



CRIAÇÃO DE UMA FERRAMENTA PARA OTIMIZAÇÃO E SIMPLIFICAÇÃO NO GERENCIAMENTO DE BANCO DE DADOS BASEADOS EM GRAFOS EM AMBIENTES DE BIG DATA

Cassiano Henrique Aparecido Rodrigues
(Bacharelado em Ciências da Computação — UNESP / Bauru)

Prof. Assoc. Dr. Aparecido Nilceu Marana
(Orientador)

Sumário

Table of Contents

Proposta

04'

- Problemática
- Justificativa
- Objetivos

Experimentos

33'

- Mapeador
 - Análise
- Aplicação
 - Análise

Fund. Teórica

10'

- O que é mapeador
- O que é grafo
- Banco de dados

Considerações

44'

- Trabalhos futuros

OGM

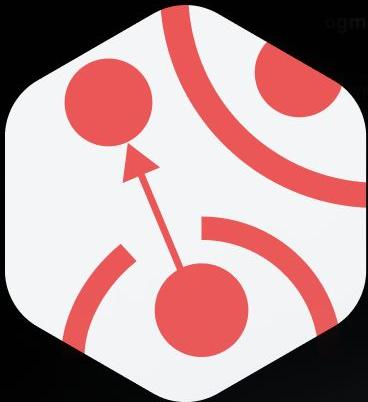
17'

- Planejamento
- Estrutura
- Módulos
- Linguagem DDL



Cassiano Rodrigues

Estudante de Ciências da Computação
Desenvolvedor Frontend



Mapeador Objeto-Grafo

ogm-neo4j Public

Actions Projects Wiki Security Insights Settings

Pin Unwatch 1 Fork 0

2 branches 0 tags Go to file Add file Code

About

Object-Graph-Mapping for easy and flexible node and operations

MIT license

Activity

0 stars

1 watching

0 forks

Releases

No releases published Create a new release

Packages

No packages published Publish your first package

Languages

TypeScript 100.0% 3

repository understand your project by adding a README.

Add a README



PROBLEMÁTICA

Assim como pontuado por Moniruzzaman e Hossain (2013), a **quantidade de dados** gerados pelas empresas vem **aumentando exponencialmente**, gerando uma crescente preocupação em relação ao **armazenamento, processamento e análise** desses dados.

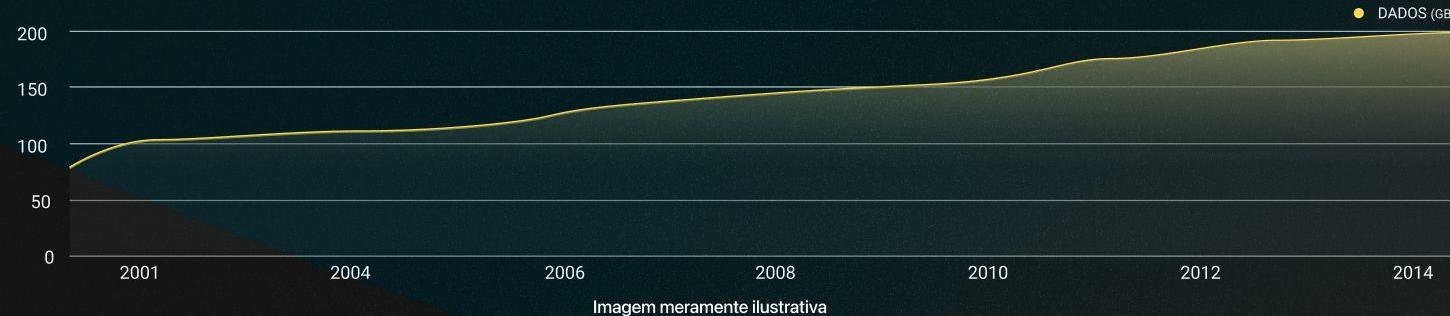


Imagen meramente ilustrativa



PROBLEMÁTICA

Desta forma, soluções e alternativas estão sendo implementadas como:

Banco de dados alternativos aos **relacionais** é uma das possibilidades adotadas, uma vez que a modelagem dos dados pode afetar o desempenho e uso de espaço.



Alternativos



Escalabilidade



Servidores



PROBLEMÁTICA

Desta forma, soluções e alternativas estão sendo implementadas como:

Banco de dados alternativos aos **relacionais** é uma das possibilidades adotadas, uma vez que a modelagem dos dados pode afetar o desempenho e uso de espaço.



Alternativos



Escalabilidade



Servidores



PROBLEMÁTICA

Diante desse contexto, os **bancos de dados baseados em grafos** têm se mostrado uma **alternativa viável**. Eles são especialmente eficientes em lidar com dados que possuem relacionamentos complexos e variedades de tipos de dados.

No entanto, a utilização desses bancos é um **desafio para o desenvolvimento**, devido à **dificuldade de integrar as consultas e definição dos dados com segurança**, uma vez que não possui validação na maioria dos gerenciadores, além da **dificuldade em integrar às ferramentas de alto nível**, o que dificulta sua adoção.



Banco de Dados
Baseado em grafo

COMPLEXIDADE



Usuários



JUSTIFICATIVA

Nesse sentido a **criação** de um **mapeador objeto-grafo** (OGM) é uma **solução**.

Um OGM consegue mapear objetos e, assim:

- Permitir o acesso aos dados de maneira mais intuitiva e eficiente
- Facilitar o desenvolvimento e o manuseio dos dados,
- Fornece uma solução para integrar outras ferramentas disponíveis de alto nível.

Assim é criado uma **camada de abstração** entre o banco de dados e a aplicação.





OBJETIVOS

GERAL

O objetivo geral deste trabalho é desenvolver e aplicar um OGM para um banco de dados baseado em grafos.

