Comparação de Desempenho entre os ORMs TypeORM, Prisma e Sequelize em Aplicações Node.js

Eduardo Aparecido de Oliveira
Vinicius Aparecido De Souza
Vinicius Cardoso Jungers
Fabio Codo

Resumo

Este artigo apresenta uma comparação entre o desempenho dos ORMs (*Object-Relatioal Mappers*) TypeORM, Prisma e Sequelize no ambiente de execução Node.js. Para esta comparação, foram realizados testes para cada operação CRUD (*Create, Read, Update, Delete*), usando a ferramenta de testes K6, um aplicativo desenvolvido pela Grafana Labs, juntamente com um projeto para gerar e executar os testes. A comparação foi realizada com base nas requisições por segundo, tempo de resposta e consistência de requisições de cada um dos ORMs utilizando o banco de dados PostgreSQL. O objetivo deste artigo é auxiliar na escolha do ORM mais adequado para diferentes cenários de uso.

Palavras-chave*:* ORM; Desempenho; Comparação; Node.js.

Performance Comparison between TypeORM, Prisma and Sequelize ORMs in Node.js Applications

Abstract

The present paper discusses a comparison of performance between the TypeORM, Prisma and Sequelize object-relational mappers, on Node.js execution environment. For the purpose of comparison, tests were executed for each of the registers creation, reading, updating and deleting operations. Thus, the k6 testing tool, created by Grafana Labs, has been employed along with a project developed for generating and executing tests. The comparison has been carried out based on the parameter of requisitions per second, response time as well requisitions consistency of each object-relational mapper through a PostgreSQL database. Therefore, its article aim is to assist with the choice of the most suitable object-relational mapper for distinct use scenarios.

Keywords*: Object-relational mapping; Performance; Comparison; Node.js.*

---

1 INTRODUÇÃO

O Mapeamento Objeto-Relacional (ORM) é um padrão para acessar bancos de dados relacionais em sistemas orientados a objetos, permitindo que desenvolvedores utilizem diretamente conceitos de programação orientada a objetos sem se preocupar com conversões para utilização de bancos de dados relacionais. As principais funcionalidades dos ORMs envolvem estabelecer conexões com bancos de dados e realizar operações CRUD (Criação, Leitura, Atualização e Exclusão). Segundo Zmaranda (2020), a principal vantagem de utilizar um ORM é o fato de que o desenvolvimento é focado no nível de domínio (modelagem), determinando um nível maior de abstração em como a aplicação vai armazenar e recuperar dados, resultando em uma maior facilidade de desenvolvimento e manutenção de código.

Existem diversas ferramentas de ORM, cada uma com suas particularidades e funcionalidades únicas, que estão presentes em diversas linguagens de programação. Um

ponto negativo na implementação destas ferramentas é o impacto no desempenho geral, uma vez que a adição de uma camada de abstração resulta em um código mais complexo e, portanto, menos eficiente. De acordo com Güvercin (2020), como as ferramentas ORM diminuem o desempenho geral de aplicações onde são implementadas, a importância do desempenho de um ORM é fundamental.

Este artigo visa comparar o desempenho entre os ORMs TypeORM, Prisma e Sequelize do Node.js, que são os mais baixados no site npm trends (https://npmtrends.com/), que compara o número de downloads de pacotes do tipo NPM (*Node Package Manager*). Os critérios de avaliação utilizados foram o tempo de resposta da execução de operações CRUD, assim como o número de requisições por segundo e a consistência dessas requisições. Os testes foram feitos com base em um projeto com diversas entidades e relações, realizando operações com quantidades de dados variadas para determinar o ORM mais performático em diferentes cenários com a utilização do banco de dados PostgreSQL.

