Comparando soluções para o problema de N+1 queries com APIs GraphQL em Ruby on Rails

Igor Simões
ibps@cin.ufpe.br
Centro de Informática da Universidade Federal de Pernambuco
Recife, Pernambuco, Brasil

RESUMO

Atualmente, APIs (*Application Programming Interface*) enfrentam desafios comuns relacionados ao problema de *N+1 queries*, como as APIs GraphQL escritas em Ruby on Rails (Rails). Este problema pode impactar significativamente a eficiência e a performance das aplicações. No entanto, existem diversas soluções que têm sido exploradas para contornar essa questão. Dentro do ecossistema Rails, soluções nativas são frequentemente adotadas.

O Active Record, um ORM (*Object-Relational Mapping*) utilizado pelo Rails, possui um método chamado *includes*. Esse método pré-carrega associações entre objetos, o que minimiza a quantidade de *queries* ao banco de dados. Além disso, algumas *gems* têm ganhado destaque no combate ao problema de *N+1 queries* em GraphQL. *Gems*, no contexto do Ruby, são pacotes de código que fornecem funcionalidades específicas, funcionando como bibliotecas que podem ser facilmente adicionados a projetos Ruby. Entre elas, podemos destacar a graphql-batch e a batch-loader. Ambas oferecem ferramentas poderosas para agrupar e resolver queries de maneira mais eficiente, eliminando a necessidade de múltiplas consultas individuais ao banco de dados. Ao desenvolver ou otimizar APIs GraphQL em Rails, é essencial considerar estas soluções e avaliar qual se encaixa melhor no contexto da aplicação, tendo em vista os prós e contras de cada abordagem e a necessidade de performance do sistema. Para aprofundar nossa compreensão, as soluções foram implementadas e testadas considerando métricas quantitativas, como o tempo total de execução, e qualitativas, como flexibilidade e usabilidade.

Em conclusão, enquanto cada solução pode ter suas vantagens e desvantagens, a escolha ideal para abordar o problema de *N+1 queries* em APIs GraphQL em Rails deve levar em consideração as especificidades e necessidades de cada projeto. Neste estudo, esperamos fornecer uma base sólida para tomada de decisão para desenvolvedores e equipes que enfrentam desafios semelhantes.

ABSTRACT

Currently, APIs (Application Programming Interface) face common challenges related to the *N+1 queries* problem, like GraphQL APIs written in Ruby on Rails (Rails). This issue can significantly impact the efficiency and performance of applications. However, several solutions have been explored to address this matter. Within the Rails ecosystem, native solutions are often adopted.

The Active Record, an ORM (*Object-Relational Mapping*) used by Rails, has a method called *includes*. This method pre-loads associations between objects, which minimizes the number of *queries* to the database. Moreover, some *gems* have gained prominence in tackling the *N+1 queries* problem in GraphQL. *Gems*, in the context of Ruby, are code packages that provide specific functionalities, acting as libraries that can be easily added to Ruby projects. Notably among them are graphql-batch and batch-loader. Both provide powerful tools for batching and resolving queries more efficiently, eliminating the need for multiple individual database queries. When developing or optimizing GraphQL APIs in Rails, it's crucial to consider these solutions and evaluate which fits best within the application's context, considering the pros and cons of each approach and the system's performance requirements. To deepen our understanding, the solutions were implemented and tested, considering quantitative metrics, such as the total execution time in milliseconds, and qualitative ones, like flexibility and usability.

In conclusion, while each solution may have its pros and cons, the ideal choice for addressing the *N+1 queries* problem in GraphQL APIs in Rails should take into account the specificities and needs of each project. With this study, we hope to provide a solid foundation for decision-making for developers and teams facing similar challenges.

KEYWORDS

GraphQL, Ruby on Rails, APIs, N+1, Queries

1 INTRODUÇÃO

Serviços web compreendem um conjunto de protocolos e padrões adotados para a transferência de informações entre sistemas online. Software, desenvolvidos em múltiplas linguagens de programação e operando em diferentes plataformas, recorrem a esses serviços para compartilhar dados por meio de redes, como a Internet. Tais serviços garantem a compatibilidade na comunicação entre diferentes sistemas[1].

Proposta publicamente em 2015 pelo Facebook, GraphQL foi concebido como uma linguagem de consulta para facilitar a construção de arquiteturas de serviços web, oferecendo sintaxe e sistema flexíveis e intuitivos para detalhar suas demandas de dados[5]. O GraphQL emergiu como uma alternativa promissora para resolver problemas enfrentados por outros estilos arquiteturais populares para a criação de APIs web, como o REST (*Representational Style Transfer*)[2–4]. Como resultado, a linguagem começou a ganhar força e é atualmente suportada por APIs web de diversas empresas e seus respectivos produtos, como Github, Airbnb, Netflix e Twitter[1].

Dentre os problemas que GraphQL se propõe a resolver, nós temos o problema de *under-fetching*. Em geral, o *under-fetching* ocorre quando uma API não fornece todos os dados necessários em uma única solicitação, exigindo requisições adicionais para obter as informações desejadas, por exemplo [7].

No contexto das ORM, este problema pode se manifestar na forma de *N+1 queries*, resultando em múltiplas consultas ao banco de dados

quando apenas uma seria suficiente. Roksela et al[7] apontou um cenário comum: a primeira *query* ao banco de dados retorna uma lista de referências (1 *query* retorna N objetos), e depois N *queries* são feitas para acessar cada um desses objetos, causando *N+1 queries* ao banco de dados.

Em APIs GraphQL implementadas em Rails, o problema de *N+1 queries* é bastante comum no Active Record[13]. Neste artigo, nós focamos em comparar soluções para esse problema em APIs GraphQL desenvolvidas em Rails, tanto de forma quantitativa, analisando aspectos de performance vistos na Subseção 4.1, como de forma qualitativa, investigando aspectos das implementações apontados na Subseção 4.2. Dessa forma, visamos oferecer diretrizes valiosas para a comunidade de desenvolvimento.

Para ilustrar esse problema, usamos um simples sistema clone do Twitter, em que os usuários podem criar vários *tweets*, armazenados no banco de dados em suas respectivas tabelas. Na Listagem 1, vemos uma consulta ao banco de dados usando o Active Record para obter os *tweets* de id 5, 7 e 9. Ao tentar acessar o usuário para cada um desses *tweets*, terão sido feitas mais 3 queries, uma para cada usuário, desse modo ocorrendo o problema de *N+1 queries*. Ilustramos o problema na Figura 1.

```
1 Tweet.where(id: [5, 7, 9]).each do |tweet|
2   tweet.user
3 end
```

Listagem 1: Consulta do Active Record para obter *tweets* por ids

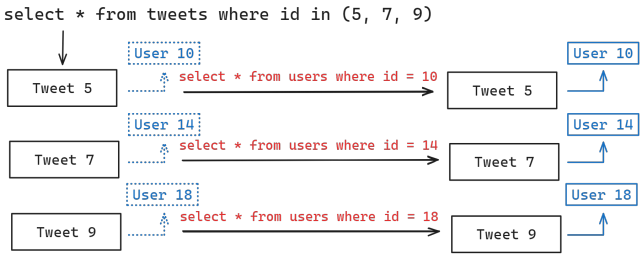

Figura 1: Demonstração de *N+1 queries* ao acessar autores dos *tweets*

A estrutura do trabalho está descrita a seguir. A Seção 2 apresenta os principais conceitos do GraphQL. A Seção 3 explica a necessidade de comparar as soluções para o problema de *N+1 queries* e quais soluções serão comparadas. A Seção 4 mostra como a comparação é feita. A Seção 5 exibe todos os passos da comparação. A Seção 6 compreende os resultados da comparação. A Seção 7 trata dos trabalhos relacionados na literatura. A Seção 8 conclui o artigo.

2 PRINCIPAIS CONCEITOS DO GRAPHQL

Nesta sessão apresentamos os principais conceitos do GraphQL, definidos de forma simples e eficaz por Gleison Brito e Marco Valente[1].

Em essência, o GraphQL dá aos clientes a capacidade de fazer consultas a um banco de dados representado por um *Schema* específico. Esse banco de dados é modelado por um *Schema* que toma a forma de um grafo direcionado, onde os nós são objetos que definem tipos e possuem um conjunto de campos. Cada campo, por sua vez, possui um nome e um tipo. Esses objetos podem ser acessados por meio do tipo *Query*, a porta de entrada da API GraphQL. Considerando o mesmo sistema clone do Twitter apontado na seção anterior, mostramos a definição dos tipos *Tweet*, *User* e *Query*, utilizando o GraphQL, na Listagem 2.

```
1  type Tweet {
2    id: Int!
3    content: String!
4    author: User!
5  }
6
7  type User {
8    id: Int!
9    nickname: String!
10 }
11
12 type Query {
13   feed: [Tweet!]
14 }
```

Listagem 2: Schema GraphQL exemplo

O GraphQL, além de propor um *Schema*, também fornece uma linguagem de consulta para os clientes. Os resultados das consultas são retornados em formato JSON, facilitando a análise e processamento pelos clientes. A Listagem 3 mostra um exemplo de *query* nessa linguagem, enquanto a Listagem 4 exibe seu resultado.

```
1  query ViewerFeed {
2    feed {
3      id
4      content
5      author {
6        id
7        nickname
8      }
9    }
10 }
```

Listagem 3: *Query* GraphQL exemplo para obter tweets do *feed*

Em um servidor GraphQL, os desenvolvedores estabelecem funções *resolver* para cada tipo de consulta. Quando uma consulta é realizada, estas funções são ativadas, buscando dados, comumente, de um banco de dados. No cenário que tratamos, utilizamos um banco relacional controlado pelo ORM do Rails. Na Listagem 5 temos a implementação do tipo *Query* em Rails. Nesse exemplo, vemos um *resolver* definido para o campo *feed* (linhas 4-6). Esse resolver garante o resultado visto na Listagem 4.

