running	e-commerce-analysis	neo4j:5-enterprise	2023-10-18 21:36:18	none	9090:7687	private
kszinhu.ecommerce-analysis.d...	running	e-commerce-analysis	postgres:16-alpine	2023-10-18 21:36:18	none	1995:5432	private

Fonte: Elaborada pelo autor.

Ademais, foi necessário criar adicionar elementos no servidor de DNS próprio para garantir que os domínios fossem associados às portas e endereços apropriados.

3.4.1.2.1 Otimização

Foi realizada uma otimização essencial no contêiner do banco de dados Neo4j, que envolveu a modificação de dois parâmetros críticos: a quantidade da “thread pool” que corresponde a quantidade de *threads*, e o tamanho da página de armazenamento. Essas mudanças foram fundamentais para otimizar o desempenho do banco de dados Neo4j, permitindo uma alocação mais eficiente de recursos e melhorando a capacidade de processamento de consultas e transações. Através dessas otimizações, a aplicação pôde alcançar um nível superior de eficiência e responsividade, proporcionando aos usuários uma experiência mais ágil e satisfatória.

3.4.1.3 Importação do conjunto de dados

Esta seção envolve a realização de procedimentos semelhantes para os dois banco de dados, utilizando o terminal do servidor, devido ao grande volume de dados a serem importados.

Para ambos, foram criados arquivos de consulta que executavam a inserção dos dados, que coletavam todos os arquivos disponíveis no volume de dados descrito na subseção 3.4.1.2, e uma conexão SSH (Secure Socket Shell) foi estabelecida com o servidor, onde os processos de importação foram executados com o comando “nohup”⁸.

3.4.1.3.1 Neo4j

Para o Neo4j, foi elaborado um arquivo seguindo a estrutura apresentado na Figura 59. Neste processo, foram criados nós para representar produtos, usuários, sessões e categorias, enquanto os eventos foram mapeados como relações entre produtos e sessões, com rótulos distintos para cada tipo de evento. Além disso, a utilização da extensão APOC facilitou a

⁸ É uma abreviação do inglês para “no hang up”, utilizado no sistema Unix para executar um processo desassociando da sessão do terminal, desta forma o programa continua executando independentemente.

criação de *threads* que permitiram a execução em paralelo à realização de salvamento das operações durante o processo de importação.

Figura 59 – Consulta para realizar a importação do conjunto de dados ao Neo4j usando a linguagem Cypher

```

CALL apoc.periodic.iterate(
  'CALL apoc.load.csv?url=) yield map as row RETURN row',
  MERGE (product:Product {id: toInteger(row.product_id)})
    ON CREATE SET product.brand = row.brand,
      product.price = toFloat(row.price)
  MERGE (session:Session {id: row.user_session })

  FOREACH (ignoreMe IN CASE WHEN row.event_type = "view" THEN [1] ELSE [] END |
    MERGE (session)-[:VIEW {type: row.event_type, event_time: datetime(REPLACE((REPLACE(row.event_time, " UTC", "") + "Z"), " ", "T")) }]->(product)
  )
  FOREACH (ignoreMe IN CASE WHEN row.event_type = "cart" THEN [1] ELSE [] END |
    MERGE (session)-[:CART {type: row.event_type, event_time: datetime(REPLACE((REPLACE(row.event_time, " UTC", "") + "Z"), " ", "T")) }]->(product)
  )
  FOREACH (ignoreMe IN CASE WHEN row.event_type = "purchase" THEN [1] ELSE [] END |
    MERGE (session)-[:PURCHASE {type: row.event_type, event_time: datetime(REPLACE((REPLACE(row.event_time, " UTC", "") + "Z"), " ", "T")) }]->(product)
  )
  FOREACH (ignoreMe IN CASE WHEN row.event_type = "remove_from_cart" THEN [1] ELSE [] END |
    MERGE (session)-[:REMOVE_FROM_CART {type: row.event_type, event_time: datetime(REPLACE((REPLACE(row.event_time, " UTC", "") + "Z"), " ", "T")) }]->(product)
  )

  MERGE (user:User {id: toInteger(row.user_id)})
  MERGE (user)-[:HAS]->(session)
  MERGE (category:Category {id: row.category_id})
    ON CREATE SET category.code = row.category_code

  MERGE (product)-[:BELONGS_TO]->(category)
  ', {batchSize:10000, iterateList: true, parallel:true});
  
```

Fonte: Elaborada pelo autor.

3.4.1.3.2 PostgreSQL

Enquanto para o PostgreSQL, a abordagem foi ligeiramente diferente. Primeiramente uma tabela temporária foi criada. Em seguida, todos os arquivos de dados foram percorridos, e os dados brutos foram inseridas nessa tabela temporária. Ao final do processo, os dados únicos da tabela temporária foram utilizados para popular as tabelas de usuário, produto, categoria e evento, evitando conflito. Esse método permitiu uma importação organizada dos dados no PostgreSQL.

Figura 60 – Tempos das etapas de execução de uma consulta usando Prisma

(a) Parâmetros de transação e criação da tabela temporária

```
--  
-- PostgreSQL database  
  
\connect ecommerce  
  
SET statement_timeout = 0;  
SET lock_timeout = 0;  
SET idle_in_transaction_session_timeout = 0;  
SET client_encoding = 'UTF8';  
SET standard_conforming_strings = on;  
SELECT pg_catalog.set_config('search_path', '', false);  
SET check_function_bodies = false;  
SET xmloption = content;  
SET client_min_messages = warning;  
SET row_security = off;  
  
-- "event_time", "event_type", "product_id", "category_id", "category_code", "brand", "price", "user_id", "user_session"  
  
DROP TABLE IF EXISTS public."temp_csv";  
  
CREATE TABLE public."temp_csv"  
(  
    "event_time"      timestamp,  
    "event_type"     public."EventKind",  
    "product_id"     bigint,  
    "category_id"    text, -- so big 2,1...E+18  
    "category_code"  varchar(255),  
    "brand"          varchar(255),  
    "price"          numeric(10, 2),  
    "user_id"        bigint,  
    "user_session"   varchar(255)  
);
```

(b) Importação da tabela temporária para as tabelas das entidades

```
DO  
$block$  
DECLARE  
    path text := '/var/lib/postgresql/data/import';  
    files text[] := ARRAY ['2019-Oct.csv', '2019-Nov.csv', '2019-Dec.csv', '2020-Feb.csv', '2020-Mar.csv', '2020-Apr.csv'];  
    file text;  
BEGIN  
    FOREACH file IN ARRAY files  
    LOOP  
        RAISE NOTICE 'Processing file: %', file;  
        EXECUTE $$COPY public."temp_csv"("event_time", "event_type", "product_id", "category_id",  
                                         "category_code", "brand", "price", "user_id", "user_session") FROM '$$ ||  
                                         path || '/' || file || $$' WITH (FORMAT csv, HEADER true)$$;  
    END LOOP;  
  
    RAISE NOTICE 'Done';  
  
    INSERT INTO public."User" ("id")  
    SELECT DISTINCT ON ("user_id") "user_id" AS id  
    FROM public."temp_csv";  
    INSERT INTO public."Category" ("id", "code")  
    SELECT DISTINCT ON ("category_id") "category_id" AS id, "category_code" AS code  
    FROM public."temp_csv";  
    INSERT INTO public."Product" ("id", "category_id", "brand", "price")  
    SELECT DISTINCT ON ("product_id") "product_id" AS id, "category_id", "brand", "price"  
    FROM public."temp_csv";  
    INSERT INTO public."Event" ("time", "kind", "user_session", "user_id", "product_id")  
    SELECT "event_time", "event_type", "user_session", "user_id", "product_id"  
    FROM public."temp_csv"  
    WHERE "user_session" IS NOT NULL  
        AND "user_id" IS NOT NULL  
        AND "product_id" IS NOT NULL;  
END;
```

Fonte: Elaborado pelo autor.