2 DESENVOLVIMENTO

2.1 Materiais e Métodos

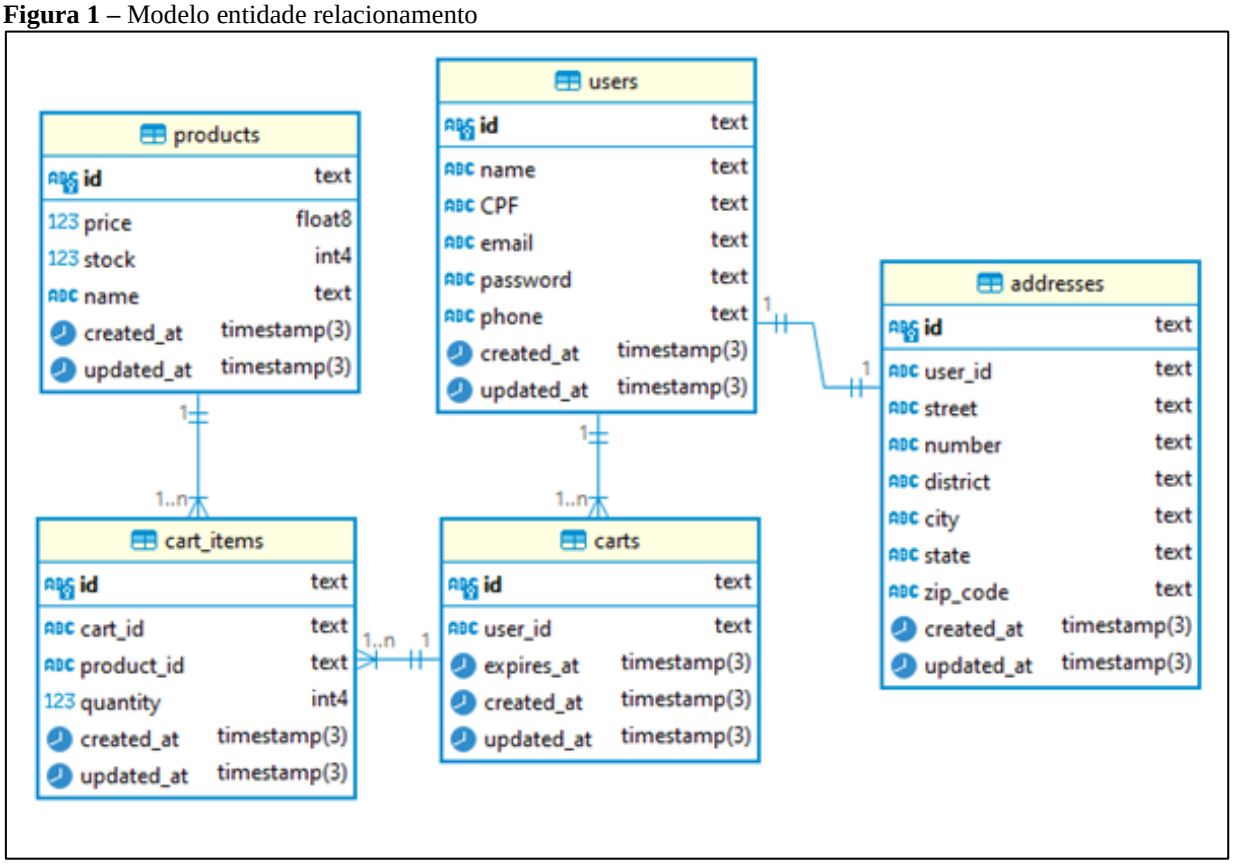
Os testes foram efetuados em uma máquina com as seguintes configurações: processador Ryzen 5600 com frequência base de 3.5GHz, com 12 núcleos lógicos e 6 núcleos físicos, 16GB de memória RAM DDR4 operando a 3200MHz, armazenamento secundário com HD 1TB 5400RPM, placa de vídeo RTX 2060 6GB e utilizando o sistema operacional Microsoft Windows 10 Home versão 22H2 de 64-bits.

Para a realização dos testes foi necessário a instalação dos seguintes programas e arquivos: Node.js v18.12.1, PostgreSQL v15.2 64-bit, K6 v0.50.0.

Foram utilizados para a execução dos projetos os seguintes pacotes NPM: Prisma v5.11.0, TypeORM v0.3.20, Sequelize v6.37.1, UUID v9.0.1, Express v4.18.2 e Babel v7.12.6.

Com os programas e os arquivos instalados, foi primeiramente criada a modelagem do banco de dados, que será empregado nas comparações e é composta pelas entidades *Users*, *Adresses*, *Products*, *Cart Itens* e *Carts*, como exibido na Figura 1.

Figura 1 – Modelo entidade relacionamento



Fonte: dos autores.

Também foi elaborado um projeto para conduzir os testes e processar os resultados, utilizando a ferramenta de testes k6 para a realização de requisições HTTP (*HyperText Transfer Protocol*). Visando realizar essas requisições, foi necessário gerar os dados de forma prévia, utilizando a biblioteca *Faker* (versão 8.0.2), que permite a criação de dados realísticos, e os armazenando localmente como arquivos no formato JSON (*JavaScript Object Notation*).