ESPECÍFICOS



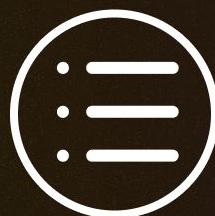
Revisão bibliográfica



Avaliar o banco



Avaliar o OGM



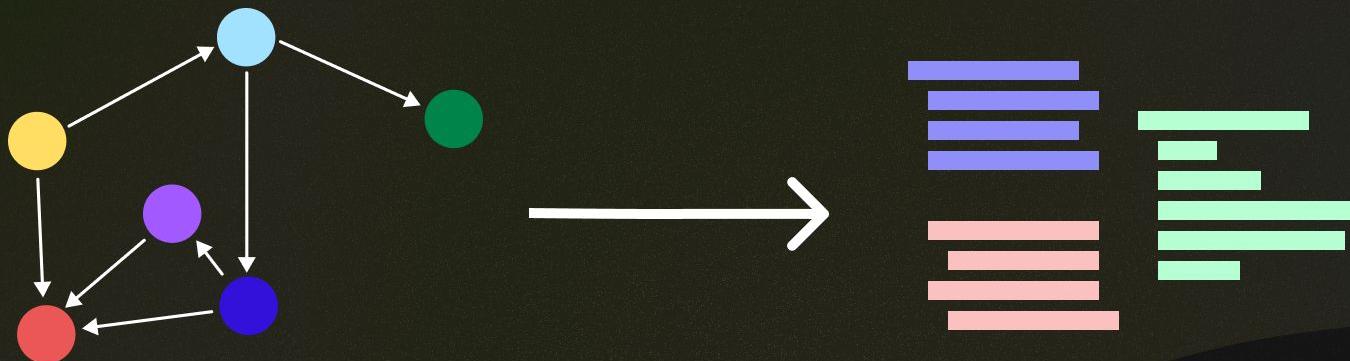
Projetar e criar



Implementar e validar

MAPEADOR

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** (Torres et al, 2017).



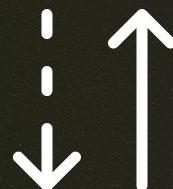


MAPEADOR

Os mapeadores executam diversas etapas elencadas a seguir:

- Conexão com o banco de dados;
- Mapeamento;
- Interface de interação com o banco de dados;
- Gerenciamento de transações; e
- Gerenciamento do esquema de dados.

Garantindo **integridade, otimização e praticidade**.

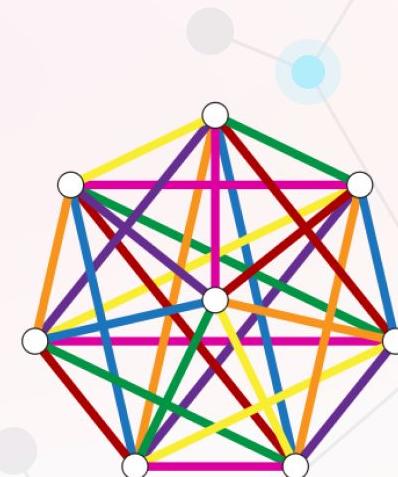


O QUE É GRAFO

Um grafo é uma que **representação abstrata** de um conjunto de objetos e das **relações existentes entre eles**. É definido por um **conjunto de nós ou vértices**, e pelas ligações ou arestas, que ligam pares de nós (Levada, A. L. M, 2020)



Fonte: Documentação do Neo4j

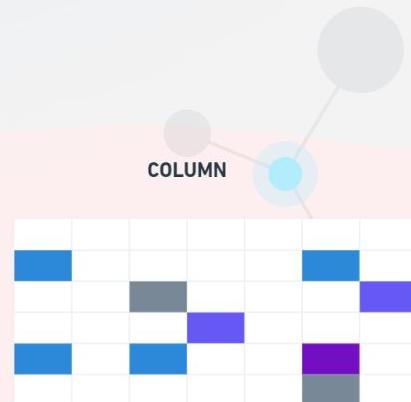
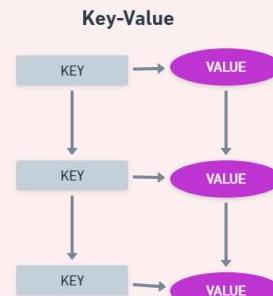
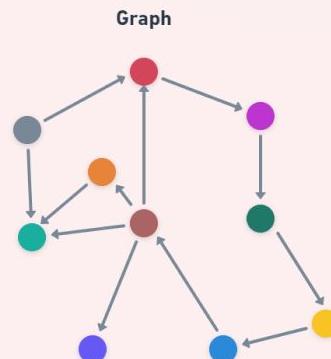
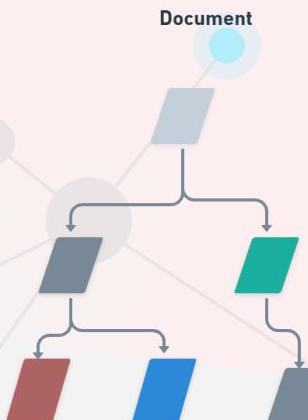


Fonte: Wikipédia Images

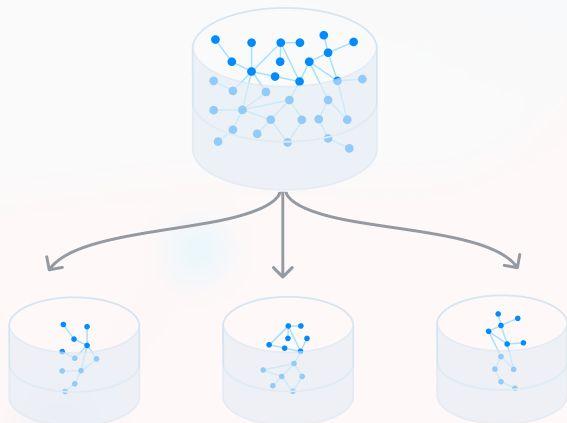


TIPOS DE BANCO DE DADOS

Banco de dados baseados em grafos são os **únicos (NoSQL)** que utilizam o **modelo de distribuição**, semelhante ao modelo relacional.