4 Experimentação e análise dos resultados

Neste capítulo são apresentados os experimentos realizados para a medição dos indicadores e métricas, bem como a análise dos resultados obtidos.

4.1 Experimentos

A seção de experimentos representa o momento em que as teorias, estratégias e otimizações concebidas ao longo do projeto são submetidas a testes práticas. Esses experimentos visam avaliar o desempenho, a eficácia e o impacto das mudanças e melhorias implementadas na aplicação, bem como analisar as relações e correlações entre variáveis específicas. Os experimentos oferecem indicadores sobre o funcionamento da aplicação, permitindo entender como as otimizações e implementações afetaram a desempenho, a usabilidade e a eficiência do sistema.

4.1.1 Experimentos sobre as otimizações

Um dos experimentos envolveu a realização de requisições à API da aplicação que utiliza o OGM desenvolvido. Um aspecto notável foi a diferença dos tempos de resposta, que destacou a eficácia das otimizações realizadas na subseção 3.3.5. Conforme as imagens na Figura 61, foram realizadas três requisições que visavam coletar os dados dos 300 primeiros usuários, no entanto, na Figura 62a o banco de dados não possuía índice nos usuários, obtendo o tempo de resposta de 829ms. No entanto, ao adicionar o índice ao banco de dados, usando o módulo de esquema (3.2.3), o tempo de resposta caiu significativamente para apenas 14ms, como apresentado na Figura 62b. Além disso, a substituição do provedor HTTP mostrou um impacto, uma vez o que tempo de resposta diminui para 5.18ms na Figura 62c.

Figura 61 – Requisição realizada no formato do comando Curl

```
curl --request GET \
--url 'https://api.ecommerce.neo4j.kszinhu.dev.br/users?fields[user]=id&per=300' \
--header 'User-Agent: Insomnia/2023.5.7'
```

Fonte: Elaborada pelo autor.

Na Figura 61 é possível identificar dois parâmetros utilizados: “fields[user]” e “per”. Os quais são utilizados para separar quais dados serão selecionados do modelo de usuário e a quantidade de dados paginados, respectivamente.

Figura 62 – Três requisições a aplicação desenvolvida usando o OGM.

GET `_baseUrl /users` Send 200 OK 829 ms 32 B
Body Auth Query Headers 1 Docs
Preview Headers Cookies Timeline
1 {
2 "520088904": {
3 "id": "520088904"
4 }
5 }

(a) Requisição para a aplicação no qual o banco de dados não contêm índices.

GET `_baseUrl /users` Send 200 OK 14 ms 32 B
Body Auth Query Headers 1 Docs
Preview Headers Cookies Timeline
1 {
2 "520088904": {
3 "id": "520088904"
4 }
5 }

(b) Requisição para aplicação no qual o banco de dados contêm índices e aplicação usa o provedor Express.

GET `_baseUrl /users` Send 200 OK 5.18 ms 32 B
Body Auth Query Headers 1 Docs
Preview Headers Cookies Timeline
1 {
2 "520088904": {
3 "id": "520088904"
4 }
5 }

(c) Requisição para aplicação no qual o banco de dados contêm índices e aplicação usa o provedor Fastify.

Fonte: Elaborada pelo autor.

Além disso, foi conduzido testes para avaliar o desempenho e a eficiência dos parâmetros especificados na configuração do contêiner do banco de dados Neo4j, no qual foram executadas 2000 consultas, conforme a Figura 63. Essas consultas foram projetadas para buscar 65 elementos entre os três tipos de nós presentes na aplicação: usuários, Produtos e Categorias.

Figura 63 – Consultas que buscam 65 elementos das respectivas categorias: usuário, produto e categoria.

```
MATCH (n:User)
RETURN n
LIMIT 65
```

(a) Consulta para a etiqueta de “Usuário”.

```
MATCH (n:Product)
RETURN n
LIMIT 65
```

(b) Consulta para a etiqueta de “Produto”.

```
MATCH (n:Category)
RETURN n
LIMIT 65
```

(c) Consulta para a etiqueta de “Categoria”.

Fonte: Elaborada pelo autor.

4.1.2 Experimentos sobre o mapeador

Com base nas consultas que formaram a Figura 25, foi desenvolvida uma extensa bateria de consultas com objetivo de mensurar o tempo de resposta e o tempo de criação das consultas. Esse conjunto de testes compreendeu um total de 7.500 consultas usando o mapeador executadas no ambiente de execução Node.js.

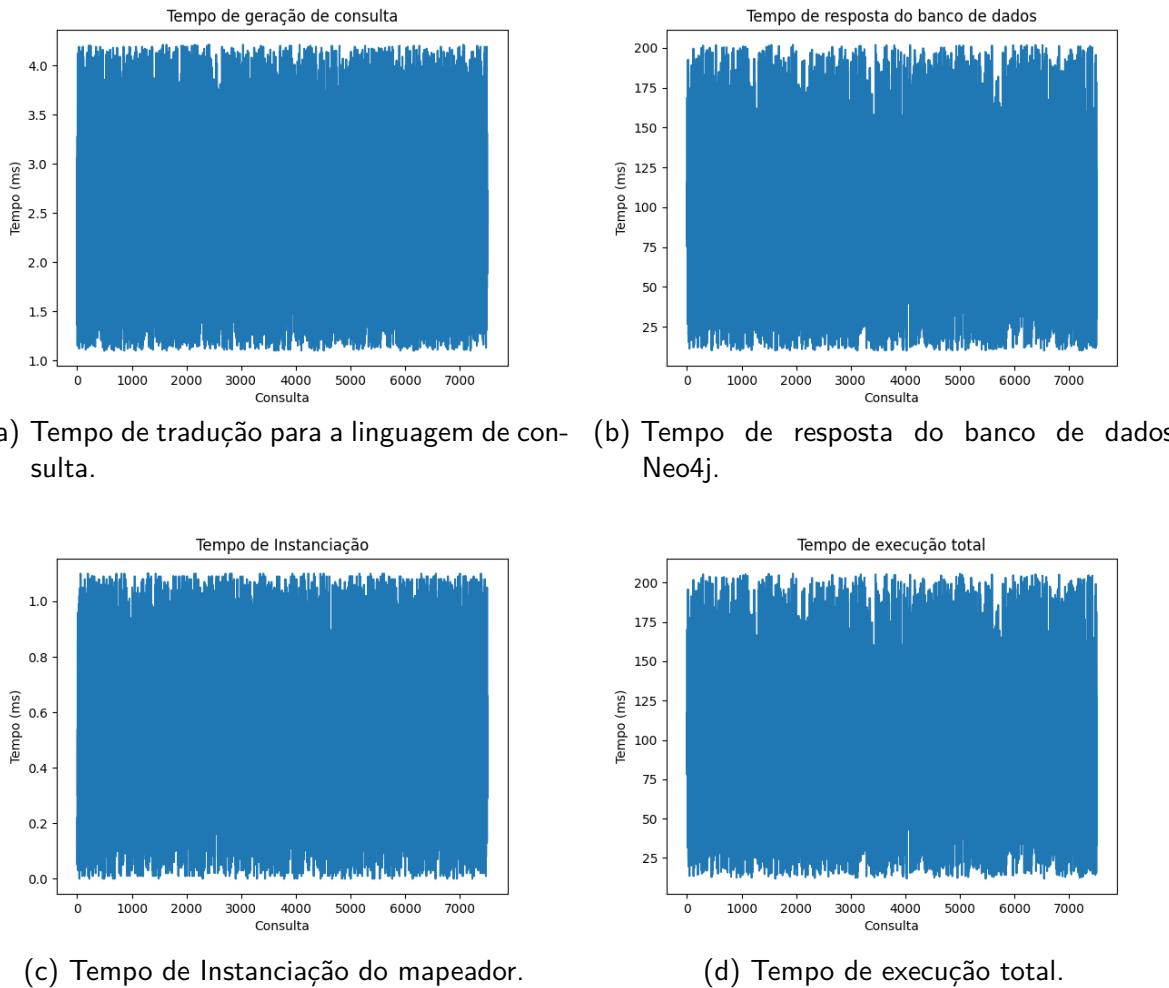
O objetivo principal desses experimentos é analisar como o OGM desenvolvido lida com as consultas e a eficiência com que é traduzido para a linguagem de consulta. Para garantir a precisão da avaliação, pontos informativos foram incorporados durante a execução dos testes, permitindo a medição precisa dos tempos de resposta e de criação das consultas.