Após a geração e armazenamento dos dados, foi criado um arquivo que será executado pelo k6, com o comando "k6 run", em conjunto com o nome do arquivo e variáveis de ambiente. Neste arquivo, é especificado a origem dos dados utilizados, o formato da requisição, o número de iterações, o *endpoint* utilizado, os usuários virtuais e o endereço da aplicação, definidos a partir das variáveis de ambiente, conforme mostrado na Figura 2.

Figura 2 – Arquivo de execução k6

```
import { scenario } from "k6/execution";
import http from "k6/http";

let body_datas = __ENV.DATA_FILE
  ? JSON.parse(open(String(__ENV.DATA_FILE)))
  : [];

let path_vars = __ENV.PATH_VARS_FILE
  ? JSON.parse(open(String(__ENV.PATH_VARS_FILE)))
  : [];

export const options = {
  vus: __ENV.VUS,
  iterations: __ENV.ITERATIONS,
};

const selectedHost = __ENV.HOST;
const url = `${selectedHost}/${__ENV.ENDPOINT}`;

export default function () {
  const path_var = path_vars[scenario.iterationInTest % path_vars.length];

  let body =
    JSON.parse(body_datas[scenario.iterationInTest % body_datas.length]) || {};
  body.id = path_var;

  http[String(__ENV.HTTP_METHOD).toLowerCase()](url, JSON.stringify(body), {
    headers: {
      "Content-Type": "application/json",
    },
  });
}
```

Fonte: dos autores.

Por último, é enviado no comando de execução a *flag* "--out cloud", cujo objetivo é gerar gráficos com os resultados obtidos dos testes. A criação do arquivo de execução e os gráficos foram elaborados conforme a documentação da ferramenta de testes k6 (Grafana, 2024), sendo os testes executados localmente e os resultados processados na plataforma online da ferramenta.

Após a criação da aplicação responsável pelos testes, foram elaborados projetos para cada um dos ORMs, de acordo com a arquitetura MVC (*Model View Control*), onde a *Model* gerencia e representa entidades e relações do banco de dados dentro do projeto, o *Controller* recebe e gerencia as requisições HTTP recebidas, e a *View* responsável por exibir os resultados em formato JSON.

É importante ressaltar que cada ORM possuí diferentes formas de implementação, sendo cada uma delas baseada na sua respectiva documentação (Prisma, 2024; TypeORM, 2024; Sequelize, 2024). A fim de resolver as diferenças entre as implementações, a *Model* foi

dividida em duas camadas, a *Model* em si (representa as entidades e relacionamentos) e o *repository* (executa as operações CRUD de acordo com a implementação da *Model*).

Por fim, foi desenvolvida uma interface para padronização das operações CRUD que serão realizadas pelo *repository*, que define os métodos *create* e *update*, recebem e retornam uma *Model*, uma *read* que retorna uma lista de *Models*, e o *delete*, que recebe apenas o *id* da entidade e não retorna nenhum valor.

2.2 Resultados e discussões

Para realizar as comparações, os testes foram separados por cada operação dos métodos CRUD, sendo elas: Operações de criação, operações de leitura, operações de atualização e operações de exclusão.

2.2.1 Operações de criação

Nas operações de criação, foram inseridos usuários conforme os valores da entidade *Users*, apresentados na Figura 1. Nesta primeira operação, os usuários foram inseridos sem nenhuma relação adicional, sendo as requisições por segundo e o tempo de resposta os primeiros parâmetros a serem analisados, com base nos valores apresentados na Tabela 1.

Tabela 1 – Requisições por segundo (*Create*).

Name	Max	Min	Mean
TypeORM	1.26K req/s	435 req/s	1.22K req/s
Sequelize	1.09K req/s	335 req/s	1.05K req/s
Prisma	1.02K req/s	231 req/s	985 req/s

Fonte: dos autores.

É possível constatar a partir do Quadro 1 que o TypeORM obteve os melhores resultados, com uma média de tempo de resposta por requisição de 0.819 ms, seguido pelo Sequelize com 0.952 ms e pelo Prisma com 1.015 ms.

Os três ORMs geraram *queries* semelhantes, mas apenas a *query* do TypeORM retorna somente o id, enquanto as outras *queries* retornam todos os campos inseridos. As *queries* geradas são compostas por um *INSERT* e um *RETURNING* para a devolução dos dados inseridos, conforme mostra a Figura 3.