Fonte: Documentação do Neo4j



Fonte: Documentação do Neo4j

ESCALABILIDADE, GRANULARIDADE E DESEMPENHO

Escalabilidade: capacidade de um sistema lidar com um aumento na carga de trabalho

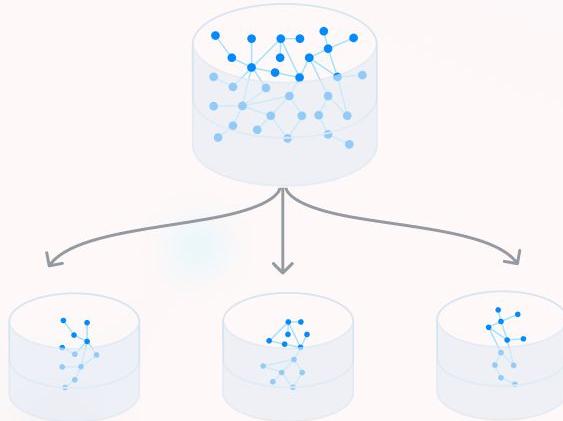
- **Vertical:** capacidade de manipular maior volume de dados e execução de consultas mais complexas (profundas).
- **Horizontal:** distribuição dos dados e consultas em vários nós.
- **Escalabilidade elástica:** capacidade de adicionar ou remover nós do cluster de **forma dinâmica**.



ESCALABILIDADE, GRANULARIDADE E DESEMPENHO

Granularidade: Nível em que as operações são realizadas no banco de dados: Nó, relacionamento e transações.

A granularidade escolhida pode afetar a concorrência.



Fonte: Documentação do Neo4j



MATERIAIS

Focados na simulação de um ambiente real de hospedagem de uma aplicação.



Ubuntu 22.04.3 LTS

i5-10400 CPU @ 2.90GHz

32 GB RAM DDR4

1TB, 512GB SSD M.2 NVMe // 4TB, 1TB HDs



Pop_OS 22.04

AMD Ryzen 7 5800H @ 3.20 GHz

NVIDIA GeForce GTX 1650 4GB GDDR6

20 GB RAM DDR4

512 e 256 GB SSD M.2 NVMe

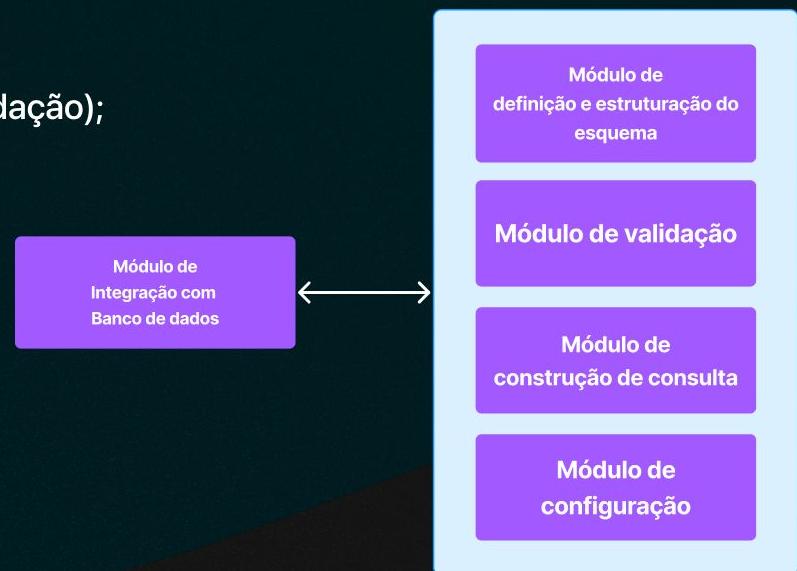


DESENVOLVIMENTO DO OGM

ESTRUTURA DO MAPEADOR

Com base na pesquisa de mercado em diversas linguagens de programação, características foram listadas.

- Uso de abstração para definição do esquema;
- Preocupação com a integridade dos dados (validação);
- Possibilidade de injeção de dependência;
- Interface para a abstração de consultas;
- Tratamentos de erros; e
- Reconhecimento de variáveis de ambiente.





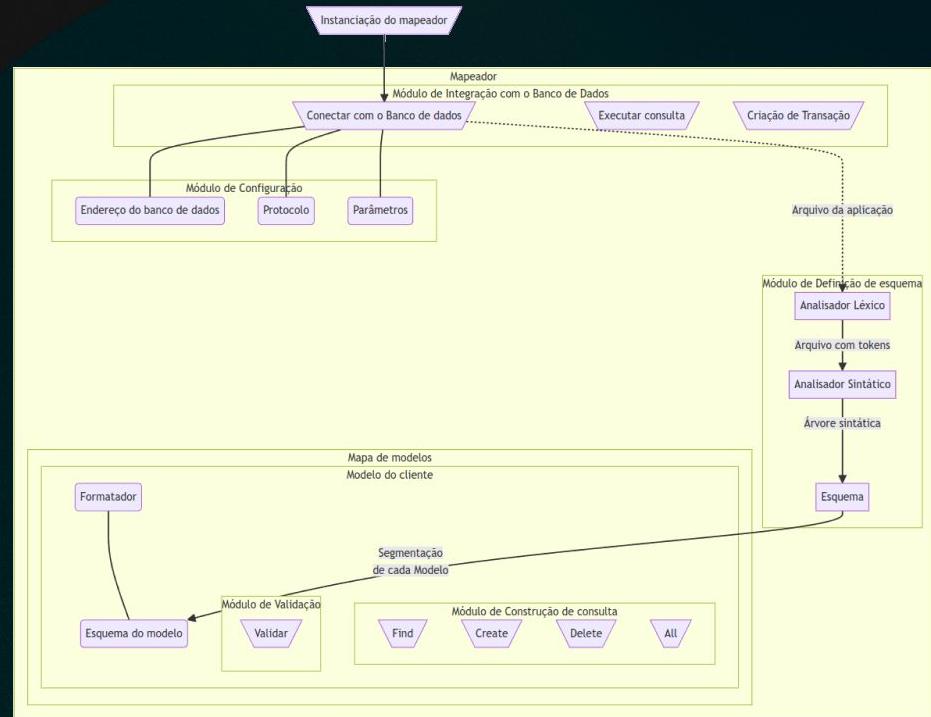
DESENVOLVIMENTO DO OGM

ESTRUTURA DO MAPEADOR

O fluxograma de interações dos módulos em conjunto com suas operações são descritas a seguir:

```
index.tsx
1 await prisma.user
2   .findUnique({
3     where: {email: 'ada@prisma.io' }
4   })
5   .posts({
6     where: {
7       title: {
8         |
9       }
10    }
11  })
```

A tooltip on the code editor highlights the `contains` operator.





