REST vs. GraphQL: Análise de desempenho com Python, PostgreSQL e Endereços Brasileiros

Patrick Silva Ferraz
Dep. de Ciência da Computação
Universidade Federal da Bahia
Salvador, Brasil
patrick.ferraz@outlook.com

Rita de Cássia Novaes Barretto, MSc
Dep. de Ciência da Computação
Universidade Federal da Bahia
Salvador, Brasil
rb.dccufba@gmail.com

Cássio Vinícius Serafim Prazeres, PhD
Dep. de Ciência da Computação
Universidade Federal da Bahia
Salvador, Brasil
cprazeres@gmail.com

***Resumo*—A busca de endereços brasileiros em mapas é de fundamental importância para diversas atividades. Permite que, ao fornecer apenas o nome do logradouro, diversos endereços sejam localizados em diferentes áreas do país. Prover funcionalidades de software por meio de RESTful APIs é uma prática comum em aplicações geoespaciais. Com o intuito de acessar endereços na base do CEP Aberto, foram desenvolvidas duas APIs, sendo uma REST e outra GraphQL. Assim, tornou-se inevitável a comparação entre elas, em termos de eficiência de desempenho, seguindo as métricas ISO/IEC 9126 e 25010. Este trabalho pretende mostrar se a API GraphQL é tão eficiente quanto a API REST, a qual vem sendo amplamente utilizada. Esta resposta pode influenciar na elaboração de melhores projetos de arquitetura de Tecnologia da Informação - TI, que venham a ser implementados com a linguagem de programação Python e o banco de dados PostgreSQL.**

***Index Terms*—rest, graphql, api, web service, python, postgresql, cep aberto**

I. Introdução

No Brasil, são muitos os problemas relativos a cadastros, especificamente à falta de completeza e confiabilidade dos dados administrativos. Nas três esferas de governo, a saber: federal, estadual e municipal, quando existem processos de negócio de registros de dados, em geral, são falhos ou não são seguidos pelos usuários como deveriam, ocasionando erros de preenchimento. Por outro lado, não existe boa interação humano-computador, de forma que a inserção dos dados nos sistemas de informação também deixa a desejar. Assim, os cadastros destes sistemas apresentam um alto percentual de endereços indevidos, ilegíveis ou simplesmente inexistentes. Em paralelo a isso, métodos para determinar uma localização em um mapa digital ou encontrar o caminho ideal para chegar até lá estão se tornando cada vez mais relevantes para a vida cotidiana e nas pesquisas acadêmicas.

Endereços existem para facilitar a localização de uma unidade imobiliária. Excetuando-se as capitais Brasília/DF e Palmas/TO, as quais são planejadas, via de regra, os endereços brasileiros consistem dos seguintes atributos: tipo de logradouro; nome do logradouro; número; complemento; bairro; localidade; município; estado e CEP. Diante da baixa qualidade dos valores destes atributos nos cadastros, a identificação do ponto exato do imóvel, torna-se muito mais difícil e onerosa. A ideia de simplificar a busca, utilizando somente o nome do logradouro como parâmetro em pesquisa numa base de CEPs colaborativa, pode facilitar a localização de diversos endereços em diferentes regiões do país.

De acordo com o contexto, este trabalho pretende saber se, para os dados administrativos com baixa qualidade, acessar um lote de endereços, utilizando-se uma *application programming interface* - API GraphQL é tão eficiente quanto o uso da API *Representational State Transfer* - REST, visando à avaliação da latência e *overfechting*?

A hipótese é que a API GraphQL será mais eficiente com relação ao *overfechting* e possivelmente à latência, uma vez que o acesso retornará um lote de endereços, em vez de um único.

A. Escopo e estrutura do artigo

O escopo deste estudo consiste do levantamento de requisitos, desenvolvimento e teste de uma API REST, baseada em HTTP, e outra API GraphQL, para acesso aos dados do projeto CEP Aberto [1]. Uma comparação da eficiência de desempenho de ambas, seguindo as métricas ISO 9126 e ISO 25010, faz-se necessária para subsidiar a tomada de decisão quanto a projetos arquiteturais que venham a ser implementados com a linguagem de programação Python e o banco de dados PostgreSQL, bem como envolvam acesso a endereços em bases de dados georreferenciados.