Figura 3 – Queries geradas (*Create*)

Prisma:

```
INSERT INTO "public"."users" ("id","name","CPF","email","password","phone","created_at","updated_at")
VALUES ($1,$2,$3,$4,$5,$6,$7,$8)
RETURNING "public"."users"."id", "public"."users"."name", "public"."users"."CPF", "public"."users"."email",
"public"."users"."password", "public"."users"."phone", "public"."users"."created_at",
"public"."users"."updated_at"
```

TypeORM:

```
INSERT INTO "users"("id", "name", "email", "password", "CPF", "phone", "created_at", "updated_at")
VALUES ($1, $2, $3, $4, $5, $6, $7, $8)
RETURNING "id"
```

Sequelize:

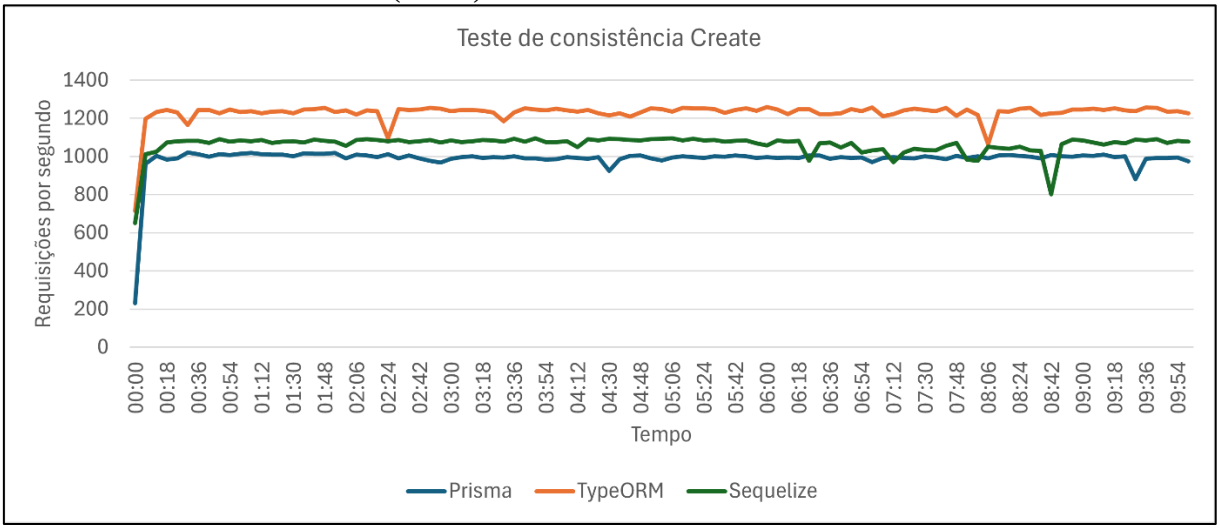
```
INSERT INTO "users" ("id","name","CPF","email","password","phone","created_at","updated_at")
VALUES ($1,$2,$3,$4,$5,$6,$7,$8)
RETURNING "id","name","CPF","email","password","phone","created_at","updated_at";
```

Fonte: dos autores.

As *queries* geradas podem ser um dos fatores para justificar a diferença de desempenho entre os ORMs, mas sendo a implementação e otimização de cada ORM a principal responsável por essa diferença. O Prisma apresenta o pior desempenho, uma vez que, além de possuir um nível de abstração elevado em comparação com os outros ORMs, também utiliza o Prisma Engine, uma camada adicional entre o ORM e o banco de dados, responsável por gerar as *queries* e manter a compatibilidade com diferentes bancos de dados, enquanto os outros dois ORMs têm abordagens mais diretas para a geração de código SQL (*Structured Query Language*).

Por fim, a consistência das requisições foi verificada e o resultado é apresentado no Gráfico 1, composto de um teste de dez minutos para cada um dos ORMs, sendo este o tempo máximo de teste contínuo da ferramenta k6, onde nestes 10 minutos, o ORM respondeu ao máximo de requisições possíveis sequencialmente, com o resultado exibido no Gráfico 1.

Gráfico 1 – Gráfico de consistência (*Create*)



Fonte: dos autores.

Observa-se que todos os ORMs apresentaram uma boa consistência, com algumas oscilações de desempenho, sendo o Sequelize o mais inconsistente e o Prisma o mais consistente. Isso pode indicar que, apesar do desempenho inferior aos outros ORMs, o Prisma pode ser mais adequado em situações em que a consistência é mais importante do que desempenho bruto.

2.2.2 Operações de leitura

É importante salientar que, para as operações de leitura, foi utilizada apenas a tabela *Users*, sem nenhuma relação adicional e com a leitura de um usuário por vez, com as requisições por segundo sendo exibidas no Tabela 2.