MÓDULO DE INTEGRAÇÃO COM O BANCO DE DADOS



É o módulo responsável por interagir diretamente com o banco de dados e oferecer métodos para executar as consultas.

Utilizando o próprio **driver** do banco de dados é possível conectá-lo de maneira simples utilizando as configurações obtidas pelo **Módulo de Configuração**.



FONTE: ADAPTADO DA DOCUMENTAÇÃO DO NEO4J



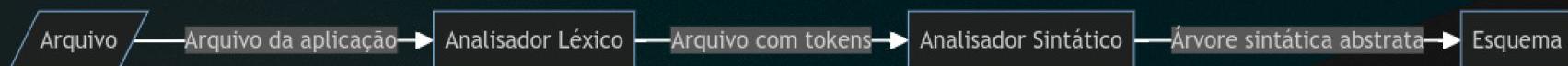


MÓDULO DE DEFINIÇÃO E ESTRUTURAÇÃO DO ESQUEMA

- = Desempenha a gestão de esquema de dados e a estruturação dos modelos do cliente, que envolve identificar os atributos e tipos definidos.
- = Foi construído uma linguagem de definição de dados, pensada para simplificar a adoção e usabilidade:

- **Sintaxe** análoga a utilizadas em **mapeadores relacionais**;
- Uso de **operadores**; e
- Capacidade de adicionar **restrições**.

Processo de análise do esquema





MÓDULO DE DEFINIÇÃO E ESTRUTURAÇÃO DO ESQUEMA

Após a **análise léxica e sintática** é construído a **árvore sintática abstrata** ao qual é possível **reconhecer e compreender o esquema da aplicação**.

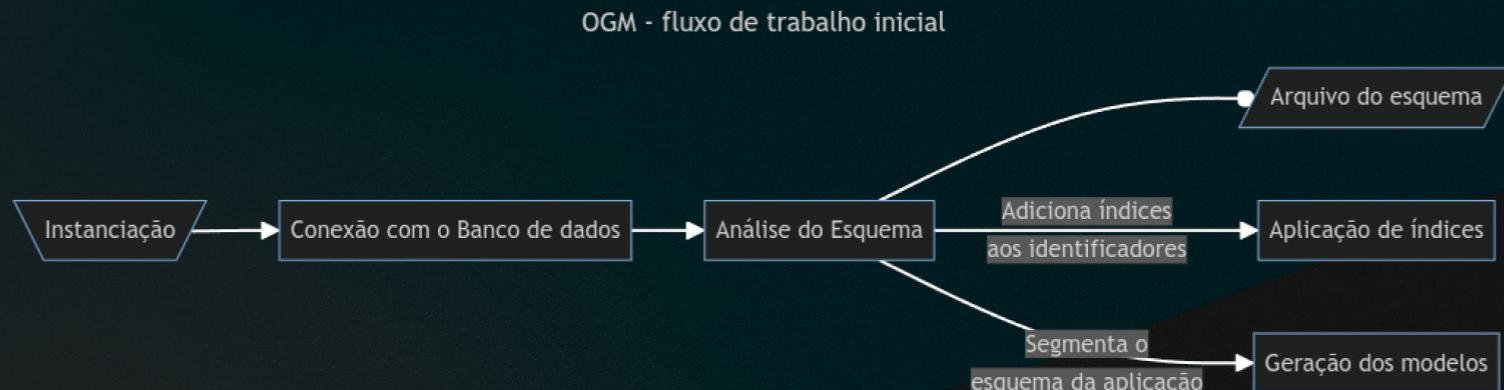
```
→ parser.schema
  ↘ {nodes: Map(10), relations: Map(1)}
    ↘ nodes: Map(10) {size: 10, User => {identifier: ..., ...}, Pilot => {identifier: ..., ...}, Fly_Attendant => [...], Ticket => [...], Company => [...], ...}
      ↘ 0: {"User" => Object}
        key: 'User'
        ↘ value: {identifier: '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: {...}}
```

FUNCIONAMENTO DO OGM

Uma vez definido o esquema da aplicação para o banco de dados os mapeadores interagem com seus banco de dados adjacentes.

O mapeador desenvolvido realiza as seguintes operações:

- Aplicação de índices
- Inicializa o gerador de modelos do cliente

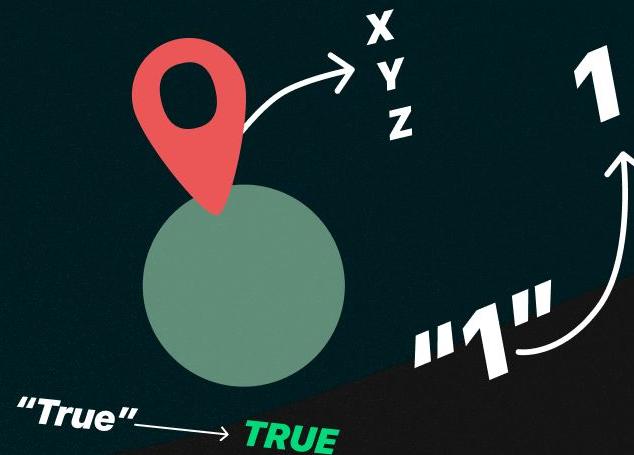
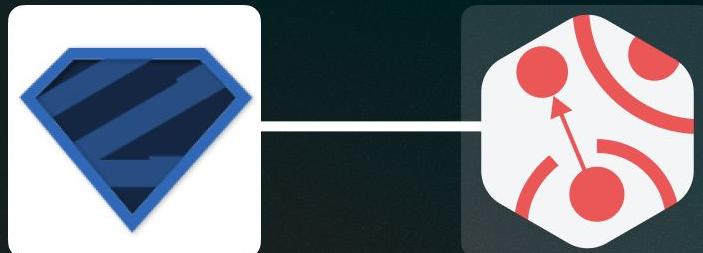




MÓDULO DE DEFINIÇÃO E ESTRUTURAÇÃO DO ESQUEMA

Com base no esquema de cada **modelo** da aplicação é **mapeado** e **instanciado** um **esquema de validação** usando o **Zod** (pacote com tipagem estática forte, usado para validação de dados).