O conteúdo do artigo está organizado em cinco seções. A seção I é a introdução, que destaca o escopo e a estrutura do artigo. A seção II é o referencial teórico, ressaltando API, REST, GraphQL, normas e padrões da ISO/IEC, assim como as métricas para comparação. A seção III apresenta as arquiteturas desenhadas. A seção IV demonstra a análise de desempenho, detalhando o pré-processamento, a aplicação e comparação dos métodos, assim como os resultados. A Seção V resume as discussões e aponta desdobramentos futuros.

II. Referencial teórico

A. API

Durante anos, diversas alternativas de software surgiram para facilitar a comunicação entre aplicações ou entre um usuário e uma aplicação. À estas alternativas, foi dado o nome de Application Programming Interface - API.

Em verdade, uma API trata-se de rotinas desenvolvidas e padrões estabelecidos e documentados por uma aplicação, para que usuários ou outras aplicações utilizem as funcionalidades daquela, sem precisar conhecer detalhes do software. Assim, as APIs implementam interoperabilidade entre aplicações.

Um exemplo é o Google Maps [2] que é um Geographical Information System - GIS baseado em nuvem e que combina mapas, imagens de satélite e informações geoespaciais, provendo APIs para que usuários ou outras aplicações possam ter acesso ao seu conteúdo. [3]

B. REST

De acordo com [4], fornecer funcionalidade de software por meio de APIs é uma prática comum em aplicativos geoespaciais. Por meio de um serviço da Web leve e escalável, clientes podem acessar ou processar dados geoespaciais sem complicação. Entre os provedores de API de geocodificação atuais, estão: a API de geocodificação do ArcGIS [5]; a API de geocodificação aberta do MapQuest [6] e a API de geocodificação what3words - w3w [7].

Neste trabalho, para implementar a busca de lotes de endereços foi projetada uma API para usar dados da plataforma CEP Aberto [1]. *Representational State Transfer* - REST é um estilo arquitetônico de rede que é aplicado a projetos de API para serviços da Web [8]. Uma API em conformidade com a mencionada arquitetura é considerada RESTful. Um cliente pode obter dados do serviço da Web, enviando uma solicitação para um Uniform Resource Locator - URL específico por meio de uma API REST, que utiliza o Hyper Text Transfer Protocol - HTTP. Segundo [9], este tipo de API apresenta problemas, principalmente com o surgimento de aplicativos para dispositivos móveis, pelo fato das consultas REST exigirem muitos HTTP *endpoints*, os quais retornam estruturas de dados fixas, podendo resultar em excesso de dados.

C. GraphQL

Diante desse cenário que apontava para a obsolescência da RESTful API, em 2012, o Facebook construiu a GraphQL para seus aplicativos Web e móveis. Em 2015, tornou-a pública. As necessidades de mais flexibilidade e eficiência na comunicação entre cliente e servidor foram decisivas para o planejamento e execução do projeto. De acordo com [9], a GraphQL possui um *endpoint* "inteligente"que pode executar operações complexas e retornar apenas os dados que o cliente precisa. A forma da resposta corresponde à forma da consulta. Portanto, a GraphQL minimiza a quantidade de dados transferidos e permite uma fácil evolução das aplicações.

D. ISO/IEC 9126 e 25010

Visando à alta qualidade, a comparação das mencionadas APIs baseou-se nos padrões internacionais das entidades: International Organization for Standardization – ISO e International Electrotechnical Commission – IEC. Especificamente, nas métricas ditadas pelas ISO/IEC 9126 [10] e ISO/IEC 25010 [11], as quais são voltadas para qualidade de software.

De acordo com [10], as métricas de eficiência listadas na ISO/IEC TR 9126-3: 2003 não pretendem ser um conjunto exaustivo. Avaliadores podem selecioná-las para avaliar produtos de software, medir aspectos de qualidade e outros propósitos. Uma vez que, a ISO / IEC TR 9126-3: 2003 não atribui intervalos de valores dessas métricas a níveis nominais ou a graus de conformidade, eles foram definidos na Subseção II-E, para atender a essa comparação entre APIs.