3 MOTIVAÇÃO

Com a crescente taxa de adoção do GraphQL em organizações, é cada vez mais importante estar ciente de como as APIs baseadas em GraphQL podem desempenhar [7].

A escolha da solução a ser adotada para tratar o problema de *N+1 queries* em APIs GraphQL desenvolvidas com Rails é um elemento vital que pode influenciar a eficiência e viabilidade do sistema. Há tanto soluções nativas do Active Record, quanto soluções *open-source* desenvolvidas pela comunidade no formato de *gems*. *Gems*

```
{
  "data": {
    "feed": [
      {
        "id": 5,
        "content": "Vou atras do Sasuke custe o que
            custar.",
        "author" : {
          "id": 10,
          "nickname": "Naruto Uzumaki"
        }
      },
      {
        "id": 7,
        "content": "Nao pertenco mais aqui nesta vila.",
        "author" : {
          "id": 14,
          "nickname": "Sasuke Uchiha"
        }
      },
      {
        "id": 9,
        "content": "Serei a mais forte.",
        "author" : {
          "id": 18,
          "nickname": "Sakura Haruno"
        }
      }
    ]
  }
}
```

Listagem 4: Resultado em formato JSON da *Query* GraphQL exemplo

```
class QueryType < Types::Base::Object
  field :feed, [TweetType], null: false

  def feed
    Tweet.where(id: [5, 7, 9])
  end
end
```

Listagem 5: Implementação exemplo em Ruby do *resolver* do campo *feed* no tipo *Query*

são pacotes de *software* que contém uma aplicação ou biblioteca Ruby empacotada, distribuídas pelo RubyGems [8]. Por sua vez, RubyGems é o serviço de hospedagem de *gems* da comunidade Ruby.

Quando falando de *gems* que resolvem um mesmo problema, é muito importante escolher a mais adequada. Se houver arrependimento em ter adicionado uma *gem* no sistema, é difícil voltar atrás depois do sistema ter evoluído e código ter sido escrito usando os recursos da *gem*[12].

Embora existam muitos recursos e discussões online[1] sobre como tratar o problema de *N+1 queries* [11], em nosso entendimento, **nota-se a ausência de estudos aprofundados que avaliem essas soluções de forma imparcial e acurada**. Uma análise detalhada é essencial para determinar qual delas é a mais adequada para as necessidades específicas de um projeto. A falta de tais informações pode culminar em escolhas que não atendem de forma ótima às demandas da aplicação.

[1]Discussão no Github https://github.com/rmosolgo/graphql-ruby/issues/189

3.1 Soluções Escolhidas

As soluções podem ser categorizadas de acordo com seus respectivos funcionamentos. Agrupamos as soluções em duas categorias de acordo com a forma que acessam o banco de dados: pré-carregamento de associações e carregamento em lote. Tendo em vista uma *query* base à uma tabela do banco de dados, o pré-carregamento de associações envolve a recuperação antecipada de entidades relacionadas a essa tabela. Já o carregamento em lote agrupa as várias operações de acesso aos dados relacionados e somente os obtém quando são explicitamente necessários. As Figuras 2 e 3 ilustram essas categorias.

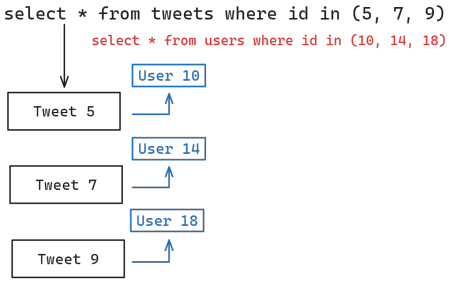

Figura 2: Estratégia de pré-carregamento de associações

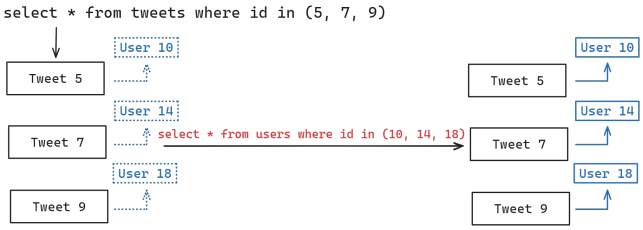

Figura 3: Estratégia de carregamento em lote

Assim, focamos a comparação entre as seguintes soluções:

- Pré-carregamento de associações
  Solução Includes Método *includes* do Active Record
- Carregamento preguiçoso em lote
  Solução GraphqlBatch *Gem* graphql-batch
  Solução BatchLoader *Gem* batch-loader

Escolhemos a Solução Includes porque julgamos importante comparar pelo menos uma solução nativa do Active Record, uma vez que não adiciona dependências externas ao projeto. Implementamos o método *includes* como sugerido no guia oficial do Rails [10] e não utilizamos outros métodos por serem usados da mesma forma, o que será um ponto de comparação qualitativa.

Quanto às *gems* escolhidas (Soluções GraphqlBatch e BatchLoader), elas são as mais populares na indústria, de acordo com a quantidade de *downloads* no RubyGems, que se propõem a resolver esse problema, sendo utilizadas por grandes empresas

- Graphql-batch[2] possui 14 milhões de downloads e foi concebida pelo Shopify, que continua fazendo amplo uso.

[2]Gem graphql-batch https://rubygems.org/gems/graphql-batch

- Batch-loader[3] possui 21 milhões de downloads e é utilizada pelo Netflix, por exemplo.

Embora funcionem da mesma forma ao acessar o banco de dados, é essencial destacar que as *gems* graphql-batch e batch-loader tem algumas diferenças relevantes entre si. A *gem* graphql-batch é específica ao GraphQL e utiliza promises[4] em sua implementação interna. No caso da *gem* batch-loader, ela é genérica e carrega os lotes de forma *lazy*[5] (preguiçosa).

4 MÉTODO PROPOSTO

Analisamos e comparamos as soluções apontadas na seção 3.1 entre si a fim de verificar tanto aspectos quantitativos como qualitativos, como explicado nas subseções a seguir. Dentro do contexto de nosso estudo, também avaliamos intencionalmente um cenário onde o problema *N+1 queries* não foi resolvido. Fizemos isso de forma deliberada para nos permitir quantificar o impacto real dessas soluções. Chamamos esse ponto de referência como ***Cenário Slow***.

4.1 Aspectos Quantitativos

Para comparar os aspectos quantitativos, escolhemos algumas métricas e ferramentas frequentemente utilizadas na comunidade[6][7] para comparação de performance.

4.1.1 IPS (Iterações Por Segundo). Utilizamos a *gem* benchmark-ips[8] para medir o IPS da solução. De forma simples, IPS se refere à quantidade de vezes que um pedaço de código pode ser executado dentro de um segundo. Quanto maior o IPS, mais rápido o código, mostrando quão mais rápida é uma solução em relação a outra.

4.1.2 Tempo Total de Execução. Recorremos ao módulo Benchmark[9] para calcular o tempo total de execução de cada solução. Quanto menor, melhor, o que indica que a solução levou menos tempo para retornar um resultado.

4.1.3 Memória Total Alocada. Lançamos mão da *gem* memory-profiler[10] para calcular a memória total alocada de cada solução. Quanto menor, melhor, exibindo que a solução demanda menos recursos do sistema para ser executada.

4.2 Aspectos Qualitativos

Além da análise quantitativa, também achamos relevante analisar qualitativamente as soluções alternativas, levando em consideração vantagens e desvantagens voltadas à escolha de adotar uma dessas soluções.

4.2.1 Usabilidade. Representa o nível de esforço necessário para usar as soluções. Quanto maior a usabilidade, menor o nível de esforço e mais fácil é sua utilização.

4.2.2 Flexibilidade. A flexibilidade se concentra na capacidade de lidar com diferentes condições ou requisitos, sem que grandes mudanças sejam necessárias.

4.2.3 Adição de Dependência Externa. Trata-se da necessidade de incluir bibliotecas ao projeto que não são fornecidadas nativamente em um sistema. Apesar da adição de dependências poder fornecer novas funcionalidades, implica também em fatores como manutenção, potenciais vulnerabilidades e carga inicial de aprendizado para usar tais funcionalidades.

5 DESENVOLVIMENTO

Para efetivamente analisar o problema das *N+1 queries* em APIs GraphQL em Rails, adotamos a seguinte abordagem: inicialmente, escolhemos uma aplicação base que serviria como nossa referência para desenvolver a aplicação de teste. Com essa aplicação em mente, avançamos para a implementação das entidades do banco de dados, estabelecendo seus relacionamentos específicos no Rails.