Concomitante é criado um **formatador** dos dados para cada modelo, desta forma os números, pontos geográficos são retornados corretamente.





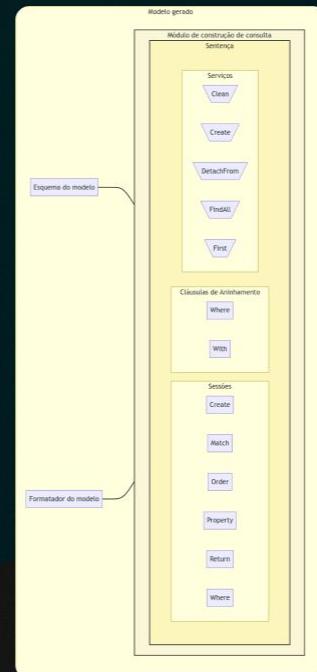
MÓDULO DE CONSTRUÇÃO DE CONSULTA

Dentro do **modelo do cliente** também possui uma instância do construtor de consulta, no qual o mesmo possui a seguinte estrutura:

- **Sentença**
 - Serviços
 - Abstração de consultas recorrentes
 - Cláusulas de aninhamento
 - Sessões

Com base nessa estrutura e usando como base o **Knex.js** o funcionamento do construtor de consulta é realizado usando conceitos de **programação dinâmica**.

```
example() {  
    this.#entityRepository.query().match('n', 'User').return('n').execute();  
}
```





MÓDULO DE CONSTRUÇÃO DE CONSULTA

Desta forma, a utilização do módulo como do mapeador ficou simples e íntegra.

The screenshot shows a code editor with a dark theme. On the left, the 'EXPLORADOR' (Explorer) sidebar displays a project structure under 'TCG (WORKSPACE)'. The 'src' folder contains 'core', 'models', 'modules', and 'app'. 'core' includes 'concerns' (with 'model' containing 'base.controller.ts', 'base.serializer.ts', 'base.service.ts', and 'index.ts') and 'config'. 'models' contains 'category', 'event', 'product', and 'user' (with 'user' containing 'controller.spec.ts', 'controller.ts', 'interface.ts', 'module.ts', 'serializer.ts', 'service.spec.ts', 'service.ts', and 'index.ts'). 'modules' contains 'spec', 'app.controller.ts', 'app.module.ts', 'app.service.ts', and 'main.ts'. 'app' contains 'configuration.ts'. 'scripts.npm' is also listed. The main editor area shows a portion of the 'base.service.ts' file:

```
19 export class BaseModelService<T extends Model<any, any>>
20   getAll(
21     ...
22     return this.#entityRepository.all(properties, options) as AllResponse<T>;
23   } catch (error) {
24     throw new BadGatewayException(objectOrError: error);
25   }
26 }
27
28 get(id: Parameters<T['find']>['0']) {
29   try {
30     return this.#entityRepository.find(identifier: id) as GetResponse<T>;
31   } catch (error) {
32     throw new BadGatewayException(objectOrError: error);
33   }
34 }
35
36 delete(id: Parameters<T['delete']>['0']) {
37   try {
38     return this.#entityRepository.delete(identifier: id) as DeleteResponse<T>;
39   } catch (error) {
40     throw new BadGatewayException(objectOrError: error);
41   }
42 }
43
44 You, há 2 meses. «feat(models)» create model concern and model module
```

The status bar at the bottom indicates: 'You, há 2 meses' 'Ln 63, Col 4' 'Espaços: 2' 'UTF-8' 'TypeScript' 'Spell'.



MÓDULO DE CONSTRUÇÃO DE CONSULTA

Observe os erros gerados a devido a tipagem estática usando o esquema do modelo.

The screenshot shows a dark-themed VS Code interface. On the left is the Explorer sidebar displaying a file tree for a project named 'TCC (WORKSPACE)'. The 'src' folder contains several sub-directories like 'core', 'models', 'config', 'database', 'modules', and 'user'. The 'user' directory is expanded, showing files such as 'user.controller.spec.ts', 'user.controller.ts', 'user.interface.ts', 'user.module.ts', 'user.serializer.ts', 'user.service.ts', and 'index.ts'. The main editor area shows a file named 'user.service.ts' with the following code:

```
1 import { Injectable } from '@nestjs/common'; 202.4K (gzipped: 42.4k)
2 import { BaseModelService } from 'core/concerns/model';
3 import { OGMService } from 'core/database/ogm-neo4j/ogm.service';
4
5 import type { UserModel } from './user.interface';
6
7 @Injectable()
8 export class UsersService extends BaseModelService<UserModel> {
9   constructor(private readonly ogmService: OGMService) {
10     super(service: ogmService, name: 'User');
11   }
12 }
```

A red squiggly underline is under the line 'super(service: ogmService, name: 'User');'. A tooltip for the error says: 'You, há 2 meses • feat(models): create model concern and model rou...'.

At the bottom of the editor, there are status bars for 'DBMS(neo4j)', 'File', 'Edit', 'Search', 'Terminal', 'Output', 'Tasks', 'Issues', 'Linter', 'Performance', 'Performance (Experimental)', 'Help', and 'About'. The bottom right corner shows the page number '26'.