E. Métricas para comparação

É possível avaliar um sistema computacional com a utilização de diversos parâmetros, que tem como propósito computar o uso de recursos como a memória, o processamento e até a taxa transmissão em rede. Alguns destes estão relacionados com o cálculo da latência e *overfetching*, respectivamente, o tempo que leva uma chamada atômica na API e o fato do cliente baixar mais informações do que é realmente necessário no aplicativo.

Segundo [12], o que evidencia a superioridade da GraphQL em relação a aplicações com a arquitetura REST, refere-se principalmente ao *overfetching*. Isto é o que este trabalho busca comprovar.

Em sistemas REST, *endpoints* são utilizados para obter uma certa informação da API. Eles possuem o corpo de retorno padrão, muitas vezes sendo a estrutura completa da entidade do banco de dados. GraphQL pode ou não possuir uma estrutura semelhante, a grande diferença está na flexibilidade em relação ao que se quer obter, ou seja, é possível escolher os campos específicos que se pretende obter da entidade, evitando assim o *overfetching*. [12] afirma que, com a utilização do GraphQL "Você pede o que quer e obtém resultados previsíveis".

Por fim, é possível utilizar os resultados obtidos para avaliar a latência. Embora a API GraphQL aparente retornar o resultado com atraso em relação à API REST, o corpo do conteúdo da GraphQL pode ser menor. Assim, pacotes menores serão transmitidos na rede, diminuindo a latência.

Abaixo, segue uma descrição das métricas adaptadas de [13]:

1) Categoria - Comportamento em relação ao tempo.
   1) Nome da Métrica - Latência:
   a) Propósito da Métrica - Determinar o tempo para realizar uma chamada atômica na API;
   b) Método da Métrica - Avaliar a velocidade de resposta a chamadas para backend para operações atômicas. Será medido o tempo completo desde que se realize a chamada a API até que se retorne a resposta desejada;
   c) Fórmula:
   $$x = tempo\ calculado \tag{1}$$
   d) Interpretação - Quanto menor o valor obtido, melhor;
   e) Tipo de Medida: $x = ms$;
   f) Origem da Medida - Tempo esperado de uma chamada atômica para o backend.
2) Categoria - Utilização de recursos.

2) Nome da Métrica - Overfechting:

a) Propósito da Métrica - Determinar a porcentagem de informações supérfluas, em média, num conjunto de chamadas;
b) Método da Métrica - Estimar a eficiência das respostas recebidas por meio da API;
c) Fórmula:

$$x = total\ de\ bytes - bytes\ usados \quad (2)$$

d) Interpretação - Quanto menor o valor, melhor;
e) Tipo de Medida: $x = bytes$;
f) Origem da Medida - Número de bytes retornados pela API e número de bytes que foram usados.

III. Arquitetura

Dois modelos arquiteturais foram elaborados para o desenvolvimento das APIs (REST e GraphQL). O padrão Model View Controller - MVC foi adotado, visando à criação de certa semelhança entre as arquiteturas.

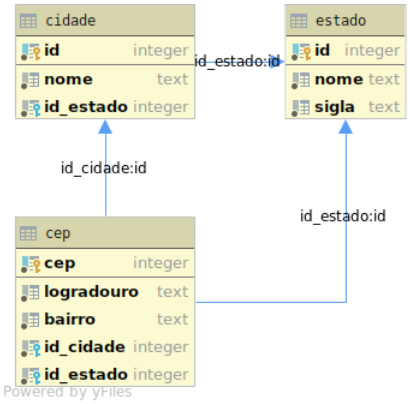

Figura 1. Diagrama de entidade e relacionamento do banco de dados.

As arquiteturas contam com três entidades: *state* (estado), *city* (cidade), *place* (logradouro), modelados conforme a Figura 1, onde *state* e *city* possuem uma *id* como chave primária e campos referentes aos respectivos nomes, siglas e relações entre si, enquanto que *place* possui o CEP como chave primária, *ids* para referenciar o estado e a cidade ao qual pertence, nome do bairro e nome do logradouro.