Tabela 2 – Requisições por segundo (*Read*).

Name	Max	Min	Mean
Prisma	1.25K req/s	183 req/s	1.23K req/s
TypeORM	1.65K req/s	92.3 req/s	1.61K req/s
Sequelize	1.62K req/s	308 req/s	1.57K req/s

Fonte: Os autores.

O Quadro 2 mostra que o TypeOrm obteve o melhor desempenho médio, com um tempo de resposta médio por requisição de 0.621 ms, seguido pelo Sequelize, com 0.636 ms e Prisma, com 0.813. É possível notar que o TypeORM e Sequelize apresentaram resultados muito semelhantes, enquanto o Prisma apresentou resultados bem inferiores.
Ao analisarmos as *queries* geradas por cada um dos ORMs, percebemos que todas elas são semelhantes, o que indica que a *query* gerada não teve um impacto significativo no desempenho de cada um dos ORMs. As *queries* são compostas por *SELECT*, *FROM* e *WHERE*, conforme mostra a Figura 4.

Figura 4 – Queries geradas (*Read*)

Prisma:

```
SELECT "public"."users"."id", "public"."users"."name", "public"."users"."CPF", "public"."users"."email",
"public"."users"."password", "public"."users"."phone", "public"."users"."created_at",
"public"."users"."updated_at"
FROM "public"."users"
WHERE ("public"."users"."id" = $1 AND 1=1) LIMIT $2 OFFSET $3
```

TypeORM:

```
SELECT "User"."id" AS "User_id", "User"."name" AS "User_name", "User"."email" AS "User_email",
"User"."password" AS "User_password", "User"."CPF" AS "User_CPF", "User"."phone" AS "User_phone",
"User"."created_at" AS "User_created_at", "User"."updated_at" AS "User_updated_at"
FROM "users" "User"
WHERE (("User"."id" = $1)) LIMIT 1
```

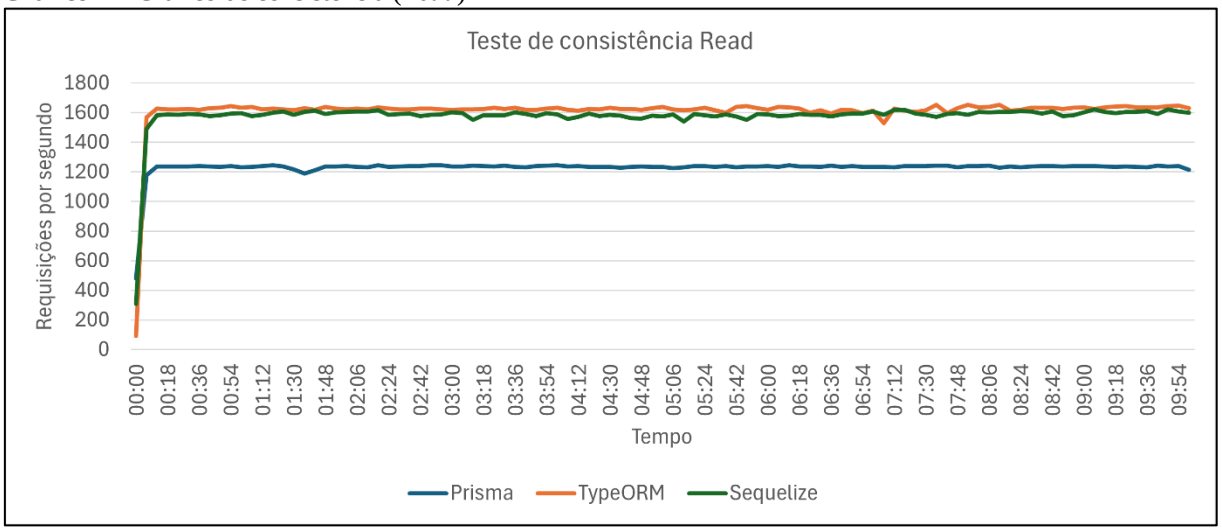
Sequelize:

```
SELECT "id", "name", "CPF", "email", "password", "phone", "created_at" AS "createdAt", "updated_at" AS
"updatedAt"
FROM "users" AS "users"
WHERE "users"."id" = '0000101c-00ab-45bc-991b-e04cf5f82d22';
```

Fonte: dos autores.

É importante salientar que o Sequelize usa o método “findByPk”, com a *query* resultante deste método não utilizando *placeholders*. Apesar disso, ele ainda possui métodos para evitar possíveis problemas de segurança.
Por último, foi verificada a consistência das requisições, exibida no Gráfico 2.