CONJUNTO DE DADOS

O conjunto de dados utilizado “*eCommerce behavior data from multi-category store*” é um conjunto de dados disponível na plataforma Kaggle composto por **285 milhões de eventos** de usuários de uma plataforma de vendas multimarcas. Em que 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**.

Motivos:

- Maior quantidade de dados sem presença de arquivos estáticos;
- Integridade; e
- Possíveis análises.

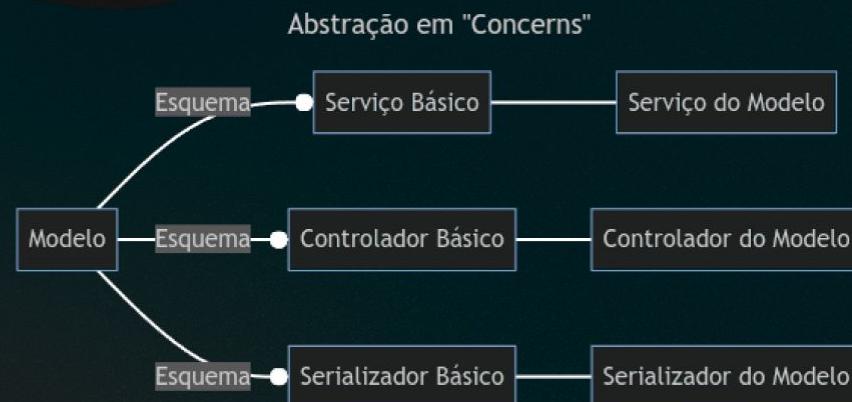
Propriedade	Descrição
event_time	Tempo quando o evento ocorreu no formato UTC.
event_type	Tipo do evento que ocorreu, sendo eles: visualizar, adicionar ou remover do carrinho e comprar.
product_id	Identificador único do produto na plataforma.
category_id	Categoria do produto.
category_code	Código da categoria do produto.
brand	Marca do produto.
price	Preço do produto.
user_id	Identificador único do usuário.
user_session	Identificador temporário da sessão do usuário na plataforma. É alterado toda vez que o usuário retorna a plataforma após um longo período.



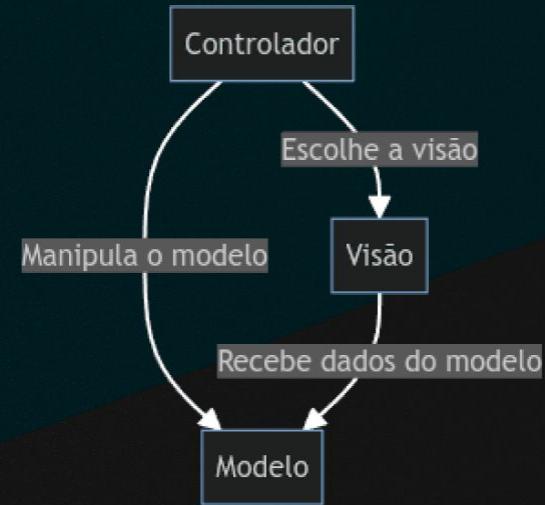
DESENVOLVIMENTO DA API

Usado o **NestJS** para construção das APIs, foi realizado uma **etapa de planejamento** com o objetivo de unificar e facilitar a criação das duas APIs, para que fossem equivalentes em complexidade para melhor avaliar o desempenho das aplicações.

Uso de *Concerns* para efetivar a equivalência entre as APIs



MVC (Model-View-Controller)





MÓDULO DE CONSTRUÇÃO DE CONSULTA

Conforme, que a construção da ferramenta foi utilizando a linguagem **TypeScript**, é **importante** o uso das **tipagens estáticas**. De modo comparativo, foi encontrado um **problema** no desenvolvimento da API utilizando o mapeador relacional se comparado a implementação do OGM desenvolvido.

```
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);
    }
  }
}
```

```
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;
  }

  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 });
  }
}
```



ESQUEMA DA APLICAÇÃO

```
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.Timestampz()
  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])
}
```

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

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

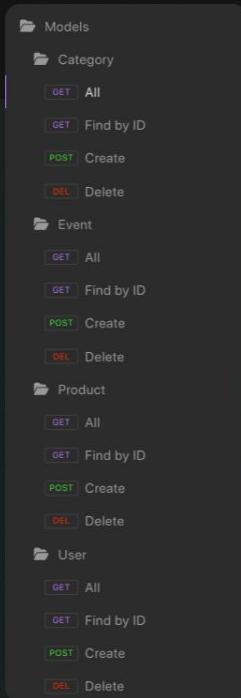
Node Category {
  id:           String       @identifier
  code:         String?
}

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



DESENVOLVIMENTO DA API

Com base nos modelos usados no esquema da aplicação foi criado as seguintes rotas para API:



- **Saúde da API**
 - **GET** /health
 - **GET** /database/health
- **Rotas dos modelos**
 - **GET** /<nome-do-modelo>
 - **GET** /<nome-do-modelo>/:id
 - **POST** /<nome-do-modelo>
 - **PUT** /<nome-do-modelo>/:id
 - **PATCH** /<nome-do-modelo>/:id



HOSPEDAGEM

Para hospedagem foi utilizado o servidor especificado anteriormente, usando a estrutura de contêineres. Sendo assim foi criado uma imagem do **Docker** e depois a criação de *docker-compose* para configuração do sistema da aplicação como um todo.



```
#-----
# Build stage
#-----
FROM node:20.1-alpine as builder

# -----
# Install system dependencies
# -----
RUN apk add --no-cache --virtual git

WORKDIR /usr/src/app

COPY package*.json .

# -----
# Install dependencies
# -----
RUN npm ci --only=production \
  && npm cache clean --force

COPY . .

RUN npm run build

#-----
# Run stage
#-----
FROM node:20.1-alpine as runner

WORKDIR /usr/src/app
ENV NODE_ENV=production

COPY --from=builder /usr/src/app .

EXPOSE 3000

CMD ["npm", "run", "start:prod"]
```



EXPERIMENTOS

EXPERIMENTOS SOBRE O MAPEADOR

Para mensurar o **tempo de resposta** e o **tempo de criação de consulta**.