Na arquitetura REST um chamada para API é realizada através do método *GET* e dos seguintes *endpoints*:

- **/state**
  Retorna todos estados brasileiros.
- **/state/*:state***
  Retorna todos estados brasileiros que combinem com *:state*.
- **/state/*:state*/city**
  Retorna todas cidade de *:state*.
- **/state/*:state*/city/*:city***
  Retorna todas cidade de *:state* que combinem *:city*.

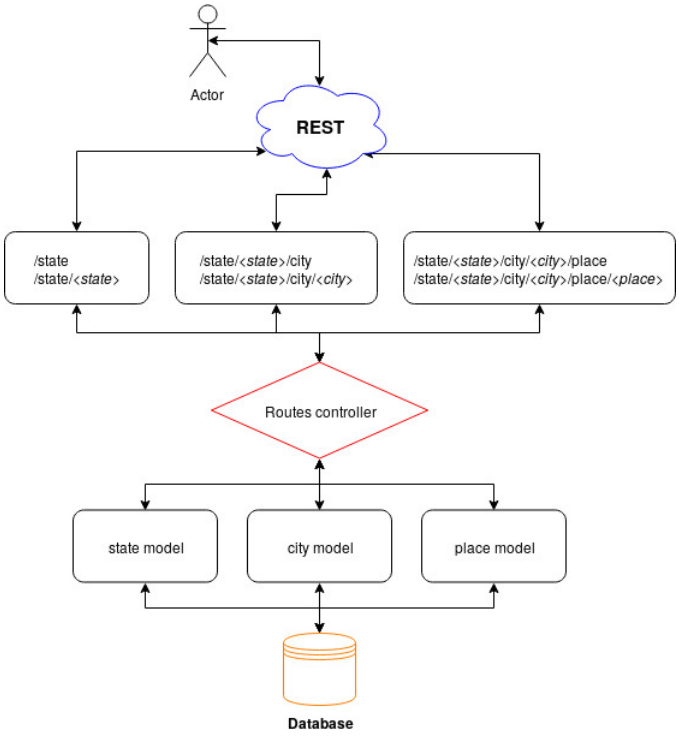

Figura 2. Arquitetura da API REST.

- **/state/*:state*/city/*:city*/place**
  Retorna todos logradouros da cidade *:city* pertencentes ao estado *:state*.
- **/state/*:state*/city/*:city*/place/*:place***
  Retorna todos logradouros que combinem com *:place*, da cidade *:city* pertencente ao estado *:state*.

Os campos com prefixo ':' indica onde o texto da requisição é inserido para buscar por semelhança, sendo assim, para consultar todas as cidades do estado da Bahia poderia ser utilizado */state/**Bahia**/city* ou */state/**ba**/city* para retornar todas cidades (*city*) de estados (*state*) que possuem a subcadeia **ba** em seu nome. Ressalta-se também que a busca não utiliza *case sensitive*, ou seja, não faz diferença o uso de **Bahia** ou **bahia**. A mesma situação acontece para busca de nome de cidades. O caso especial está na busca por *place*, na qual a combinação é comparada com CEP, bairro e logradouro da tabela.

Com todos conceitos informados anteriormente, torna-se simples entender o fluxo apresentado na Figura 2. Uma solicitação por *endpoint* é feita na API REST, a mesma aciona o controlador de rotas (*Routes controller*) que indica qual ação deve ser tomada, utilizando o *model* específico para a resposta. O *model*, por sua vez, manipula o banco de dados (*database*), reagindo à ação, e retorna o resultado para o caminho inverso com a resposta da solicitação.