Em seguida, desenvolvemos o *Schema* GraphQL da nossa aplicação Rails. Isso envolveu a definição de cada tipo, juntamente com seus campos e *resolvers*. Com o *Schema* GraphQL pronto, passamos para a etapa de determinação das *queries* usadas em nossos testes quantitativos. Estas *queries* servem como nossos casos de teste para avaliar a performance das soluções.

Uma vez tendo as *queries* definidas, criamos scripts individuais no Rails, cujo propósito é executar as análises quantitativas. Estes scripts foram cuidadosamente projetados para garantir que os testes fossem repetíveis e consistentes. Na fase subsequente, executamos estes scripts, realizando a análise quantitativa e capturando os resultados para avaliação.

Finalmente, além da avaliação quantitativa, realizamos uma análise qualitativa seguindo os aspectos apontados na Subseção 4.2. Essa análise qualitativa complementou nossos resultados quantitativos, oferecendo uma visão abrangente do problema e das soluções propostas.

Definimos os seguintes passos a fim de atingir o objetivo de comparar as três soluções:

(1) Escolher a aplicação base
(2) Implementar as entidades do banco de dados e seus respectivos relacionamentos em Rails
(3) Desenvolver o *Schema* GraphQL da aplicação Rails, definindo cada tipo com seus respectivos campos e *resolvers*
(4) Determinar *queries* a serem efetuadas nos testes quantitativos
(5) Criar scripts individuais para executar as análises quantitativas em Rails
(6) Executar os scripts, efetuando a análise quantitativa
(7) Fazer a análise qualitativa

5.1 Escolha da Aplicação Base

Nos baseamos em uma aplicação clone do Twitter[11]. A nossa aplicação é um simples servidor GraphQL implementado em Rails seguindo a seguinte arquitetura: Cliente, GraphQL *Controller*, *Schema* GraphQL e um banco de dados. Para implementar os elementos

[3]Gem batch-loader https://rubygems.org/gems/batch-loader
[4]Promises/A+ https://promisesaplus.com/
[5]Lazy Loading
https://developer.mozilla.org/en-US/docs/Web/Performance/Lazy_loading
[6]Gem graphql-ruby public benchmark
https://github.com/rmosolgo/graphql-ruby/blob/master/benchmark/run.rb
[7]Gem ar_lazy_preload public benchmark
https://github.com/DmitryTsepelev/ar_lazy_preload/blob/master/benchmark/main.rb
[8]Gem benchmark-ips https://github.com/evanphx/benchmark-ips
[9]Módulo Benchmark https://ruby-doc.org/3.2.2/stdlibs/benchmark/Benchmark.html
[10]Gem memory-profiler https://github.com/SamSaffron/memory_profiler

[11]Aplicação Clone do Twitter
https://gist.github.com/DmitryTsepelev/d0d4f52b1d0a0f6acf3c5894b11a52ca

GraphQL, utilizamos a *gem* graphql-ruby[12], amplamente utilizada na indústria. O fluxo comum de uso é fazer uma *query* GraphQL por meio de uma requisição HTTP ao servidor, que será tratada pelo GraphQL *Controller* e por sua vez usará o *Schema* GraphQL para executar a *query* recebida. Durante a execução, o *Schema* GraphQL é responsável por obter os dados necessários por meio de *queries* ao banco de dados, utilizando o Active Record.

Essa aplicação permite consultarmos uma API GraphQL que fornece o *feed* do usuário autenticado no sistema, exibindo *tweets* de outros usuários. Para cada *tweet*, podemos obter seu *content* (conteúdo), seu *author* (autor) e seus *comments* (comentários). Cada *author* possui um *nickname* (apelido) e um *avatar*. Por sua vez, o *avatar* tem apenas a url da imagem de perfil do usuário. De cada *comment*, podemos obter seu *content* e seu *author*, de forma bem similar ao *tweet*. A arquitetura da aplicação pode ser vista na Figura 4.

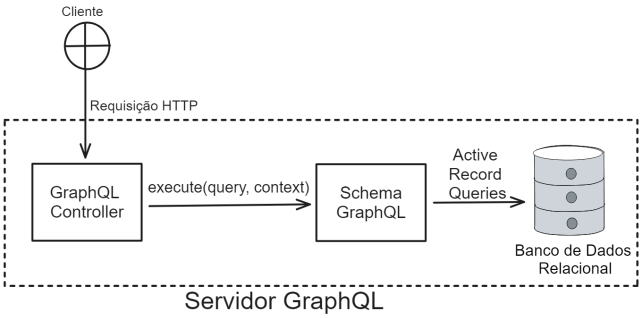

Figura 4: Arquitetura da aplicação Rails

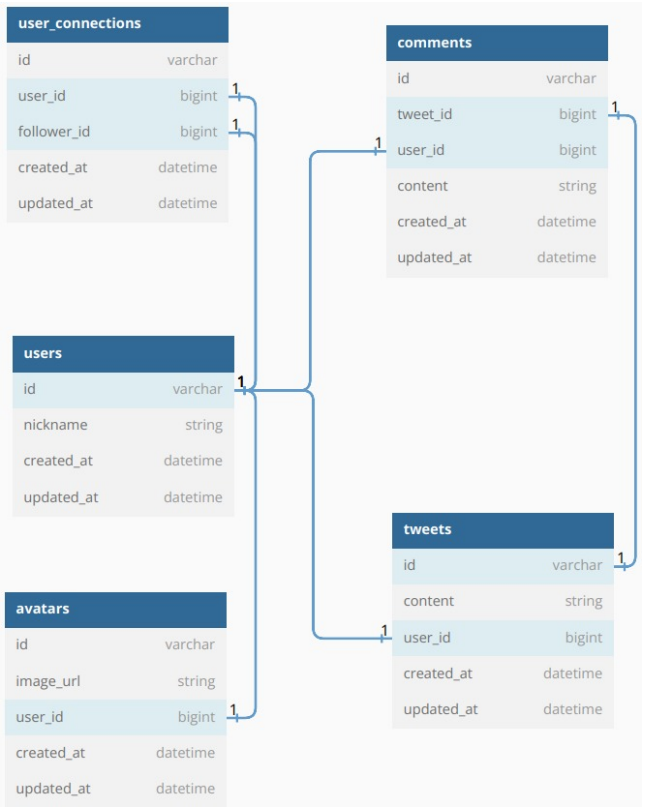

Figura 5: Diagrama Entidade-Relacionamento do banco de dados

5.2 Implementação das Entidades e Relacionamentos em Rails

No esquema do banco de dados proposto, temos várias entidades interconectadas para modelar o sistema clone do Twitter, ou seja, um sistema típico de rede social. Cada *user* tem um único *avatar*, que representa seu perfil e tem como atributo a imagem do usuário. Definimos a entidade *avatar* à parte porque ele pode, em uma aplicação real, compreender outros atributos, como configurações para o usuário. Esse usuário pode criar *tweets*, que não passam de mensagens, e também pode responder ou interagir com os *tweets* de outros usuários por meio de *comments* (comentários). Na Figura 5, temos o diagrama entidade-relacionamento do banco de dados.

Para aprofundar as relações sociais, o esquema permite que um usuário siga outros usuários. Essas relações de *follow* (seguir são representadas na tabela *user_connection* (conexão do usuário), que registra quem cada usuário decide seguir. Esta representação permite um fluxo de informações onde um usuário pode ver as atualizações de quem ele decide seguir, simulando uma típica dinâmica de rede social.

Cada *tweet*, além de pertencer a um usuário, pode ser a base para uma série de comentários. Esses comentários, assim como os *tweets*, estão vinculados a um usuário específico, indicando quem os escreveu.

[12]GraphQL Ruby https://graphql-ruby.org/

Assim, capturamos as principais atividades e interações em uma plataforma de mídia social: postar mensagens, comentar sobre elas, representar-se por meio de um avatar e formar conexões seguindo outros usuários.

Em Rails, as entidades são implementadas na forma de *models* (modelos), classes que permitem o acesso ao banco de dados. Por sua vez, os relacionamentos entre entidades são definidos por meio de *associations* (associações) entre *models*, implementados de forma declarativa em cada *model*[9]. Na Listagem 6 temos a implementação dos seguintes relacionamentos:

- *User* 1 : 1 *Avatar*
- *User* 1 : N *Tweet*
- *User* 1 : N *Comment*
- *User* N : N *User*
- *Tweet* 1 : N *Comment*

Em nossa implementação, fizemos uso de 4 associações diferentes. A função *has_one* estabelece uma relação um-para-um e foi usada para definir que cada usuário tem apenas um *avatar* associado a ele.