Tabela 2 – Trecho dos tempos capturados das consultas em OGM desenvolvido em milissegundos (ms).

instantiation_time	query_generation_time	response_time
1.15	1.65	38.65
1.22	1.81	15.70

Fonte: Elaborada pelo autor.

Figura 64 – Tempos das etapas de execução de uma consulta usando OGM.



Fonte: Elaborada pelo autor.

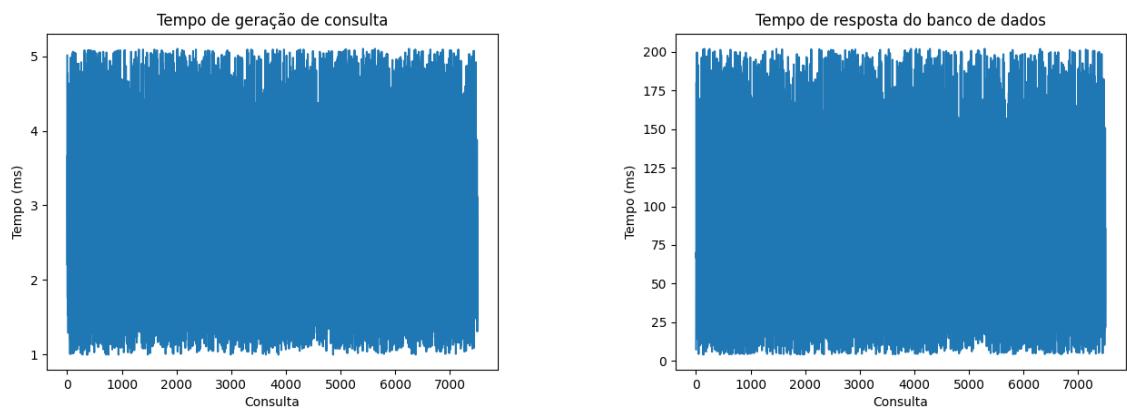
Em paralelo com os experimentos realizados para avaliar o desempenho do mapeador desenvolvido, também foi executada a mesma bateria de 7500 consultas utilizando o Prisma. No entanto, é importante destacar que, ao contrário dos testes conduzidos com o OGM, não foi possível mensurar o tempo de instanciação, isso se deve ao fato de que medir o tempo de criação das instâncias poderia afetar o código do pacote e não tem como mensurar o tempo de geração dos módulos do cliente do Prisma, uma vez que entrariam na equação. Enquanto para o OGM desenvolvido, todo o processo de instanciação, mapeamento e manipulação dos objetos estava sob análise, no caso do Prisma, será conduzida somente a captura dos tempos de criação de consulta e o tempo de resposta do banco de dados relacional.

Tabela 3 – Trecho dos tempos capturados das consultas usando o Prisma em milissegundos (ms)

query_time	response_time
183.32381309104508	2038.701460647799
501.2483866261722	1453.9984913845944

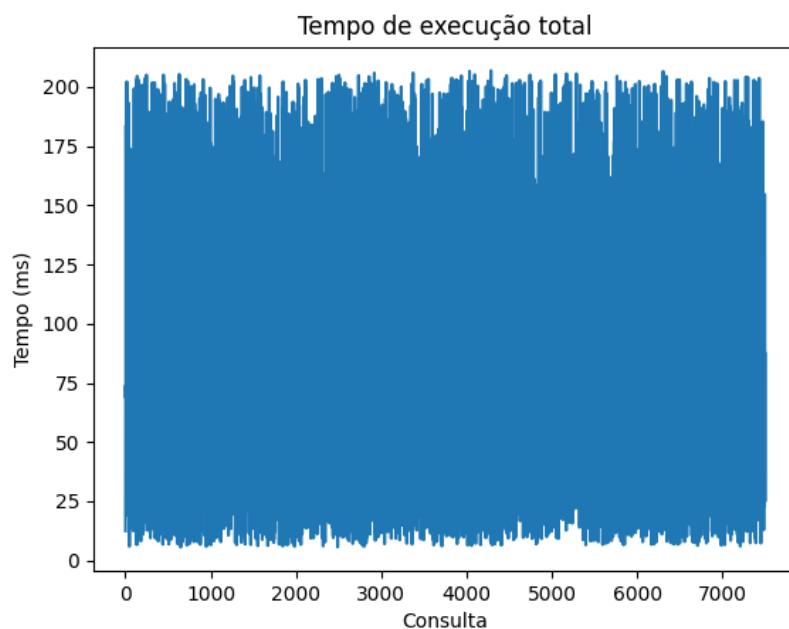
Fonte: Elaborado pelo autor.

Figura 65 – Tempos das etapas de execução de uma consulta usando Prisma



(a) Tempo de tradução para a linguagem de consulta

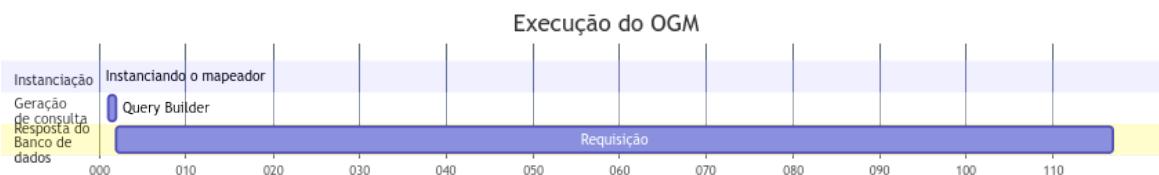
(b) Tempo de resposta do banco de dados PostgreSQL



(c) Tempo de execução total

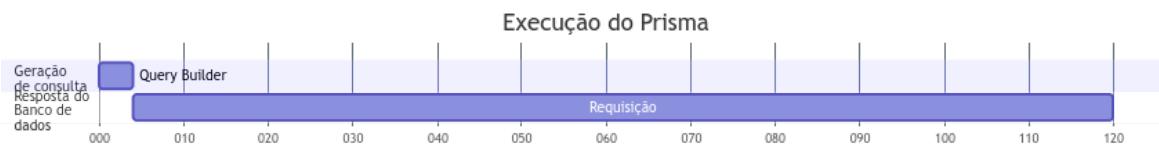
Fonte: Elaborado pelo autor.