A estrutura e comportamento da aplicação GraphQL é semelhante à REST. A diferença pode ser observada nos *endpoints* e *models*. Na GraphQL, um único *endpoint* é utilizado com requisição por método *POST* e, seus *models* são representados como *schemas*. Outro aspecto, refere-se à adição de mais uma

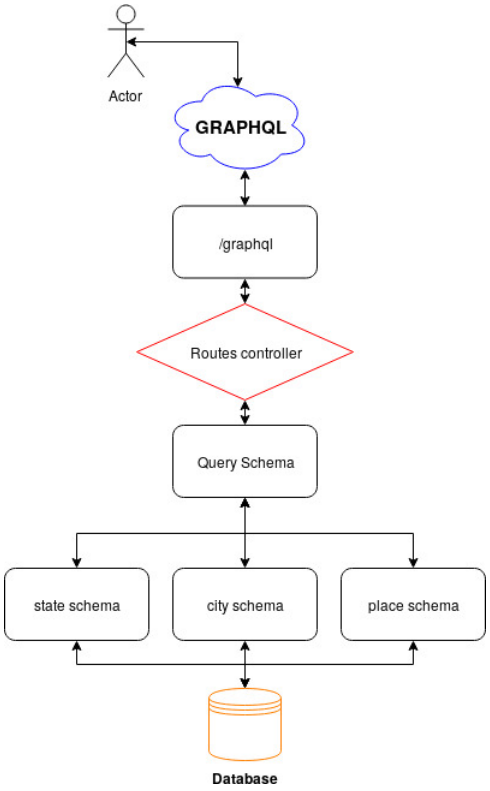

Figura 3. Arquitetura da API GraphQL.

camada de controle (*Query Schema*) que é responsável por intermediar a comunicação do *controller* com os *schemas*. A Figura 4 exemplifica o *endpoint* e corpo de uma requisição para a REST e a GraphQL.

GET /state/ba

content-type: application/json
payload: {}

POST /graphql

```
content-type: application/json
payload: {
  query {
    getState(name:"ba") {
      id
      name
      acronym
    }
  }
}
```

Figura 4. Corpo da requisição com REST (esquerda) e GraphQL (direita).

De forma resumida, fica perceptível que toda arquitetura foi desenvolvida para fornecer variações de consultas de endereços seguindo um modelo em árvore, no qual a raiz seria os estados (Figura 5).

Caso haja necessidade de realizar a requisição sem nenhum filtro, basta fornecer o valor '*%*' para o campo de busca, que possibilita o retorno de todos os registros. Devido à limitações físicas, por padrão, existe um tamanho fixo de 10 (dez) registros retornados por requisição, a fim de evitar insucessos durante as consultas. Podem ser utilizados para modificar tal comportamento, o parâmetro *limit* (padrão 10) e também o *page* (padrão 0), respectivamente, o que indica

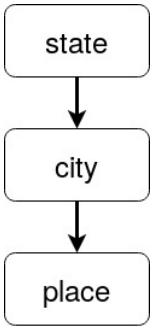

Figura 5. Árvore de requisição.

a quantidade de registros máximos por página e o que indica qual página deseja-se obter. A Figura 6 exemplifica a alteração dos parâmetros.

GET

/state/ba?limit=50&page=5

content-type: application/json
payload: {}

POST /graphql

```
content-type: application/json
payload: {
  query {
    getState(name:"ba", limit:50, page:5) {
      id
      name
      acronym
    }
  }
}
```

Figura 6. Corpo da requisição com REST (esquerda) e GraphQL (direita), parametrizando o tamanho limite da resposta.

```
{
    "estado": {
        "id": "INT NOT NULL PRIMARY KEY,",
        "nome": "TEXT NOT NULL,",
        "sigla": "TEXT NOT NULL"
    },
    "cidade": {
        "id": "INT NOT NULL PRIMARY KEY,",
        "nome": "TEXT NOT NULL,",
        "id_estado": "INT NOT NULL REFERENCES estado(id)"
    },
    "cep": {
        "cep": "INT NOT NULL PRIMARY KEY,",
        "logradouro": "TEXT NOT NULL,",
        "bairro": "TEXT NOT NULL,",
        "id_cidade": "INT NOT NULL REFERENCES cidade(id),",
        "id_estado": "INT NOT NULL REFERENCES estado(id)"
    }
}
```

Figura 7. Arquivo JSON utilizado para modelagem física.