Já a função *has_many* define uma relação um-para-muitos. Como visto na classe *User*, para estabeler que um usuário pode criar vários *tweets*, mas cada *tweet* individual pertence a apenas um usuário.

```ruby
class User < ApplicationRecord
  has_one :avatar

  has_many :tweets
  has_many :comments

  has_many :followers_connections,
    class_name: "UserConnection",
    foreign key: :user_id
  has_many :followed_connections,
    class_name: "UserConnection",
    foreign_key: :follower_id

  has_many :followers,
    through: :followers_connections,
    class_name: "User",
    source: :follower
  has_many :followed_users,
    through: :followed_connections,
    class_name: "User",
    source: :user,
end

class Avatar < ApplicationRecord
  belongs_to :user
end

class Tweet < ApplicationRecord
  belongs_to :user
  has_many :comments
end

class Comment < ApplicationRecord
  belongs_to :user
  belongs_to :tweet
end

class UserConnection < ApplicationRecord
  belongs_to :user
  belongs_to :follower, class_name: "User"
end
```

Listagem 6: Associações entre models

Estabelecemos a relação de pertencimento com o *belongs_to*. É frequentemente a contraparte do *has_one* ou *has_many*. Sua declaração na classe *Avatar* afirma que um *avatar* está associado a um e apenas um usuário.

E finalmente, há o modificador *through*, que é frequentemente usado com *has_many* para estabelecer associações muitos-para-muitos através de uma tabela intermediária. Utilizamos para representar as relações de *follow* entre usuários.

5.3 Desenvolvimento do *Schema* GraphQL em Rails

A construção e evolução do *Schema* GraphQL desempenhou um papel crucial na comparação das soluções. Inicialmente, adotamos uma abordagem simplista, criando um *Schema* sem nenhuma solução específica para o problema, visando reproduzir o problema de *N+1 queries*. Assim, tivemos o ponto de referência mencionado na Seção 4, o *Cenário Slow*. Vemos o *Schema* inicial na Figura 6.

No início, implementamos o tipo *Viewer* com apenas um campo *feed* que utiliza o *resolver* FeedResolver para buscar e construir o *feed* do usuário. Nele, utilizamos a função *for* definida no módulo FeedBuilder, cujo retorno são os 10 primeiros *tweets* mais recentes dos *followed users* (usuários seguidos) pelo usuário recebido como argumento. Na Listagem 7 vemos a implementação desse módulo.

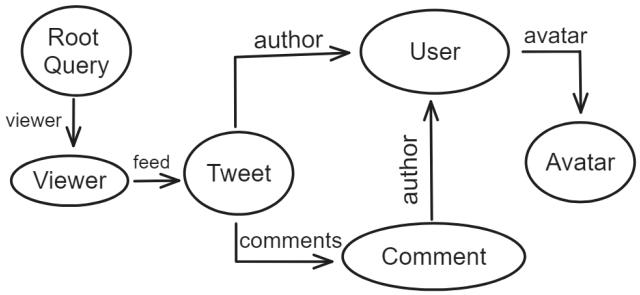

Figura 6: *Schema* GraphQL inicial

```ruby
module FeedBuilder
  module_function

  def for(user)
    Tweet.where(user: user.followed_users)
         .order(created_at: :desc)
         .limit(10)
  end
end
```

Listagem 7: Módulo FeedBuilder responsável pela *query* de *tweets*

Os outros tipos do *Schema* foram desenvolvidos seguindo a abordagem simplista, apenas definindo campos simples. Por meio da Listagem 8 vemos as implementações iniciais do tipo Viewer mencionado e como exemplo da implementação dos outros tipos vemos o tipo Comment.

```ruby
class ViewerType < Types::Base::Object
  field :feed, resolver: Resolvers::FeedResolver
end

class CommentType < Types::Base::Object
  field :content, String, null: false
  field :author, UserType, null: false, method: :user
end

module Resolvers
  class BaseFeedResolver < Base
    type [Types::TweetType], null: false

    def resolve
      raise NotImplementedError
    end
  end

  class FeedResolver < BaseFeedResolver
    def resolve
      FeedBuilder.for(current_user)
    end
  end
end
```

Listagem 8: Implementação inicial do tipo GraphQL Viewer e Comment

Na primeira evolução do *Schema*, acrescentamos um novo campo chamado *feed_with_includes* no tipo *Viewer*, para compreender e demonstrar a eficácia da Solução Includes. O *resolver* desse campo, FeedResolverIncludes, incorporou a otimização chamando o método

includes para pré-carregar associações e evitar múltiplas *queries*. Apontamos essa evolução na Listagem 9.

```
class ViewerType < Types::Base::Object
  field :feed, resolver: Resolvers::FeedResolver
  field :feed_with_includes, resolver: Resolvers::
      FeedResolverIncludes
end

module Resolvers
  class FeedResolverIncludes < BaseFeedResolver
    def resolve
      user_includes = {user: :avatar}

      FeedBuilder.for(current_user).includes(
          user_includes, comments: user_includes)
    end
  end
end
```

Listagem 9: Implementação final do tipo GraphQL Viewer

Para explorar as Soluções GraphqlBatch e BatchLoader, evoluímos o *Schema* incrementando os tipos como *Tweet*, *User* e *Comment* com novos campos específicos para essas soluções. Por exemplo, no tipo *Comment*, originalmente, tínhamos um simples campo *author* que trazia o usuário associado ao comentário. Com a evolução, adicionamos campos adicionais: *author_graphql_batch* e *author_batch_loader*, que empregam respectivamente as Soluções GraphqlBatch e BatchLoader para resolver o problema de *N+1 queries*. Essa alteração está na Listagem 10.

```
class CommentType < Types::Base::Object
  field :content, String, null: false
  field :author, UserType, null: false, method: :user

  field :author_graphql_batch, UserType, null: false

  def author_graphql_batch
    Loaders::AssociationLoader.for(Comment, :user).load(
        object)
  end

  field :author_batch_loader, UserType, null: false

  def author_batch_loader
    BatchLoader::GraphQL.for(object.user_id).batch do |
        user_ids, loader|
      User.where(id: user_ids).each { |author| loader.
          call(author.id, author) }
    end
  end
end
```

Listagem 10: Implementação final do tipo GraphQL *Comment*

O *Schema* final, após as evoluções mencionadas, é visto na Figura 7. Essa abordagem progressiva de desenvolver e evoluir o *Schema* GraphQL permitiu uma comparação justa e detalhada entre as várias soluções, explicada na próxima subseção.

5.4 Queries Determinadas

O *Schema* GraphQL foi estrategicamente construído para permitir testar cada solução de forma isolada e direta. A chave para essa abordagem modular foi incorporar diferentes *resolvers* no *Schema*, cada um implementando uma das soluções. Dito isso, para obter

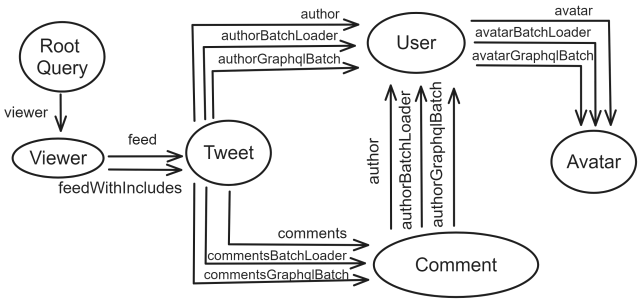

Figura 7: *Schema* GraphQL completo

uma visão mais abrangente na análise de performance e eficácia das soluções propostas, não nos restringimos a uma única abordagem de consulta. Em nossos testes, definimos dois tipos distintos de *queries*: Completa e Parcial.

Uma *query* **Completa** solicita todo o fluxo de dados apontado na Subseção 5.1. Como ponto de partida, definimos a *query* completa SlowCompleta, padrão, sem a aplicação de qualquer solução, ou seja, simulando o *Cenário Slow*. A partir da adaptação dessa *query*, pudemos testar as diferentes implementações garantindo o mesmo resultado, porém com diferentes performances.

Para testar a Solução Includes, definimos a *query* IncludesCompleta, apenas solicitando o campo feed_with_includes ao invés do campo feed. Do mesmo modo, para avaliar as Soluções GraphqlBatch e BatchLoader, criamos as *queries* GraphqlBatchCompleta e BatchLoaderCompleta, solicitando os campos especificamente adicionados para cada uma dessas soluções. Por exemplo, para avaliar a Solução GraphqlBatch, basta trocarmos na *query* o campo *author* pelo campo *author_graphql_batch*, enquanto no caso da Solução BatchLoader, é suficiente trocarmos pelo campo *author_batch_loader*. As *queries* SlowCompleta, IncludesCompleta, GraphqlBatchCompleta e BatchLoaderCompleta apontadas estão presentes na Figura 8.

Em contraste, uma *query* **Parcial** é simplesmente uma versão reduzida de uma *query* Completa em que omitimos intencionalmente os *comments* associados a cada *tweet*. Assim, para cada *query* Completa definida, também criamos uma *query* Parcial.

Por meio dessa abordagem, conseguimos investigar mais profundamente as soluções quanto ao desempenho, às vantagens e possíveis limitações de cada solução, tanto em situações de alta demanda de dados quanto em operações mais simplificadas.