Figura 66 – Gráfico de gantt de um dos processos capturados do OGM entre as 7500 consultas criadas



Fonte: Elaborado pelo autor.

Figura 67 – Gráfico de gantt de um dos processos capturados do ORM entre as 7500 consultas criadas



Fonte: Elaborado pelo autor.

4.1.3 Experimentos sobre a aplicação

Esta subseção representa a avaliação das duas aplicações desenvolvidas, uma das aplicações utiliza o ORM prisma, enquanto outra usa o OGM desenvolvido. O objetivo principal desses experimentos é certificar que ambas as aplicações operam como sistemas reais, com baixa latência a um nível mínimo de erros.

Um dos experimentos realizados para avaliar as duas aplicações foi a execução de requisições HTTP às APIs, utilizando a seleção de atributos, “fields” por meio de parâmetros. Essa seleção permite especificar quais atributos dos objetos eram desejados nas respostas das requisições, o que é fundamental para otimizar o consumo de recursos e a eficiência da transferência de dados. Além disso, foi limitado a exibição de 25 elementos.

Nessa bateria de testes, concentrou-se a atenção no modelo de produtos, executando um total de 3.000 requisições HTTP. A escolha do modelo de produtos foi determinada pelo número de objetos únicos presentes no conjunto de dados, além de conter mais atributos, conforme descrito na subseção 3.1.3.1. Ademais, para simular um tráfego intenso, as chamadas foram realizadas paralelamente a cada 15 consultas, isto é, a cada 15 consultas enviadas a próxima parcela de 15 consultas eram enviadas paralelamente e não serial, usando uma máquina Linux com o comando “xargs”.

Tabela 4 – Trecho das capturas do teste de requisições para aplicação que utiliza o OGM desenvolvido em milissegundos (ms)

response_time	response_status
25,63	200
23,63	200
25,04	503

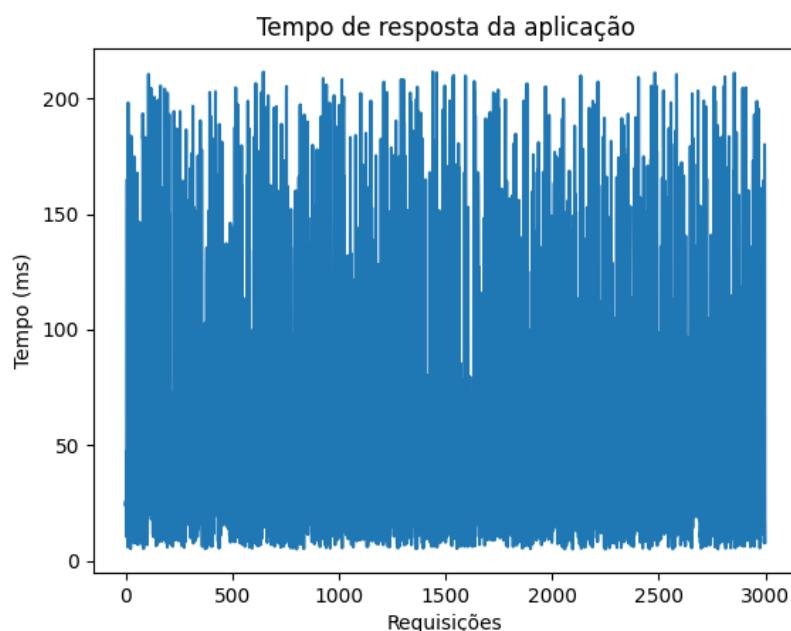
Fonte: Elaborado pelo autor.

Tabela 5 – Trecho das capturas do teste de requisições para aplicação que utiliza o ORM desenvolvido em milissegundos (ms)

response_time	response_status
53,83	200
17,47	200
44,35	200

Fonte: Elaborado pelo autor.

Figura 68 – Tempo de respostas da aplicação usando o OGM desenvolvido



Fonte: Elaborado pelo autor.

Figura 69 – Proporção dos *status* das respostas da aplicação usando o OGM.

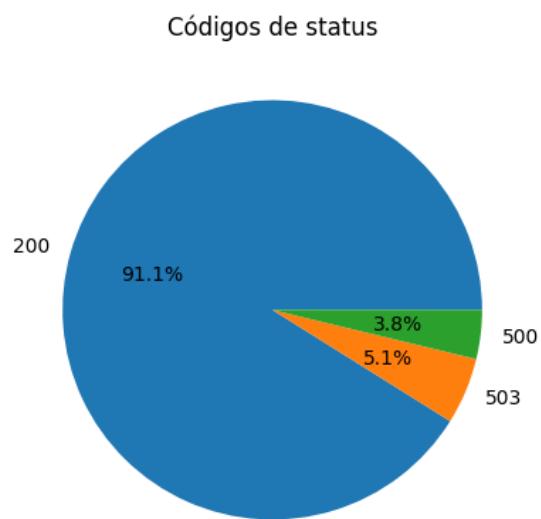
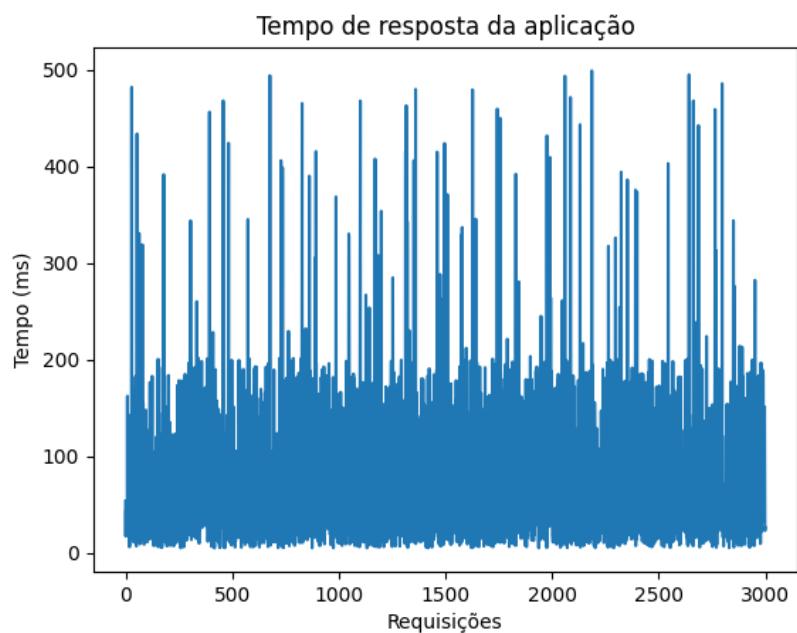
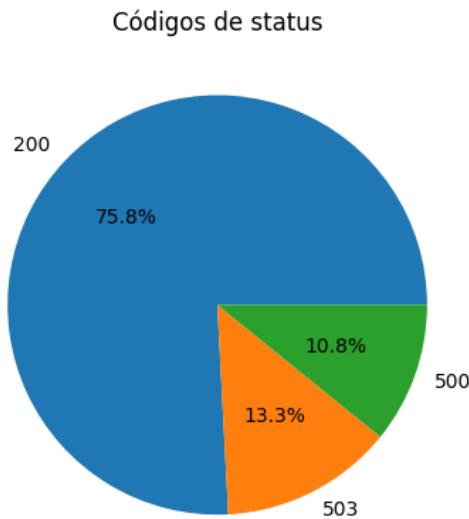


Figura 70 – Tempo de respostas da aplicação usando o Prisma



Fonte: Elaborado pelo autor.