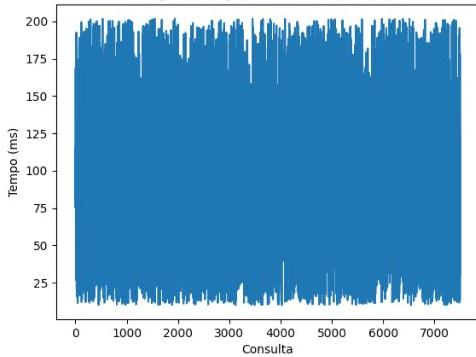
- **7.500 consultas**
- Ambiente Node.js

ATRIBUTO	DESCRIÇÃO
instantiation_time	Tempo de instanciação
query_generation_time	Tempo de geração da consulta
response_time	Tempo de resposta do banco de dados

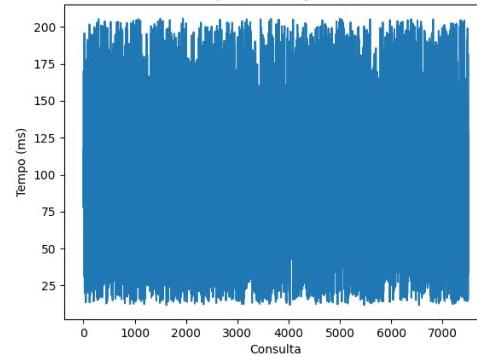




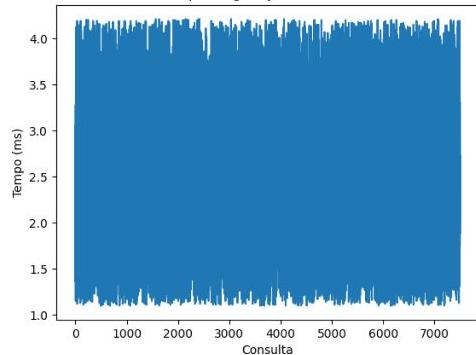
Tempo de resposta do banco de dados



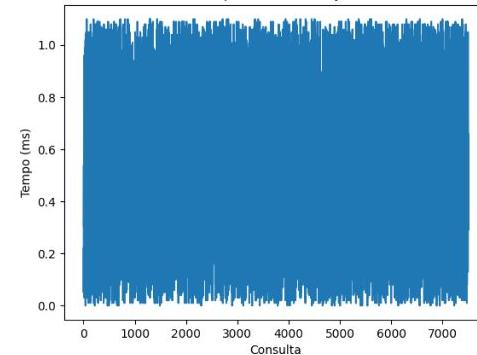
Tempo de execução total



Tempo de geração de consulta



Tempo de Instanciação





EXPERIMENTOS

EXPERIMENTOS SOBRE O MAPEADOR

Para mensurar o **tempo de resposta** e o **tempo de criação de consulta**.

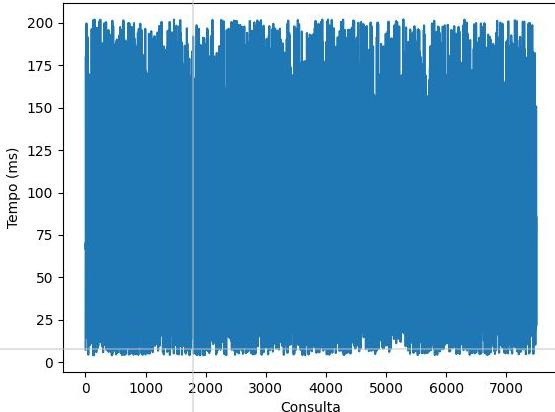
- **7.500 consultas**
- Ambiente Node.js

OBS: não foi possível mensurar o tempo de instanciação para não alterar o código fonte.

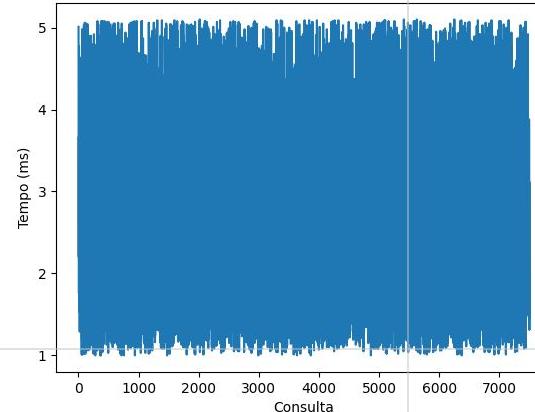


ATRIBUTO	DESCRIÇÃO
query_generation_time	Tempo de geração da consulta
response_time	Tempo de resposta do banco de dados

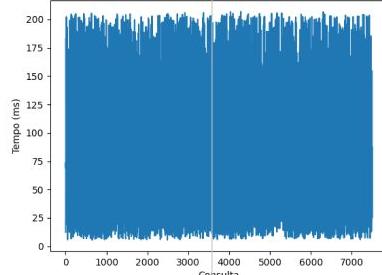
Tempo de resposta do banco de dados



Tempo de geração de consulta



Tempo de execução total





ANÁLISE

EXPERIMENTOS SOBRE O MAPEADOR

Para análise foi obtida as métricas do experimento dirigido anteriormente.

ORM (Prisma)

MÉTRICA	TEMPO (ms)
Média de geração de consulta	2,80
Média de tempo de resposta	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

OGM

MÉTRICA	TEMPO (ms)
Média de geração de consulta	2,51
Média de tempo de resposta	93,42
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



ANÁLISE

EXPERIMENTOS SOBRE O MAPEADOR

Para melhor entendimento, o gráfico de Gantt segmenta os tempos capturados para cada etapa.





EXPERIMENTOS

EXPERIMENTOS SOBRE A APLICAÇÕES

Com o objetivo de certificar que ambas as aplicações operam como sistemas reais, com baixa latência a um nível mínimo de erros.