Gráfico 2 – Gráfico de consistência (*Read*)



Fonte: dos autores.

É possível notar que todos os ORMs apresentaram um desempenho estável, uma vez que a leitura é uma das operações de menor complexidade. No entanto, o Prisma apresentou um desempenho novamente inferior ao dos outros ORMs devido aos mesmos fatores apresentados no *Create*.

2.2.3 Operações de atualização

Nas operações de atualização, todos os dados da tabela *Users* foram atualizados, com exceção do id. As operações foram realizadas sem alterar nenhuma outra relação e apenas com um usuário por requisição, com as requisições por segundo sendo exibidas no Tabela 3.

Tabela 3 – Requisições por segundo (*Update*).

Name	Max	Min	Mean
TypeORM	994 req/s	121 req/s	885 req/s
Prisma	806 req/s	35.2 req/s	706 req/s
Sequelize	907 req/s	162 req/s	807 req/s

Fonte: Os autores.

O Quadro 3 mostra que o TypeORM obteve os melhores resultados gerais novamente, com um tempo médio de resposta por requisição de 1.129 ms, seguido pelo Sequelize com 1.239 ms e o Prisma com 1.416 ms.

Ao verificarmos as *queries* geradas por cada ORM, podemos notar algumas diferenças que potencialmente impactam o desempenho, com o Prisma compondo sua *query* com *UPDATE*, *SET*, *WHERE* e *RETURNING*, sendo o único ORM que retorna os dados depois de atualizados. O TypeORM faz uma *query* similar, utilizando *UPDATE*, *SET*, *WHERE* e *IN*,

sem retornar nenhum dado, enquanto o Sequelize gera a *query* mais simples, sendo composta apenas de um *UPDATE*, *SET* e *WHERE*, como exibido na Figura 5.

Figura 5 – Queries geradas (*Update*)

Prisma:

```
UPDATE "public"."users"
SET "name" = $1, "CPF" = $2, "email" = $3, "password" = $4, "phone" = $5, "created_at" = $6, "updated_at" = $7
WHERE ("public"."users"."id" = $8 AND 1=1)
RETURNING "public"."users"."id", "public"."users"."name", "public"."users"."CPF", "public"."users"."email",
"public"."users"."password", "public"."users"."phone", "public"."users"."created_at",
"public"."users"."updated_at"
```

TypeORM:

```
UPDATE "users"
SET "id" = $1, "name" = $2, "CPF" = $3, "email" = $4, "password" = $5, "phone" = $6, "created_at" = $7,
"updated_at" = $8
WHERE "id"
IN ($9)
```

Sequelize:

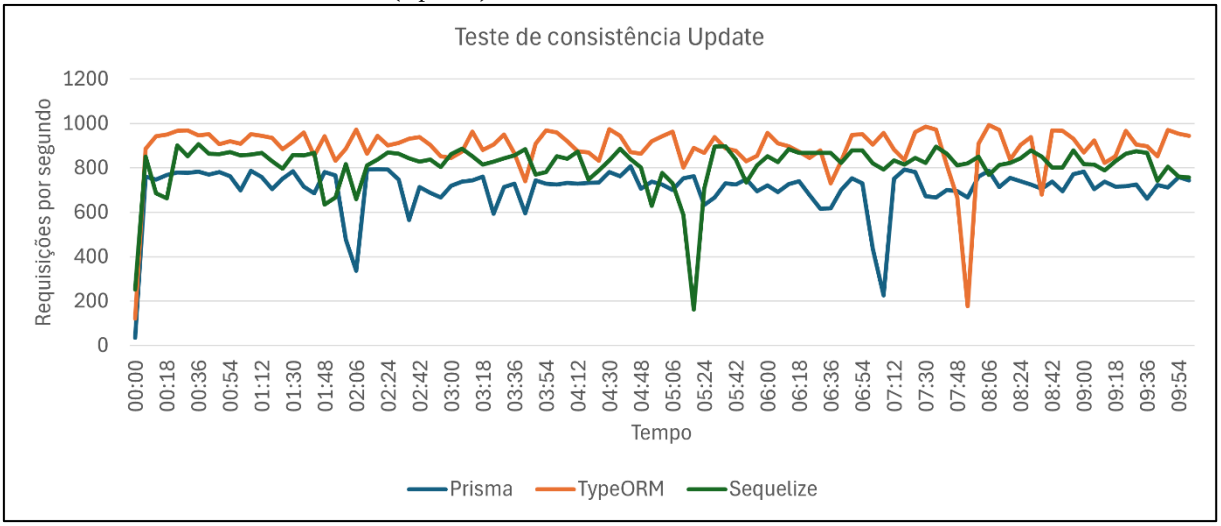
```
UPDATE "users"
SET "id"=$1,"name"=$2,"CPF"=$3,"email"=$4,"password"=$5,"phone"=$6,"updated_at"=$7
WHERE "id" = $8
```

Fonte: dos autores.