Figura 71 – Proporção dos *status* das respostas da aplicação usando o Prisma



Fonte: Elaborado pelo autor.

4.1.4 Experimentos sobre o banco de dados

Esta subseção dedica-se à exploração de experimentos sobre os bancos de dados subjacentes utilizados na aplicação.

4.1.4.1 PostgreSQL

Foram executadas funções sobre o banco dados para coletar informações sobre o banco dados.

Figura 72 – Informações sobre o banco de dados PostgreSQL usando “\l+”

List of databases									Description
Name	Owner	Encoding	Collate	Ctype	Access privileges	Size	Tablespace		
ecommerce	root	UTF8	en_US.utf8	en_US.utf8		42 GB	pg_default		
postgres	root	UTF8	en_US.utf8	en_US.utf8		7337 kB	pg_default	default administrative connection database	
template0	root	UTF8	en_US.utf8	en_US.utf8	=c/root	+ 7337 kB	pg_default	unmodifiable empty database	
template1	root	UTF8	en_US.utf8	en_US.utf8	=c/root	+ 7401 kB	pg_default	default template for new databases	

(4 rows)

Fonte: Elaborado pelo autor.

4.1.4.2 Neo4j

Para obter informações relevantes sobre o estado do Neo4j, foi executada a função “:sysinfo”, que forneceu detalhes sobre a configuração, a capacidade e o desempenho do banco de dados.

Quadro 4 – Informações sobre o banco de dados Neo4j usando “:sysinfo”

Store Size					
Total	979.07 MiB				
Database	205.35 MiB				
Id Allocation					
Node ID	568439				
Property ID	2229879				
Relationship ID	2068169				
Relationship Type ID	5				
Page Cache					
Hits	1521742				
Page Faults	21990				
Hit Ratio	100.00%				
Usage Ratio	33.60%				
Transactions					
Last Tx Id	270				
Current Read	1				
Current Write	0				
Peak Transactions	2				
Committed Read	21				
Committed Write	0				
Databases					
Name	Address	Role	Status	Default	Error
ecommerce	localhost:7687	primary	online	-	-
neo4j	localhost:7687	primary	online	true	-
system	localhost:7687	primary	online	-	-
test	localhost:7687	primary	online	-	-

Fonte: Elaborado pelo autor.

4.2 Análise dos resultados

4.2.1 Análise do mapeador desenvolvido

Com base nos experimentos conduzidos na seção 4.1, em particular na subseção 4.1.2, e nas métricas apresentadas na Tabela 2, é possível realizar uma análise comparativa do desempenho e da eficiência dos processos de tradução de consultas entre o Prisma e o OGM desenvolvido.

Tabela 6 – Análise das métricas do mapeador relacional(3).

Métrica	Tempo (ms)
Média de geração de consulta	2,80
Média de tempo de resposta de banco de dados	79,11
Mínimo de geração de consulta	1,00
Máximo de geração de consulta	5,10
Desvio padrão de geração de consulta	1,22
Variância de geração de consulta	1,48

Fonte: Elaborada pelo autor.

Tabela 7 – Análise das métricas do mapeador desenvolvido(2).

Métrica	Tempo (ms)
Média de instanciação	0,54
Média de geração de consulta	2,51
Média de tempo de resposta de banco de dados	93,42
Mínimo de instanciação	0,10
Mínimo de geração de consulta	1,10
Máximo de geração de consulta	4,21
Desvio padrão de geração de consulta	0,92
Variância de geração de consulta	0,85

Fonte: Elaborada pelo autor.

Primeiramente, com relação ao processo de tradução fica evidente que o algoritmo baseado do Knex.js, utilizado na aplicação que faz de Cypher, oferece um desempenho mais rápido em comparação com o uso da linguagem do SQL no PostgreSQL. Isso se deve, em parte, ao fato de que consultas simples, como a seleção completa de um modelo de dados, requerem apenas a ativação de três funções aninhadas com programação funcional no mapeador desenvolvido (FIDELIS, 2023).

Além disso, é possível observar que o processo de tradução no mapeador tem limites de tempo estabelecidos em 1,10ms e 4,21ms, enquanto, para o Prisma os limites são 1,00ms e 5,10ms. No entanto, ao analisar o desempenho geral é possível notar que o tempo de execução total do prisma é menor do que o tempo de execução do OGM desenvolvido, isso se deve, em parte, ao tempo de resposta do banco de dados PostgreSQL, que registrou uma média de 79,11ms, enquanto o tempo de resposta do banco de dados Neo4j ficou em 93,42ms, conforme indicado na Tabela 6 e em Tabela 7. Ademais, usando a Figura 66 e Figura 67 é possível destacar que o processo todo é atrasado por conta do tempo de resposta do banco de dados subjacente.

O OGM desenvolvido demonstrou um desempenho superior em comparação com o Prisma em uma série de indicadores. Um destes é a métrica de desvio padrão, que serve para medir a consistência no tempo de geração de consultas. Notavelmente, o OGM demonstrou

um desvio padrão de apenas 0,92ms, enquanto o ORM registrou um valor mais elevado de 1,22ms. Esses resultados sugerem que o processo de geração de consultas pelo OGM é mais previsível e consistente, o que é crucial para manter a eficiência e a baixa latência nas operações da aplicação. Além disso, a variância, que mede a dispersão dos valores em torno da média, também mostra uma vantagem clara para o OGM, enquanto o ORM apresentou uma variância de 1,48ms, o OGM exibiu uma variância ligeiramente inferior, de 0,85ms, indicando que o OGM tem uma variabilidade menor e tende a manter um desempenho mais estável em relação à geração de consultas.

4.2.1.1 Análise de praticidade

Para a análise da praticidade do OGM, considerando o desenvolvimento das aplicações como descritas em seção 3.3, torna-se evidente que ambos os mapeadores possuem méritos distintos.

O OGM desenvolvido se destaca pela sua interface que tornou prático desenvolver um *concern* e mantendo a tipagem estática usando inferência, que resulta uma experiência de desenvolvimento mais intuitiva, especialmente utilizando ferramentas de ambiente de desenvolvimento, como o *autocomplete*.