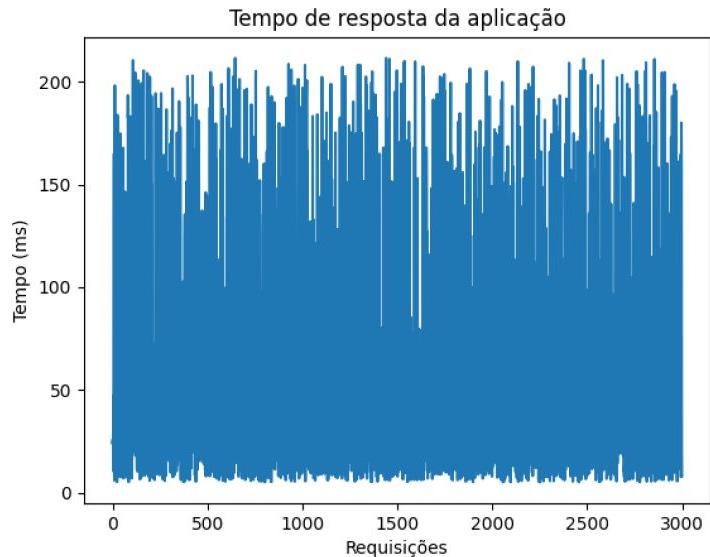
- 3.000 Requisições HTTP (**GET, POST e PUT** sobre o modelo “Produtos”)

OBS: A cada 15 requisições as próximas 15 eram paralelas.

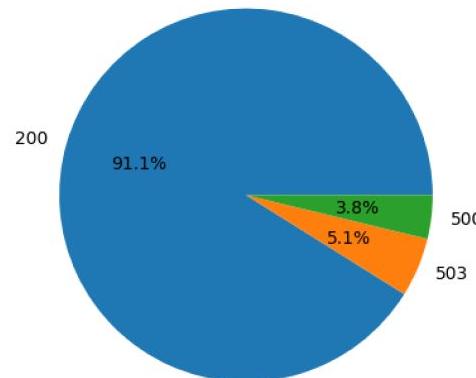
ATRIBUTO	DESCRIÇÃO
response_time	Tempo de resposta
response_status	Qual status HTTP retornou na resposta



OGM DESENVOLVIDO

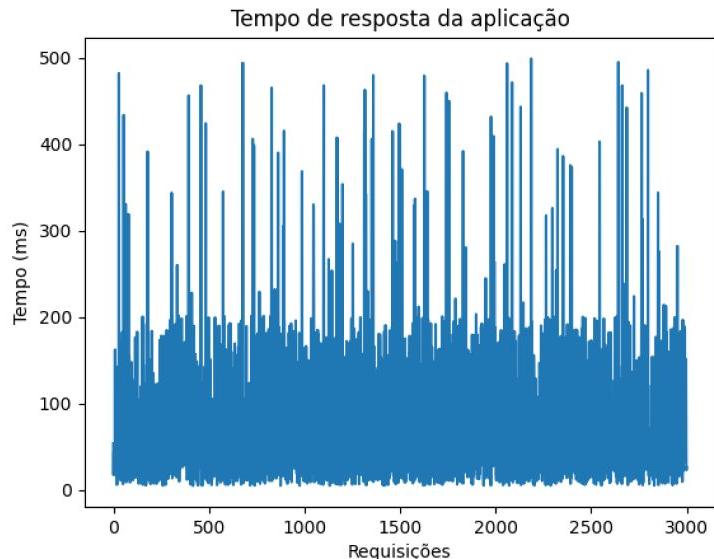


Códigos de status

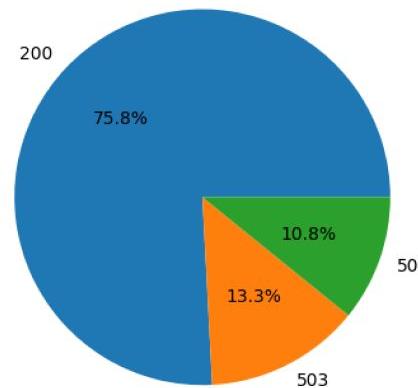




ORM (PRISMA)



Códigos de status





ANÁLISE

EXPERIMENTOS SOBRE AS APLICAÇÕES

Para análise foi obtida as métricas do experimento dirigido anteriormente.

ORM (Prisma)

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

OGM

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



CONCLUSÃO

Portanto o mapeador **comprovou** sua **eficácia e praticidade**, apresentando **tempos menores e consistentes**.



Mapeador Objeto-Grafo

ogm-neo4j Public

Actions Projects Wiki Security Insights Settings

ogm-neo4j Public

2 branches 0 tags Go to file Add file Code

Contribute

feat(lexer/parser): add initial implementation of lexer and parser 5 days ago 7 commits

feat(lexer/parser): add initial implementation of lexer and parser 5 days ago

feat(lexer/parser): add initial implementation of lexer and parser 5 days ago

init: base project structure 2 months ago

fix(database-integration): fix connection through Neo4j-Driver 5 days ago

init: base project structure 2 months ago

Initial commit 2 months ago

fix(database-integration): fix connection through Neo4j-Driver 5 days ago

fix(database-integration): fix connection through Neo4j-Driver 5 days ago

fix(database-integration): fix connection through Neo4j-Driver 5 days ago

About

Object-Graph-Mapping for easy and flexible node and operations

MIT license

Activity

0 stars

1 watching

0 forks

Releases

No releases published Create a new release

Packages

No packages published Publish your first package

Languages

TypeScript 100% 43

repository understand your project by adding a README.

Add a README

CONSIDERAÇÕES FINAIS

Para aprimoramento do projeto seria importante considerar o desenvolvimento dos seguintes tópicos:

- Adição de tipos;
- Implementação do conceito de migrações;
- Aprimorar a geração de módulo de clientes;
- Aprimorar o suporte a paginação; e
- Importar formatadores dos modelos em memória.

Obrigado!

Agradeço a presença



Cassiano Rodrigues

Estudante de Ciências da Computação
Desenvolvedor Frontend