5.5 Criação dos Scripts

Antes de criar os *scripts*, isolamos a etapa de execução do *Schema* GraphQL, a parte do servidor que foi testada. Para isso, desenvolvemos o **BaseExecutor**. Esta classe tem a responsabilidade de executar uma *query* diretamente no *Schema* GraphQL, abstraindo a etapa da requisição HTTP mostrada na Figura 4. A classe estabelece também uma assinatura padrão para um método, chamado *query_string*. Aqueles que herdam dessa classe base têm a tarefa de implementar este método. Adicionalmente, o BaseExecutor define um método *context*, que provê informações sobre o usuário atual do sistema, que é um simples dicionário contendo dados específicos da aplicação, como o usuário autenticado por exemplo. Nele, apenas passamos um mesmo usuário como usuário atual do sistema,

```
query SlowCompleta {
  viewer {
    feed {
      content
      author {
        nickname
        avatar {
          imageUrl
        }
      }
      comments {
        content
        author {
          nickname
          avatar {
            imageUrl
          }
        }
      }
    }
  }
}
```

```
query IncludesCompleta {
  viewer {
    feedWithIncludes {
      content
      author {
        nickname
        avatar {
          imageUrl
        }
      }
      comments {
        content
        author {
          nickname
          avatar {
            imageUrl
          }
        }
      }
    }
  }
}
```

```
query GraphqlBatchCompleta {
  viewer {
    feed {
      content
      authorGraphqlBatch {
        nickname
        avatarGraphqlBatch {
          imageUrl
        }
      }
      commentsGraphqlBatch {
        content
        authorGraphqlBatch {
          nickname
          avatarGraphqlBatch {
            imageUrl
          }
        }
      }
    }
  }
}
```

```
query BatchLoaderCompleta {
  viewer {
    feed {
      content
      authorBatchLoader {
        nickname
        avatarBatchLoader {
          imageUrl
        }
      }
      commentsBatchLoader {
        content
        authorBatchLoader {
          nickname
          avatarBatchLoader {
            imageUrl
          }
        }
      }
    }
  }
}
```

Figura 8: Queries Completas

visando garantir o mesmo resultado entre execuções. A execução de uma *query*, então, ocorre ao passar ambos a *query_string* e o *context* como parâmetros no método *execute* do *Schema* GraphQL.

Construído esse alicerce, **modelamos um Executor filho específico para cada solução proposta**, de modo que cada executor tinha sua respectiva *query* Completa embutida em sua implementação do método *query_string*. Claro, não nos esquecemos de criar um Executor adicional para o *Cenário Slow*. Dito isso, criamos as classes **SlowExecutor**, **IncludesExecutor**, **GraphqlBatchExecutor** e **BatchLoaderExecutor**, vistas na Figura 9.

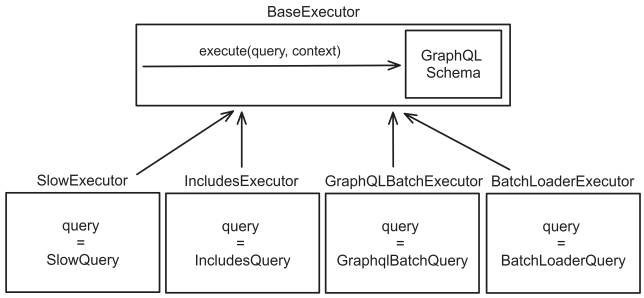

Figura 9: Esquema de herança com cada Executor

Em seguida, para cada aspecto quantitativo mencionado na subseção 4.1, criamos um *script*, cujo propósito é executar e medir as execuções de cada Executor. O fluxo de execução dos scripts foi meticulosamente determinado:

(1) Instanciação do Executor e Limpeza do Ambiente
(2) O Executor é então executado dentro de uma função anônima, passada para o método de comparação adequado.
(3) Este ciclo de instanciação e limpeza do ambiente, seguido pela execução de um Executor é feito na ordem: SlowExecutor, IncludesExecutor, GraphqlBatchExecutor e BatchLoaderExecutor.
(4) Concluídas as execuções, realizamos a Limpeza da Cache do BatchLoader.

O passo de **Instanciação do Executor** apenas cria uma nova instância do Executor que está sendo executado no momento. Na **Limpeza do Ambiente**, por sua vez, executamos manualmente o *Garbage Collector* e em seguida o desabilitamos. Externos ao método de comparação, fizemos esses passos para evitar impactos nos resultados dos testes e apenas avaliarmos a execução de cada *query* isoladamente. Ao final do processo, efetuamos a **Limpeza da Cache do BatchLoader**, em que limpamos também a *cache* específica da Solução BatchLoader. Isso normalmente seria feito entre requisições HTTP, considerando a arquitetura da Figura 4. Para garantir precisão e uma amostragem robusta, **repetimos esse processo inteiro em um loop, resultando em dez execuções distintas para cada Executor.**

Todo esse processo está ilustrado visualmente na Figura 10, facilitando a compreensão do fluxo de execução e das etapas envolvidas. Por fim, coletamos os dados ao longo das execuções e os armazenados meticulosamente em arquivos CSV, um para cada script. Posteriormente, esses arquivos foram usados para gerar gráficos, permitindo uma análise visual mais intuitiva dos resultados, apresentados na Seção 6. No caso da avaliação da *query* Parcial, apenas ajustamos manualmente as *queries* nos executores e agrupamos os dados em um CSV distinto.

A execução de cada script foi realizada usando o comando *rails runner*[13], que permite a execução de código Ruby no contexto de uma aplicação Rails. Especificamos o ambiente de produção utilizando a *flag -e*. Vale mencionar que este ambiente manteve as configurações padrão encontradas em um novo projeto Rails.

6 RESULTADOS

Após a execução de cada *script*, pudemos enfim realizar a análise comparativa apontada na seção 4. Ressaltamos que aplicamos o

[13]Linha de Comando Rails https://guides.rubyonrails.org/command_line.html#bin-rails-runner

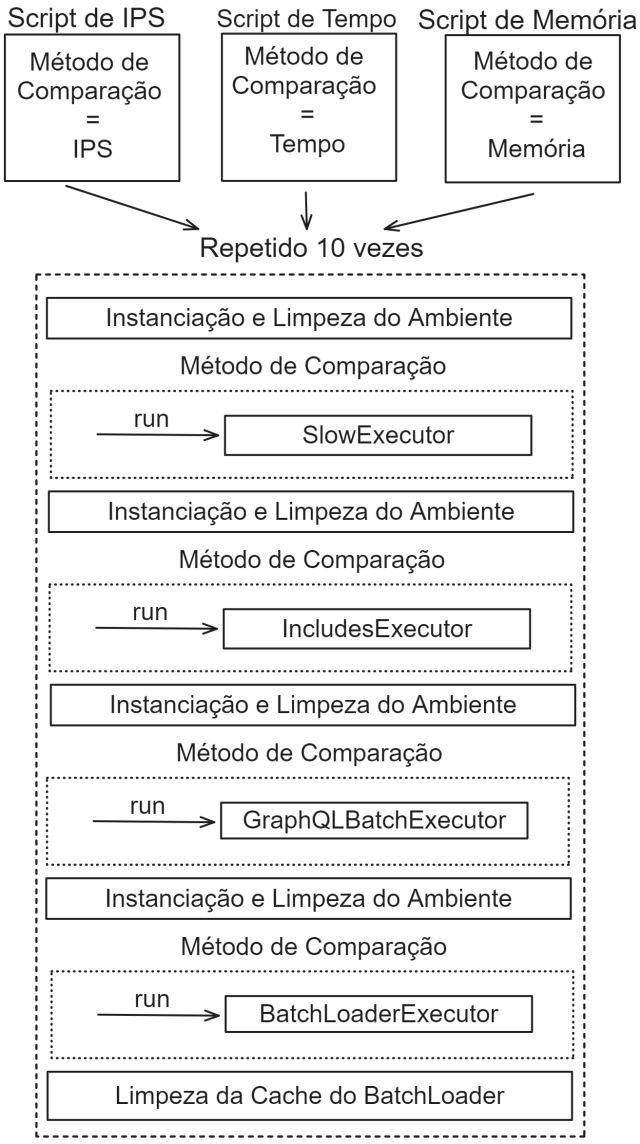

Figura 10: Esquema de execução dos scripts

método Tukey[14] para lidar com dados atípicos e os removemos dos gráficos para que a visualização não fosse comprometida, para cada uma das métricas.

6.1 IPS

Ao avaliar os resultados medianos das dez execuções em termos de IPS, observamos diferenças notáveis entre as soluções propostas e o *Cenário Slow*. Os resultados obtidos podem ser vistos lado a lado para cada tipo de *query* na Tabela 1.

Para a *query* Completa, o *Cenário Slow* apresentou uma performance mediana de 29 i/s, estabelecendo nosso padrão de referência. Utilizando a Solução Includes, vemos um notável aumento para

[14]https://towardsdatascience.com/detecting-and-treating-outliers-in-python-part-1-4ece5098b755

Tabela 1: Mediana aproximada do IPS para cada Executor e tipo de *query*

Executor	Query Completa	Query Parcial
Slow	29 i/s	85 i/s
Includes	87 i/s	121 i/s
GraphQLBatch	79 i/s	170 i/s
BatchLoader	122 i/s	231 i/s

87 i/s, demonstrando uma melhoria considerável na eficiência. Já a Solução GraphQLBatch apresentou um desempenho de 79 i/s, ligeiramente abaixo da Solução Includes, mas ainda muito acima do cenário base. A Solução BatchLoader destacou-se, alcançando uma marca impressionante de 122 i/s, tornando-se a solução de melhor performance para a *query* Completa. Apresentamos os resultados de IPS da *query* Completa por meio de *violin plots* em um gráfico geral na Figura 11. Para ter uma visão melhor de cada solução individualmente, temos as Figuras 12, 13, 14 e 15, para as classes SlowExecutor, IncludesExecutor, GraphqlBatchExecutor e BatchLoaderExecutor respectivamente.