Por outro lado, a integração do Prisma apresentou problemas como descrito no final da subseção 3.3.4. Sobretudo devido à disponibilidade de módulos prontos para o *framework* NestJS. Essa integração oferece arquiteturas robustas, colaborando na implementação de funcionalidades avançadas. Contudo, é importante ressaltar que a ferramenta desenvolvida foi criada usando desenvolvimento de código aberto como descrito na subseção 2.6.5, o que implica que, à medida que o projeto evolui, a contribuição de desenvolvedores externos pode proporcionar módulos adicionais equivalentes, conforme ocorre com o Prisma, simplificando ainda mais a implementação de recursos complexos. Assim, à medida que a ferramenta ganha apoio e contribuições, a perspectiva de facilitar a integração e a implementação de funcionalidades avançadas se torna uma realidade.

4.2.2 Análise das aplicações

Com base nos experimentos detalhados em subseção 4.1.3 e nos resultados apresentados na Tabela 8b e Tabela 8a, conduzimos uma análise minuciosa do desempenho das aplicações, considerando diversos indicadores e métricas essenciais.

A Tabela 8, agrega as métricas coletadas, oferece informações sobre o desempenho comparativo das aplicações que fazem uso do OGM e do Prisma. Destacam-se particularmente as médias de tempo de resposta, que indicam o tempo médio que cada aplicação leva para atender a uma solicitação. A aplicação com o OGM apresenta uma média de resposta notavelmente inferior, registrando 52,39ms, enquanto a aplicação com Prisma registra uma média de

76,42ms. Isso sugere que a aplicação com o OGM é mais ágil e responsiva a relação à entrega de resultados.

Tabela 8 – Métricas coletas das aplicações

(a) Métricas sobre o tempo de resposta do teste de requisições da aplicação usando o Prisma

Métrica	Tempo (ms)
Média	76,42
Mediana	50,93
Desvio padrão	73,77
Variância	5442,48
Máximo	499,03
Mínimo	5,17

(b) Métricas sobre o tempo de resposta do teste de requisições da aplicação usando o OGM

Métrica	Tempo (ms)
Média	52,39
Mediana	26,60
Desvio padrão	57,01
Variância	3250,13
Máximo	211,65
Mínimo	5,00

Fonte: Elaborado pelo autor.

Os limites de tempo também são indicativos do comportamento das aplicações de pico ou sob uso intenso. A aplicação com o OGM tem limites estabelecidos em 5ms (mínimo) e 211,65ms (máximo), demonstrando um intervalo de tempo aceitável para a maioria das operações. Por outro lado, a API usando o Prisma possui limites mais amplos, variando de 5,17ms a 499,03ms.

No que diz respeito à variância, a aplicação com o Prisma possui uma variância de 5442,48ms, enquanto, a aplicação com o OGM registra uma variância de 3250,13ms, sugerindo que a aplicação com o Prisma tem uma gama mais ampla de tempos de resposta, o que pode ser associado a uma maior variabilidade no desempenho. Assim, associando ao desvio padrão é possível identificar que a API usando Prisma, contém uma maior variabilidade nos tempos de resposta, em contraste com a aplicação usando o OGM com um desvio menor, 57,01ms, sugerindo ser mais consistente.

Ao comparar as porções dos *status* de respostas obtidas pelas APIs utilizando a Figura 69 e Figura 71 que utilizam o OGM desenvolvido e o ORM Prisma, respectivamente, notamos algumas diferenças significativas. No caso do OGM, aproximadamente 91,1% das respostas retornaram sucesso com o *status* “200 OK”, o que indica que a maioria das solicitações foi bem sucedida. Por outro lado, apenas 3,8% resultaram um erro com o *status* “500 Internal Server Error” e 5,1% em um *status* “503 Service Unavailable”, sugerindo que a aplicação manteve uma taxa relativamente baixa de erros críticos, considerando o formato que a bateria de requisições foi executada descrita em subseção 4.1.3.

4.2.3 Análise do banco de dados

Como base nos experimentos detalhados na subseção 4.1.4, os dados coletados fornecem informações como o espaço em disco dos bancos de dados e a eficácia do uso de *cache*,

conforme a Quadro 4 e Figura 72.

Foi observado que o banco de dados Neo4j, que armazena os dados orientadamente a grafos, ocupa uma capacidade de aproximadamente 205,35 MiB, que, convertido para gigabytes, corresponde a cerca de 21,53 GB. Por outro lado, o banco de dados PostgreSQL, responsável pelo armazenamento de dados, relacionais, registrou um tamanho de cerca de 42 GB, conforme indicado na Figura 72. Essa discrepância no tamanho dos bancos de dados ressalta as diferenças fundamentais na estrutura e no modelo de armazenamento de dados entre os dois sistemas.

Além disso, o indicador “*hitratio*” atinge 100%, o qual é uma métrica que representa a taxa de acerto nas operações de leitura em *cache*. Uma taxa de 100% indica que todas as leituras foram atendidas a partir do *cache*, o que é um indicativo positivo extremamente de uma eficaz otimização de *cache* e do desempenho do banco de dados.

Essas análises demonstram as diferenças notáveis na arquitetura dos bancos de dados utilizados nas aplicações, além de destacar o sucesso das otimizações realizadas na subseção 3.4.1.2.1.

5 Considerações Finais

O objetivo principal deste trabalho foi desenvolver e aplicar um OGM para um banco de dados baseado em grafos, visando aprimorar a manipulação e recuperação de dados. Essa abordagem foi motivada pela complexidade envolvida na consulta e interpretação dos dados em bancos de dados orientados a grafos, bem como pela necessidade de incorporar eficazmente essas bases de dados aos ecossistemas de ferramentas e aplicações existentes. O intuito foi criar uma interface intuitiva e eficiente que simplificasse o processo de desenvolvimento de aplicações que se conectem a esse tipo específico de banco de dados, tornando-o mais acessível e prático.

Neste contexto, todos os objetivos propostos foram alcançados com êxito. A revisão bibliográfica e de mercado realizada permitiu a compreensão aprofundada das características e desafios associados aos OGMs e aos banco de dados, fornecendo o embasamento necessário para o desenvolvimento. Conforme as avaliações na seção 4.2 o OGM desenvolvido em comparação com outras soluções existentes no mercado, revelou resultados positivos, demonstrando ser uma alternativa viável com uma média de tempo de resposta, tradução e limites de tempos mais restritos e rápidos em comparação com outras soluções.

Além disso, a aplicação prática do OGM desenvolvido em um projeto real validou sua eficácia na manipulação e recuperação de dados, confirmado sua praticidade e utilidade no desenvolvimento de aplicações.

5.1 Trabalhos Futuros

Para trabalhos futuros, há uma série de melhorias e expansões a serem consideradas, incluindo a implementação de suporte para mais tipos de dados na linguagem de definição, e demais módulos, a adição de funcionalidades avançadas na definição de dados como conceito de migrações e a criação de módulo de otimização na criação de transações. Ademais, aprimoramentos na geração de interfaces de tipos estáticos usando o conceito de geração de módulos completos, como realizado no Prisma, além da capacidade de importar formatadores externos e o suporte aprimorado à paginação do banco de dados.