É importante ressaltar que a diferença de desempenho entre cada uma das *queries* pode ser insignificante em operações de *update* mais simples, ou quando utilizadas com um conjunto menor de dados, sendo a otimização de cada ORM um fator muito mais relevante.

Ao observar o gráfico de consistência das requisições realizadas, é possível notar uma grande inconsistência, com nenhum dos ORMs apresentando desempenho estável, conforme mostra o Gráfico 3.

Gráfico 3 – Gráfico de consistência (*Update*)



Fonte: dos autores.

A falta de consistência pode ser atribuída a diversos fatores, sendo o primeiro deles a complexidade deste tipo de operação, que é a mais exigente. Além disso, fatores como o *Database Locking*, atualização da tabela de indexação, e a abstração de cada um dos ORMs podem afetar o desempenho desta operação. É possível notar a deterioração do desempenho após um período limitado, com o início das operações sendo relativamente estável, indicando que a falta de consistência é consequência do esgotamento de recursos. Isso indica que cada

ORM tem capacidade máxima de trabalhar em seu máximo por um período limitado e, após esse período, o desempenho é degradado.

2.3 Operações de exclusão

Por fim, foram realizadas as operações de exclusão, sendo utilizado a tabela *Users*, com todos os dados a serem excluídos sendo inseridos previamente, e com a operação afetando apenas a tabela *Users*, com as operações de requisição por segundo sendo exibidas no Tabela 4.

Tabela 4 – Requisições por segundo (*Delete*).

Name	Max	Min	Mean
TypeORM	1.12K req/s	42 req/s	1.01K req/s
Prisma	916 req/s	163 req/s	818 req/s
Sequelize	1.15K req/s	280 req/s	999 req/s

Fonte: Os autores.

No último teste a ser executado, o TypeORM obteve a melhor média, com todos os ORMs obtendo resultados próximos e com o Sequelize obtendo as melhores mínimas e máximas. A média de tempo de resposta por requisição foi de 0.990 ms para o TypeORM, 1.001 ms para o Sequelize e 1.222 ms para o Prisma.

Analisando as *queries* geradas, podemos verificar que o Prisma novamente retorna os dados, com sua *query* consistindo em um *DELETE FROM*, *WHERE* e *RETURNING*, enquanto as queries do TypeORM e Sequelize consistem apenas de um *DELETE FROM* e *WHERE*, como mostra a Figura 6.

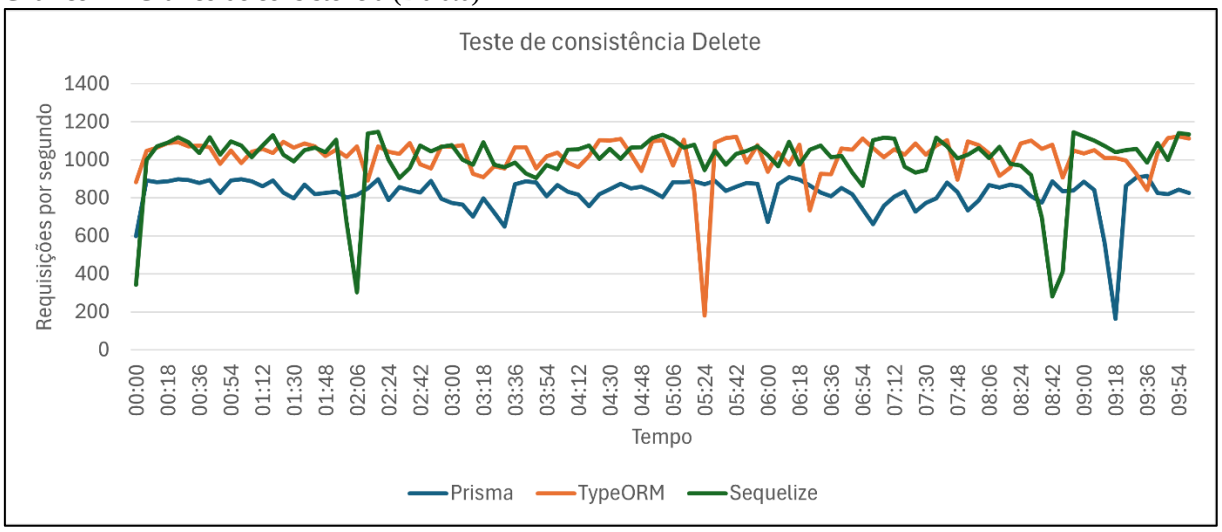
Figura 6 – Queries geradas (*Delete*)

```
Prisma:

DELETE FROM "public"."users"
WHERE ("public"."users"."id" = $1 AND 1=1)
RETURNING "public"."users"."id", "public"."users"."name", "public"."users"."CPF", "public"."users"."email",
"public"."users"."password", "public"."users"."phone", "public"."users"."created_at",
"public"."users"."updated_at"

TypeORM:

DELETE FROM "users"
WHERE "id" = $1

Sequelize:

DELETE FROM "users"
WHERE "id" = '00002287-aea4-4221-a39c-6bfd83e0bf7b'
```