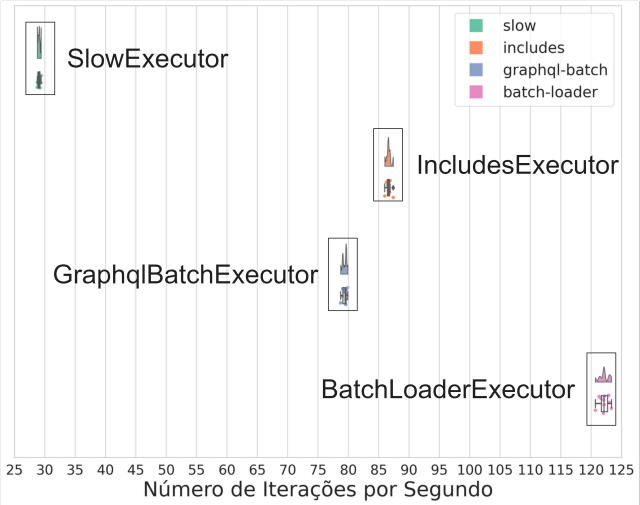

Figura 11: Gráfico com valores de IPS para cada solução - *query* Completa

Para a *query* Parcial, o *Cenário Slow* registrou 85 i/s como mediana. Vemos que nessa *query* a Solução Includes, com resultado de 121 i/s, foi ultrapassada pela Solução GraphqlBatch e suas 170 i/s. A Solução BatchLoader prevaleceu novamente com um desempenho de 231 i/s, solidificando sua posição como a solução mais eficaz entre as testadas. Na Figura 16 temos os resultados de IPS de forma abrangente para a *query* Parcial, também em *violin plots*. Novamente, temos visões individuais nas Figuras 17, 18, 19 e 20.

Esses dados mostram que, embora todas as soluções otimizadas proporcionem ganhos significativos em relação ao cenário padrão, o BatchLoader consistentemente se destaca em termos de eficiência.

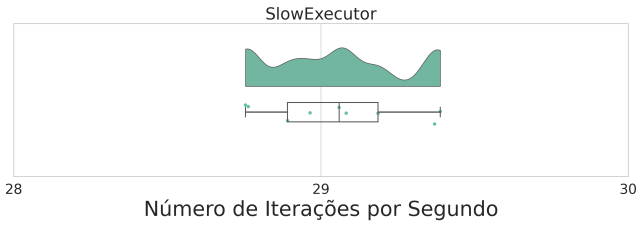

Figura 12: Gráfico com valores de IPS para o *Cenário Slow* - *query* Completa

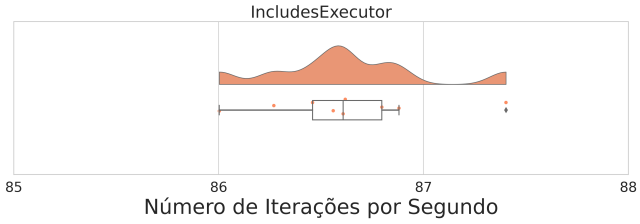

Figura 13: Gráfico com valores de IPS para a Solução Includes - *query* Completa

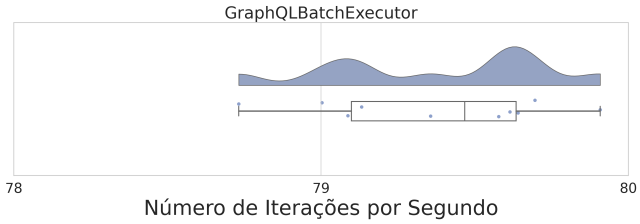

Figura 14: Gráfico com valores de IPS para a Solução Graphql-Batch - *query* Completa

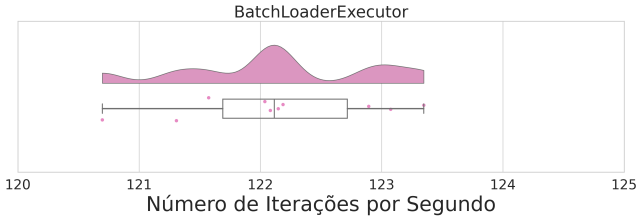

Figura 15: Gráfico com valores de IPS para a Solução Batch-Loader - *query* Completa

6.2 Tempo Total de Execução

No tocante ao tempo total de execução, ao analisar os resultados medianos das dez execuções é perceptível novamente que todas as soluções propostas para otimização tiveram um impacto significativo na melhoria do tempo de resposta em comparação com o *Cenário Slow*. De forma similar ao IPS, podemos visualizar os resultados obtidos na Tabela 2.

Quanto às queries Completas, o *Cenário Slow* mostrou um tempo de 41 ms, servindo como uma linha base para as comparações. A

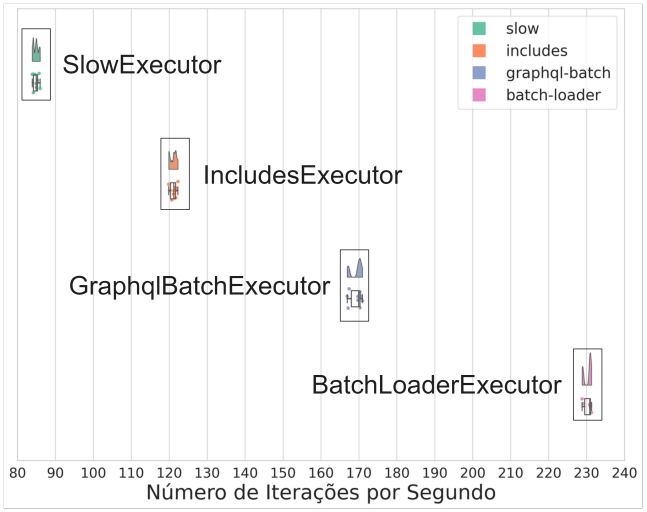

Figura 16: Gráfico com valores de IPS para cada solução - *query* Parcial

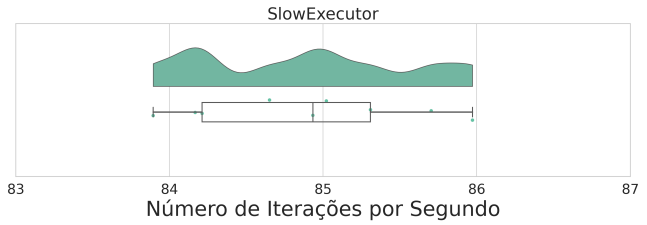

Figura 17: Gráfico com valores de IPS para o *Cenário Slow* - *query* Parcial

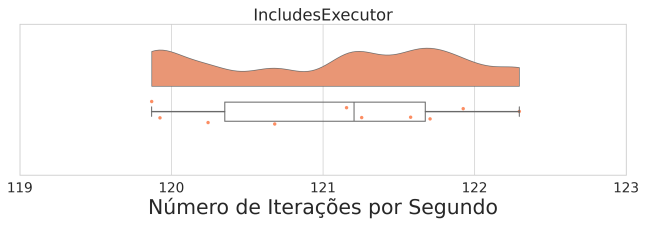

Figura 18: Gráfico com valores de IPS para a Solução Includes - *query* Parcial

Solução Includes conseguiu reduzir esse tempo para 15 ms, representando uma economia substancial. No entanto, a Solução GraphQL-Batch mostrou-se ligeiramente mais lenta, com 16 ms, mas ainda assim muito mais eficiente do que o *Cenário Slow*. A Solução Batch-Loader teve o melhor tempo de resposta, com um tempo mediano de apenas 14 ms, demonstrando ser a solução mais rápida para a *query* Completa. A Figura 21 mostra os resultados das execuções abrangentemente.