Referências

- ANGLES, R.; GUTIERREZ, C. Survey of graph database models. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, v. 40, n. 1, feb 2008. ISSN 0360-0300. Disponível em: <https://doi.org/10.1145/1322432.1322433>. Acesso em: 18 mai. 2023.
- API, J. **JSON:API Specification**. 2022. Disponível em: <https://jsonapi.org/format/>. Acesso em: 29 out. 2023.
- BROOKS, F. **O mítico homem-mês: ensaios sobre engenharia de software**. Alta Books, 2018. ISBN 9788550802534. Disponível em: <https://books.google.com.br/books?id=jjj-zwEACAAJ>. Acesso em: 24 mai. 2023.
- BRUCE, K. B. A paradigmatic object-oriented programming language: Design, static typing and semantics. **Journal of Functional Programming**, Cambridge University Press, v. 4, n. 2, p. 127–206, 1994. Disponível em: <https://www.cambridge.org/core/journals/journal-of-functional-programming/article/paradigmatic-object-oriented-programming-language-design-and-static-typing-and-semantics/DBA72A6F6E3E0235CA2321C822BC000E>. Acesso em: 2 nov. 2023.
- COPELAND, G.; MAIER, D. Making smalltalk a database system. **SIGMOD Rec.**, Association for Computing Machinery, New York, NY, USA, v. 14, n. 2, p. 316325, jun 1984. ISSN 0163-5808. Disponível em: <https://doi.org/10.1145/971697.602300>. Acesso em: 4 abr. 2023.
- COWLEY, A. **Neode**. GitHub, 2017. Disponível em: <https://github.com/adam-cowley/neode>. Acesso em: 25 out. 2023.
- COX, B. J. **Object oriented programming: an evolutionary approach**. Addison-Wesley Longman Publishing Co., Inc., 1986. Disponível em: <https://dl.acm.org/doi/abs/10.5555/16111>. Acesso em: 20 out. 2023.
- CROSS-PLATFORM. **The Cambridge Dictionary**. Cambridge University Press & Assessment, 2023. Disponível em: <https://dictionary.cambridge.org/us/dictionary/english/cross-platform>. Acesso em: 22 mai. 2023.
- DIETZE, F.; KAROFF, J.; VALDEZ, A. C.; ZIEFLE, M.; GREVEN, C.; SCHROEDER, U. An open-source object-graph-mapping framework for neo4j and scala: Renesca. In: SPRINGER. **Availability, Reliability, and Security in Information Systems: IFIP WG 8.4, 8.9, TC 5 International Cross-Domain Conference, CD-ARES 2016, and Workshop on Privacy Aware Machine Learning for Health Data Science, PAML 2016, Salzburg, Austria, August 31-September 2, 2016, Proceedings**. 2016. p. 204–218. Disponível em: <https://www.springerprofessional.de/en/an-open-source-object-graph-mapping-framework-for-neo4j-and-scal/10605490>. Acesso em: 2 nov. 2023.
- ELMASRI, R. **Sistemas De Banco De Dados**. 7. ed. Pearson, 2019. ISBN 8543025001, 9788543025001. Disponível em: <http://gen.lib.rus.ec/book/index.php?md5=F3F82290582BB19EDB264555E5A15182>. Acesso em: 13 out. 2023.

FIDELIS, W. **PG Driver vs Knex.js vs Sequelize vs TypeORM**. Medium, 2023. Disponível em: <https://washingtonfidelis.medium.com/pg-driver-vs-knex-js-vs-sequelize-vs-typeorm-f9ed53e9f802>. Acesso em: 25 out. 2023.

FRANCIS, N.; GREEN, A.; GUAGLIARDO, P.; LIBKIN, L.; LINDAAKER, T.; MARSAULT, V.; PLANTIKOW, S.; RYDBERG, M.; SELMER, P.; TAYLOR, A. Cypher: An evolving query language for property graphs. In: **Proceedings of the 2018 International Conference on Management of Data**. New York, NY, USA: Association for Computing Machinery, 2018. (SIGMOD '18), p. 14331445. ISBN 9781450347037. Disponível em: <https://doi.org/10.1145/3183713.3190657>. Acesso em: 2 nov. 2023.

FROST, R.; HAFIZ, R.; CALLAGHAN, P. Modular and efficient top-down parsing for ambiguous left-recursive grammars. In: **Proceedings of the Tenth International Conference on Parsing Technologies**. [s.n.], 2007. p. 109–120. Disponível em: <https://dl.acm.org/doi/pdf/10.5555/1621410.1621425>. Acesso em: 26 mai. 2023.

GACKENHEIMER, C. **Introduction to React**. Apress, 2015. Disponível em: <https://link.springer.com/book/10.1007/978-1-4842-1245-5>. Acesso em: 26 out. 2023.

GR1D. **Swagger: O que é e como usar**. Gr1d, 2022. Disponível em: <https://gr1d.io/2022/04/15/swagger/>. Acesso em: 25 out. 2023.

GUIDE, P. Getting started with big data. **Intel IT Center**, 2013. Disponível em: <https://www.intel.com/content/dam/www/public/emea/fr/fr/documents/guides/getting-started-with-hadoop-planning-guide.pdf>. Acesso em: 2 nov. 2023.

GÜTING, R. H. Graphdb: Modeling and querying graphs in databases. In: CITESEER. **VLDB**. 1994. v. 94, p. 12–15. Disponível em: <https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=0f9bb7d25477862558ba5491c4357e7b8fc65a16>. Acesso em: 2 nov. 2023.

GYSSENS, M.; PAREDAENS, J.; GUCHT, D. V. A graph-oriented object database model. In: **Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems**. [s.n.], 1990. p. 417–424. Disponível em: <https://dl.acm.org/doi/10.1145/298514.298593>. Acesso em: 2 nov. 2023.

HAUGE, Ø.; AYALA, C.; CONRADI, R. Adoption of open source software in software-intensive organizations—a systematic literature review. **Information and Software Technology**, Elsevier, v. 52, n. 11, p. 1133–1154, 2010. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0950584910000972>. Acesso em: 12 out. 2023.

JUNEAU, J.; JUNEAU, J. Object-relational mapping. **Java EE 7 Recipes: A Problem-Solution Approach**, Springer, p. 369–408, 2013. Disponível em: https://link.springer.com/chapter/10.1007/978-1-4302-4426-4_8. Acesso em: 2 nov. 2023.

KECHINOV, M. **Ecommerce Behavior Data from Multi-Category Store**. 2023. Disponível em: <https://www.kaggle.com/datasets/mkechinov/ecommerce-behavior-data-from-multi-category-store?select=2019-Oct.csv>. Acesso em: 29 out. 2023.

KNUTH, D. E. Semantics of context-free languages. **Mathematical systems theory**, Springer, v. 2, n. 2, p. 127–145, 1968. Disponível em: <https://link.springer.com/article/10.1007/BF01692511>. Acesso em: 23 out. 2023.