Fonte: dos autores.

Em termos de consistência, podemos notar que, assim como no *update*, as operações de delete apresentam uma grande inconsistência em todos os ORMs, sendo o Prisma o mais lento entre todos, embora seja o mais estável. As oscilações de desempenho ocorrem após um determinado período, com o início das operações permanecendo constante, o que sugere que o *delete* é afetado pelos mesmos problemas de falta de recursos do *update* quando opera de forma contínua e por um longo período, como mostra o Gráfico 4.

Gráfico 4 – Gráfico de consistência (*Delete*)



Fonte: dos autores.

3 CONSIDERAÇÕES FINAIS

Este estudo comparou o desempenho dos ORMs TypeORM, Prisma e Sequelize, utilizando um conjunto de requisições por segundo, o tempo de resposta e a consistência das operações CRUD como critérios.

A análise dos resultados revelou que o TypeORM apresentou os melhores resultados em relação ao número de requisições por segundo, seguido pelo Sequelize, que, em muitos casos, esteve muito próximo do TypeORM, e, por fim, o Prisma, que apresentou resultados inferiores em termos de desempenho.

Ao analisar a consistência das requisições, o Prisma apresentou uma consistência ligeiramente superior em relação aos outros ORMs, tendo menos inconsistências nas operações de leitura. TypeORM e Sequelize apresentaram uma consistência bastante semelhante, mas o Sequelize apresentou mais queda de desempenho em relação ao TypeORM.

Apesar dos resultados obtidos, é importante ressaltar que o desempenho de um ORM pode variar significativamente conforme o cenário onde ele é implementado, sendo necessários testes para cada situação específica, além disso, cada ORM apresenta um conjunto diferente de recursos, sintaxes e uso de recursos do sistema, que devem ser considerados na escolha do ORM mais adequado para cada situação.

REFERÊNCIAS

GRAFANA. **Cloud k6 docs**. Disponível em: https://grafana.com/docs/k6/latest/results-output/real-time/cloud/. Acesso em: 21 abr. 2024.

GRAFANA. **Running k6 docs**. Disponível em: https://grafana.com/docs/k6/latest/get-started/running-k6/. Acesso em: 21 abr. 2024.

GÜVERCİN, Abdullah Eren; AVENOGLU, Bilgin**. Performance Analysis of Object-Relational Mapping (ORM) Tools in. Net 6 Environment**. Bilişim Teknolojileri Dergisi, v. 15, n. 4, p. 453-465, 2022. Disponível em: https://dergipark.org.tr/en/download/article-file/2198861. Acesso em: 07 fev. 2024.

PRISMA. **Prisma docs.** Disponível em: https://www.prisma.io/docs. Acesso em: 13 jan. 2024.

SEQUELIZE. **Sequelize docs.** Disponível em: https://sequelize.org/docs/v6/. Acesso em: 20 jan. 2024.

TYPEORM. **TypeORM docs.** Disponível em: https://typeorm.io/. Acesso em: 14 jan. 2024.

ZMARANDA, Doina et al. Performance comparison of CRUD methods using .NET object relational mappers: A case study. **International Journal of Advanced Computer Science and Applications**, v. 11, n. 1, 2020. Disponível em: https://thesai.org/Downloads/Volume11No1/Paper_7-Performance_Comparison_of_CRUD_Methods.pdf. Acesso em: 10 mar. 2024.