Quando olhamos para as queries Parciais, o *Cenário Slow* teve um tempo mediano de 11 ms. O que identificamos aqui é a Solução

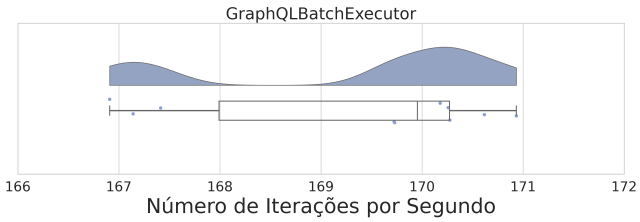

Figura 19: Gráfico com valores de IPS para a Solução Graphql-Batch - *query* Parcial

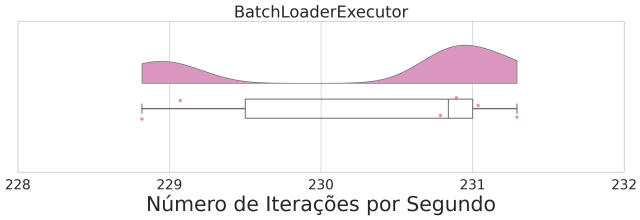

Figura 20: Gráfico com valores de IPS para a Solução Batch-Loader - *query* Parcial

Tabela 2: Mediana aproximada do tempo total de execução em milissegundos para cada Executor e tipo de *query*

Executor	Query Completa	Query Parcial
Slow	41 ms	11 ms
Includes	15 ms	9 ms
GraphQLBatch	16 ms	7 ms
BatchLoader	14 ms	6 ms

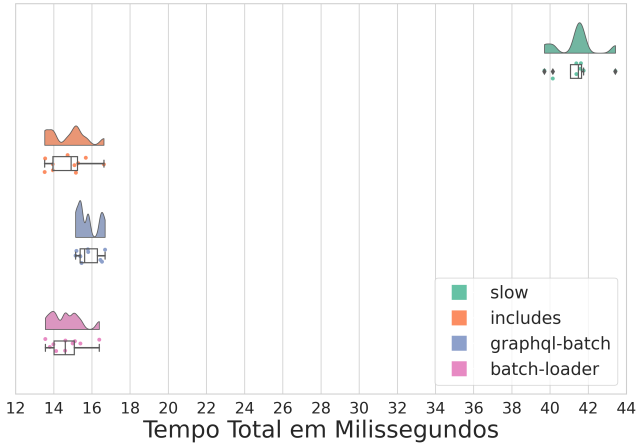

Figura 21: Tempo total de execução - *query* Completa

Includes, com 9 ms, foi novamente excedida pela Solução Graphql-Batch, com 7 ms. Mais uma vez, a Solução BatchLoader se destacou como a solução mais eficaz, entregando a *query* em apenas 6 ms. Esses números estão presentes na Figura 22.

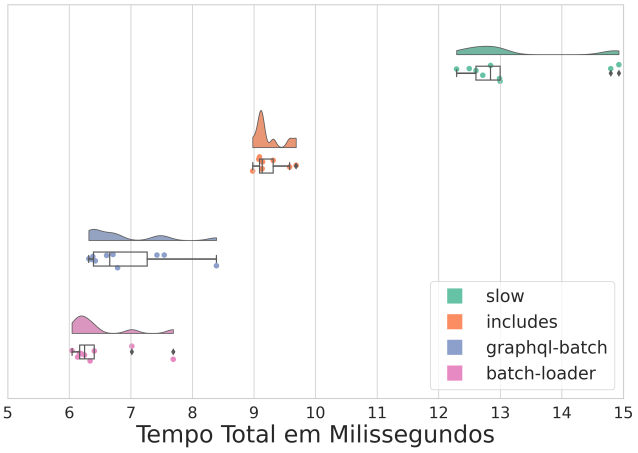

Figura 22: Tempo total de execução - *query* Parcial

Esses resultados reiteram que, enquanto todas as soluções otimizadas oferecem uma melhoria substancial em relação ao Soluçao Slow e a Solução BatchLoader tem uma vantagem consistente, seja em queries Completas ou Parciais. Além disso, podemos ver que os resultados nos testes de IPS se traduziram para o tempo total de execução, como o fato da Solução GraphqlBatch ter ultrapassado a Solução Includes na *query* Parcial.

6.3 Memória Alocada

Em resumo, a memória alocada se refere ao espaço total de memória solicitado durante a execução de um bloco de código. Notavelmente, o script de análise de memória produziu resultados consistentes em todas as dez execuções, mostrando o mesmo valor para cada um dos executores. Novamente, os exibimos na Tabela 3.

Tabela 3: Memória alocada em MB para cada Executor e tipo de *query*

Executor	Query Completa	Query Parcial
Slow	1.05 MB	0.35 MB
Includes	0.47 MB	0.36 MB
GraphQLBatch	0.53 MB	0.22 MB
BatchLoader	0.52 MB	0.21 MB

Com relação à memória alocada, no cenário da *query* Completa, o *Cenário Slow* consumiu cerca de 1.05 MB. A Solução Includes reduziu a alocação para 0.47 MB, enquanto as Soluções Graphql-Batch e BatchLoader resultaram em alocações de 0.53 MB e 0.52 MB, respectivamente. Este consumo mais elevado para as soluções baseadas em lotes pode estar potencialmente relacionado à sobrecarga introduzida por classes adicionais do GraphQL.

Quando observamos a *query* Parcial, o *Cenário Slow* tem um consumo de 0.35 MB. Sobre a Solução Includes, fica visível a influência de pré-carregar todas as associações, ultrapassando até mesmo o

cenário de referência e alocando 0.36MB. Já a Solução GraphqlBatch teve 0.22 e de forma bem similar a Solução BatchLoader alocou 0.21 MB.

6.4 Análise Qualitativa

Esta análise proporciona entendimentos sobre as qualidades e potenciais limitações de cada solução, considerando o contexto de APIs GraphQL implementadas em Rails. A Figura 23 mostra essa comparação visualmente de acordo com cada aspecto visto na Subseção 4.2 para cada solução analisada.

Aspecto \ Solução	Includes	GraphQLBatch	BatchLoader
Usabilidade	+++	++	+
Flexibilidade	+	++	+++
Adição de Dependência Externa	+++	++	++

Figura 23: Tabela de análises qualitativas com aspectos por solução

6.4.1 Usabilidade. A integração direta fornecida pela Solução Includes implica em alta usabilidade para desenvolvedores familiarizados com Rails. Quanto à Solução GraphqlBatch, seu uso é intuitivo quando já há um conhecimento com relação à *gem* graphql-ruby. No entanto, aqueles sem experiência prévia com o GraphQL e seu ecossistema enfrentarão uma curva de aprendizado. Por sua vez, a Solução BatchLoader "é mais díficil de começar do que com algo específico ao GraphQL, como graphql-batch"[13].

6.4.2 Flexibilidade. No tocante à flexibilidade, a Solução Includes começa a ter dificuldades. É uma solução ótima para situações padrão, em *queries* mais diretas, porém não se adapta tão bem a *queries* mais complexas, por exemplo quando pré-carregando associações polimórficas. Esse caso é facilmente resolvido pela Solução BatchLoader e está exemplificado em seu repositório Github. Isso é apenas um dos pontos que mostra como a Solução BatchLoader é flexível. A sua utilização pode ser facilmente customizada quando *queries* mais compostas forem necessárias[6]. Além disso, ela ainda pode ser utilizada de forma simples em situações que não tenham envolvimento algum com GraphQL, como em APIs REST. Isso a torna incrivelmente poderosa. A Solução GraphqlBatch também oferece mais flexibilidade do que a solução nativa, entretanto menos do que a Solução BatchLoader, justamente por ser específica para o GraphQL.

6.4.3 Adição de Dependência Externa. A única solução dentre as três que não exige a adição de uma dependência externa é a Solução Includes, uma vez que é um recurso nativo do Rails. Para projetos que desejam minimizar o número de dependências ou apenas em busca de uma solução rápida para *queries* mais simples, ela pode ser ideal. Apesar de ser uma dependência externa, a Solução GraphqlBatch é focada em resolver o problema *N+1 queries* no contexto da *gem* graphql-ruby, podendo ser vista como uma extensão natural. De forma similar, a Solução BatchLoader é uma dependência externa. Apesar disso, sua flexibilidade pode evitar que outras dependências venham a ser necessárias, já que sua aplicabilidade é mais ampla.

6.5 Ameaças à Validade

Alguns aspectos merecem ser destacados ao comparar as soluções para o problema *N+1 queries*, porque podem representar potenciais ameaças à validade das conclusões extraídas:

O *Schema* Graphql e as entidades do banco de dados foram estruturados com o objetivo específico de simular o problema de *N+1 queries*. Também é valido apontar que apenas analisamos com respeito a duas queries ao *Schema* GraphQL. Embora isso seja essencial para a reprodutibilidade do problema em um ambiente controlado, pode não refletir a complexidade e variações encontradas em sistemas reais. Além disso, há cenários em aplicações do mundo real que podem não ter sido totalmente capturados nesta configuração.

O estado do banco de dados ser consistente durante os testes é uma vantagem, por eliminar variáveis indesejadas que poderiam influenciar os resultados. Porém, isso também pode ser uma limitação. Em sistemas reais, a condição do banco de dados pode variar, por exemplo em termos de volume de dados ou atividades simultâneas, o que pode afetar o desempenho das soluções. Embora os resultados sejam mais confiáveis com o estado consistente do banco de dados, reconhecemos que diferentes estados podem produzir resultados distintos.

A definição de uma ordem constante para a chamada dos executores visou minimizar variáveis externas que possam afetar os resultados. No entanto, mesmo que os testes tenham sido projetados para serem o mais isolados possível, a ordem de execução, em teoria, poderia influenciar os resultados devido a aspectos como *cache* ou alocação de recursos do sistema. Embora a probabilidade seja mínima, não podemos desconsiderar completamente essa variável.

Utilizamos de uma única aplicação *toy* (um sistema sem utilização prática de fato projetado principalmente para fins experimentais ou didáticos) como base para os testes pode não refletir completamente a realidade de aplicações empresariais ou em produção. Aplicações do mundo real podem ter esquemas mais complexos, conjuntos de dados maiores e interações mais entrelaçadas, o que poderia influenciar o desempenho das soluções testadas.