- LAWSON, L. **Express.js vs. Fastify: A Showdown Between Two Popular Node.js Web App Frameworks.** The New Stack, 2022. Disponível em: <https://thenewstack.io/a-showdown-between-express-js-and-fastify-web-app-frameworks/>. Acesso em: 25 out. 2023.
- LIBKIN, L.; MARTENS, W.; VRGOČ, D. Querying graphs with data. **Journal of the ACM (JACM)**, ACM New York, NY, USA, v. 63, n. 2, p. 1–53, 2016. Disponível em: <https://dl.acm.org/doi/abs/10.1145/2850413>. Acesso em: 2 nov. 2023.
- LU, t. t. **Análise exploratória de dados e engenharia de recursos para comércio eletrônico**. Kaggle, 2022. Disponível em: <https://www.kaggle.com/code/luthien5921/commerce-eda-and-feature-engineering/notebook>. Acesso em: 29 out. 2023.
- MARTIN, R. C. **The Principles of Object-Oriented Design**. ButUncleBob.com, 2009. Disponível em: <http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>. Acesso em: 20 out. 2023.
- MELTON, J.; SIMON, A. R. **SQL: 1999: understanding relational language components**. Elsevier, 2001. Disponível em: <https://dl.acm.org/doi/10.5555/377430>. Acesso em: 2 nov. 2023.
- MENG, X.; CAI, X.; CUI, Y. Analysis and introduction of graph database. In: **2021 3rd International Conference on Artificial Intelligence and Advanced Manufacture**. New York, NY, USA: Association for Computing Machinery, 2022. (AIAM2021), p. 392396. ISBN 9781450385046. Disponível em: <https://doi.org/10.1145/3495018.3495086>. Acesso em: 12 abr. 2023.
- MICROSOFT. **Documentação do TypeScript**. Microsoft, 2023. Disponível em: <https://www.typescriptlang.org>. Acesso em: 14 out. 2023.
- MICROSOFT. **IntelliSense no Visual Studio Code**. Microsoft, 2023. Disponível em: <https://code.visualstudio.com/docs/editor/intellisense>. Acesso em: 25 out. 2023.
- MONIRUZZAMAN, A. B. M.; HOSSAIN, S. A. **NoSQL Database: New Era of Databases for Big data Analytics - Classification, Characteristics and Comparison**. 2013. Disponível em: <https://arxiv.org/abs/1307.0191>. Acesso em: 4 abr. 2023.
- MOZILLA. **Introdução ao JavaScript**. Mozilla Developer Network, 2023. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Guide/Introduction>. Acesso em: 25 out. 2023.
- NAVATHE, S. B. Evolution of data modeling for databases. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 35, n. 9, p. 112123, sep 1992. ISSN 0001-0782. Disponível em: <https://doi.org/10.1145/130994.131001>. Acesso em: 4 abr. 2023.
- NEOWAY. **O que são datasets?** 2023. Disponível em: <https://blog.neoway.com.br/dataset/#:~:text=Datasets%20s%C3%A3o%20conjuntos%20de%20dados,cont%C3%A9m%20info,rma%C3%A7%C3%B5es%20sobre%20determinado%20tema>. Acesso em: 29 out. 2023.
- PAREDAENS, J.; PEELMAN, P.; TANCA, L. G-log: a graph-based query language. **IEEE Transactions on Knowledge and Data Engineering**, v. 7, n. 3, p. 436–453, 1995. Disponível em: <https://ieeexplore.ieee.org/abstract/document/390249>. Acesso em: 2 nov. 2023.

PIATTINI, M.; CALERO, C.; SAHRAOUI, H.; LOUNIS, H. Object-relational database metrics. **L'Object**, vol. 2001. Disponível em: http://www.iro.umontreal.ca/~sahraouh/papers/lobjet00_1.pdf. Acesso em: 2 nov. 2023.

PRESSMAN, R. S.; MAXIM, B. R. **Engenharia de software-9**. McGraw Hill Brasil, 2021. Disponível em: <https://github.com/AlexGalhardo/Software-Engineering/blob/master/Book%20-%20Engenharia%20de%20Software%20-%20Sommerville%209%20Edicao.pdf>. Acesso em: 2 nov. 2023.

PRISMA. **Prisma**. NPM, 2023. Disponível em: <https://www.npmjs.com/package/prisma>.

PRISMA. **Prisma schema**. Prisma, 2023. Disponível em: <https://www.prisma.io/docs/concepts/components/prisma-schema>. Acesso em: 20 jun. 2023.

RAILS. **Concerns**. Ruby on Rails, 2023. Disponível em: <https://api.rubyonrails.org/classes/ActiveSupport/Concern.html>. Acesso em: 25 out. 2023.

RAMAKRISHNAN, R.; GEHRKE, J. **Sistemas de gerenciamento de banco de dados - 3.ed.** McGraw Hill Brasil, 2008. ISBN 9788563308771. Disponível em: <https://books.google.com.br/books?id=COUJpkH5v38C>. Acesso em: 12 abr. 2023.

SADALAGE, P.; FOWLER, M. **NoSQL Essencial: Um Guia Conciso para o Mundo Emergente da Persistência Poliglota**. Novatec Editora, 2013. ISBN 9788575223383. Disponível em: <https://books.google.com.br/books?id=m6jiDQAAQBAJ>. Acesso em: 18 abr. 2023.

SAGIROGLU, S.; SINANC, D. Big data: A review. In: **2013 International Conference on Collaboration Technologies and Systems (CTS)**. [s.n.], 2013. p. 42–47. Disponível em: <https://ieeexplore.ieee.org/document/6567202>. Acesso em: 2 nov. 2023.

SEBESTA, R. W. **Conceitos de Linguagens de Programação-11**. Bookman Editora, 2018. Disponível em: [https://github.com/filipanselmo11/Lisp/blob/master/Conceitos%20de%20Linguagem%20de%20Programa%C3%A7%C3%A3o%20\(9a%20Edi%C3%A7%C3%A3o\)%20-%20Robert%20W.%20Sebesta%20-%20trabalho.pdf](https://github.com/filipanselmo11/Lisp/blob/master/Conceitos%20de%20Linguagem%20de%20Programa%C3%A7%C3%A3o%20(9a%20Edi%C3%A7%C3%A3o)%20-%20Robert%20W.%20Sebesta%20-%20trabalho.pdf). Acesso em: 23 out. 2023.

SILVA, D. A. d. et al. Gestão de projetos de software. Florianópolis, SC, 2001. Disponível em: <https://repositorio.ufsc.br/handle/123456789/79550>. Acesso em: 24 mai. 2023.

SINGH, S.; SINGH, N. Big data analytics. In: **2012 International Conference on Communication, Information Computing Technology (ICCICT)**. [s.n.], 2012. p. 1–4. Disponível em: <https://ieeexplore.ieee.org/document/6398180>. Acesso em: 2 nov. 2023.

TORRES, A.; GALANTE, R.; PIMENTA, M. S.; MARTINS, A. J. B. Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design. **information and software technology**, Elsevier, v. 82, p. 1–18, 2017. Disponível em: https://www.sciencedirect.com/science/article/pii/S0950584916301859?casa_token=Pu6h-TtnW1wAAAAA:RNg_5T--05gkArJVla_UHF0sljeSfNDAPikaUA9Vxybqcdq2rUyHc6jtBXxiPzC7D4oIHACH0g. Acesso em: 2 nov. 2023.

WU, M.-W.; LIN, Y.-D. Open source software development: an overview. **Computer**, v. 34, n. 6, p. 33–38, 2001. Disponível em: <https://dl.acm.org/doi/10.1109/2.928619>. Acesso em: 10 out. 2023.

ZHYKHARSKYI, P.; GUBARYEVA, O. Pure sql vs orm libraries: A comparative analysis for api applications. **Collection of scientific papers**, n. June 30, 2023; Helsinki, Finland, p. 9798, Jul. 2023. Disponível em: <https://previous.scientia.report/index.php/archive/article/view/1063>. Acesso em: 2 nov. 2023.