A análise qualitativa foi conduzida por um único indivíduo, o que pode introduzir um viés subjetivo. Diferentes desenvolvedores podem ter diferentes percepções e experiências, possivelmente levando a diferentes conclusões ou ênfases na análise das soluções. Idealmente, poderíamos tornar a avaliação mais robusta considerando múltiplos avaliadores para garantir uma visão mais diversificada.

A relevância do estudo não é diminuída ao reconhecer essas ameaças, no entanto destaca áreas em que futuras pesquisas podem se aprofundar para oferecer uma compreensão mais completa quanto ao problema de *N+1 queries* em APIs GraphQL em Rails.

7 TRABALHOS RELACIONADOS

Antes de finalizar esta análise, consideramos pertinente citar estudos relacionados ao GraphQL, visando demonstrar o quanto sua relevância acadêmica vem crescendo.

O GraphQL é frequentemente comparado à arquitetura REST, de forma quantitativa. Gleison Brito e Marco Valente[1] realizaram um experimento controlado para comparar ambas tecnologias em aspectos quantitativas e qualitativos. Sobre a análise qualitativa, diferentemente da nossa abordagem em que apenas tivemos as visões de uma única pessoa, eles entrevistaram 38 desenvolvedores de software, funcionários do Github. Na perspectiva quantitativa, apesar de não fazermos nenhuma comparação com REST, é valido mostrar que eles não eliminaram as requisições HTTP ao investigar a performance de ambas as tecnologias. Em nossa visão, isso pode gerar ruídos devido aos aspectos de redes inerentes à essas requisições.

No trabalho de Roksela et al[7], uma aplicação para simular o ambiente de uma API web foi desenvolvida para comparar as diferentes estratégias de execução de *queries* GraphQL. Um dos objetivos desse estudo foi avaliar a resistência de cada estratégia ao problema *N+1*. Por exemplo, no trabalho mencionado estratégias de *cache* e carregamento em lote foram comparadas entre si e não tiveram diferenças significativas quanto ao tempo total de execução. Em nossos testes, o uso de *caches* foi explicitamente eliminado, ou seja, nem chegamos a comparar esse cenário.

Estes estudos, quando somados às nossas observações, proporcionam uma visão mais abrangente sobre o GraphQL como uma tecnologia cada vez mais relevante no mercado de desenvolvimento de software.

8 CONCLUSÃO

O problema *N+1* pode ocorrer de formas diferentes em vários sistemas e diversas tecnologias, como em APIs REST, não se restringindo apenas ao problema de *N+1 queries*. Neste estudo, comparamos de forma específica três soluções distintas para abordar o problema de *N+1 queries* em APIs GraphQL implementadas em Rails: a Solução Includes, a Solução GraphqlBatch e a Solução BatchLoader. A motivação para tal comparação deriva da prevalência deste problema em muitos projetos que utilizam GraphQL e da necessidade de otimização para melhorar a eficiência e a experiência do usuário.

Por meio da implementação e teste dessas soluções, conseguimos avaliar aspectos quantitativos, por meio da medição do tempo total de execução, do IPS e do consumo de memória, mas também, qualitativos, como a adição de dependências externas, usabilidade e flexibilidade.

Em relação à performance, observamos que a Solução BatchLoader apresentou a melhor performance, no que diz respeito ao IPS e tempo total de execução. Em termos de memória, quando comparada às outras soluções, no cenário de maior volume de dados ela por pouco não teve o pior desempenho. Quando avaliamos uma menor exigência de dados, a situação se inverteu e ela se sobressaiu em relação as outras soluções.

No contexto qualitativo, cada solução tem seus méritos. A Solução Includes se beneficia por ser nativa do Rails, proporcionando uma integração mais suave e uma curva de aprendizado menos íngreme. Por outro lado, as Soluções GraphqlBatch e BatchLoader possuem vantagens específicas, com o primeiro sendo mais otimizado para o contexto GraphQL e o segundo oferecendo maior versatilidade.

Concluindo, apesar de todos os aspectos analisados, o *Cenário Slow*, ou seja, sem uso de nenhuma das soluções, pode ser suficiente. Se muitos dos models do banco de dados não forem associados ou as associações existentes não estiverem presentes no *Schema* GraphQL, não há necessidade de empregar nenhuma das soluções.

Em contrapartida, se queries mais frequentes do sistema sempre requisitarem essas associações, a Solução Includes pode ser a mais recomendada, já que não requer nenhuma alteração no projeto Rails. Além disso, ainda que consuma mais recursos que as outras soluções, se não houver necessidade de otimizar esse consumo, seja por excesso de recurso, seja por não haver impacto suficiente no sistema, ela também pode ser a solução mais indicada.

Se não pudermos dispor desse gasto, teremos que recorrer às Soluções GraphqlBatch e BatchLoader. No contexto em que o time é menos experiente ou apenas há APIs GraphQL na aplicação, a Solução GraphqlBatch pode ser suficiente, uma vez que exige menor entendimento inicial e somente trata do problema *N+1* no contexto do GraphQL.

Por fim, se a experiência do time for suficientemente balanceada e a aplicação estiver em contato com outros serviços em que o problema *N+1* pode acontecer, como APIs REST, a Solução BatchLoader fornece a maior flexibilidade de uso. Além disso, se o requisito de performance for extremamente necessário, essa solução também é a mais adequada.

AGRADECIMENTOS

Agradeço aos meus pais da Terra, Simone e Gilson, aos meus irmãos, Ítalo, Lucas e Pedro, e à minha terceira mãe, Fátima.

Agradeço também ao meu pai Telêmaco, que com certeza está me olhando crescer, de onde estiver.

Agradeço também aos meus amigos do peito, Pedro e Tiago, presentes em todas as minhas conquistas. Embora não compartilhemos o mesmo sangue, vocês são irmãos para mim em todos os sentidos que realmente importam.

Agradeço também à Ottony e Juliana, com quem aprendi tudo o que sei sobre Ruby on Rails durante minha jornada na Incognia.

Agradeço também à minha namorada, Camila Cunha, por todo amor, carinho e companheirismo durante esse ano tão incrível.

Agradeço também aos amigos que fiz na faculdade, por iluminarem minha jornada a cada dia.

Agradeço também ao Centro de Informática e seus professores, que me concederam tantas oportunidades durante a graduação.

Agradeço por fim, ao meu orientador Paulo Borba, pelo suporte e orientação tão necessários para o desenvolvimento deste trabalho.

REFERÊNCIAS

[1] Gleison Brito and Marco Tulio Valente. 2020. REST vs GraphQL: A Controlled Experiment. arXiv:2003.04761 [cs.SE]

[2] Roy Thomas Fielding and Richard N. Taylor. 2000. *Architectural Styles and the Design of Network-Based Software Architectures.* Ph. D. Dissertation.

[3] Roy T. Fielding and Richard N. Taylor. 2000. Principled Design of the Modern Web Architecture. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE '00).* 407–416. https://doi.org/10.1145/337180.337228

[4] Roy T. Fielding and Richard N. Taylor. 2002. Principled Design of the Modern Web Architecture. *ACM Trans. Internet Technol.* 2, 2 (may 2002), 115–150. https://doi.org/10.1145/514183.514185

[5] Facebook Inc. 2023. *GraphQL specification (draft).* Retrieved September 23, 2023 from https://spec.graphql.org/draft/

[6] Nick Keuning. 2021. Solving Complex N+1 Queries in GraphQL Ruby with BatchLoader. Retrieved September 23, 2023 from https://spin.atomicobject.com/

2021/02/22/complex-n1-queries-graphql-ruby/

[7] Piotr Roksela, Marek Konieczny, and Slawomir Zielinski. 2020. Evaluating execution strategies of GraphQL queries. In *2020 43rd International Conference on Telecommunications and Signal Processing (TSP) (LAC '10)*. 640–644. https://doi.org/10.1109/TSP49548.2020.9163501

[8] RubyGems.Org. [n. d.]. Learn how RubyGems works, and how to make your own. Retrieved September 23, 2023 from https://guides.rubygems.org/

[9] RubyOnRails.Org. 2021. Active Record Associations. Retrieved September 23, 2023 from https://guides.rubyonrails.org/association_basics.html

[10] RubyOnRails.Org. 2021. Active Record Query Interface. Retrieved September 23, 2023 from https://guides.rubyonrails.org/active_record_querying.html

[11] Dmitry Tsepelev. 2020. *How to GraphQL with Ruby, Rails, Active Record, and no N+1.* Retrieved September 23, 2023 from https://evilmartians.com/chronicles/how-to-graphql-with-ruby-rails-active-record-and-no-n-plus-one

[12] Justin Weiss. 2014. *A Guide to Choosing the Best Gems for Your Ruby Project.* Retrieved September 23, 2023 from https://www.justinweiss.com/articles/a-guide-to-choosing-the-best-gems-for-your-ruby-project/

[13] Justin Weiss. 2021. *Automatically avoiding GraphQL N+1s.* Retrieved September 23, 2023 from https://www.aha.io/engineering/articles/automatically-avoiding-graphql-n-1s