O script de pré-processamento foi responsável pela modelagem física do banco de dados, consultando como será a estrutura através de um arquivo JSON com o *key-value pair* = {*atributo: descrição*} (Figura 7) e como popular as tabelas modeladas, seguindo os passos:

1) *Download* da base de dados do CEP Aberto em formato *.zip*.
2) Descompactação em uma única pasta de pré-processamento.
3) Criação da modelagem física no postgres utilizando o arquivo JSON para referência.
4) Leitura dos arquivos ".txt"descompactados

5) Limpeza das strings adicionando *scapes*, como exemplo o das aspas.
6) Inserção das tuplas nas respectivas tabelas.

A Figura 8 retrata de forma resumida o pré-processamento.

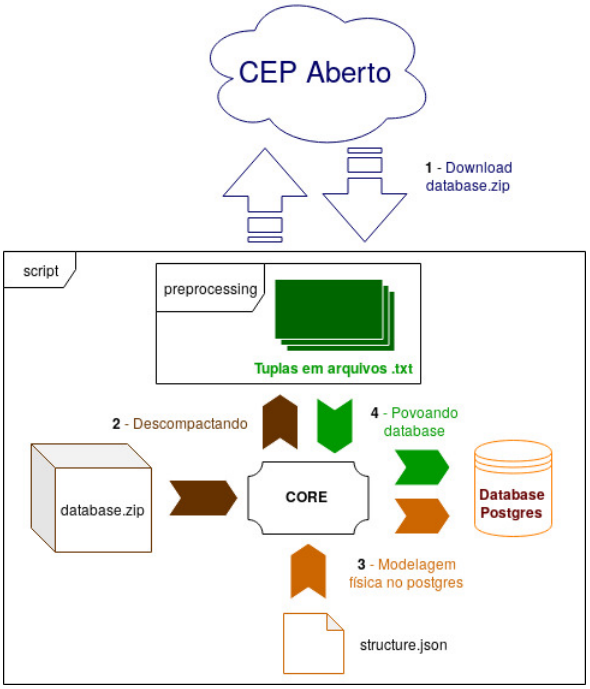

Figura 8. Fluxo do pré-processamento de dados.

IV. Análise de desempenho

A. Pré-processamento

Inicialmente, foi feita uma tentativa de utilização da base de dados geográficos do OpenStreetMap - OSM [14]. A iniciativa deste projeto de mapeamento colaborativo nasceu, visando à criação de um mapa livre e editável do mundo, inspirado por sites como a Wikipédia. Embora seja amplamente divulgado e forneça dados a muitos sites na internet, a aplicações de celular e outros dispositivos, para este trabalho a experiência não foi bem sucedida, em virtude das muitas dificuldades enfrentadas na etapa de pré-processamento dos dados extraídos do OSM.

Importante ressaltar que a maioria das bases geográficas são proprietárias, a exemplo das bases de dados de navegação da Here Technologies [15] e da TomTom [16]. Por serem serviços na nuvem disponibilizados por empresas privadas, são atualizadas constantemente e mais completas que as bases livres, mas obviamente são comercializadas. Por isso, não foram consideradas neste trabalho.

Depois de uma extensa pesquisa, optou-se por utilizar os dados do CEP Aberto [1], que é um projeto que visa prover acesso gratuito e construir, de maneira colaborativa, uma base de dados com os Códigos de Endereçamento Postal - CEP geolocalizados de todo o Brasil. A ideia é que cada indivíduo possa colaborar, melhorando a informação de CEPs que conhece. Mediante registro, qualquer pessoa, gratuitamente, pode ter acesso a API e baixar a base de dados. O CEP Aberto [1] possui informação de 980.955 CEPs distribuídos em 10.560 municípios brasileiros.

Extrair os dados corretos da base da plataforma CEP Aberto [1] foi fundamental. O seu pré-processamento e sua estruturação foram extremamente importantes para a qualidade das informações e resultados do trabalho.

O CEP Aberto [1] disponibiliza todos os endereços brasileiros em arquivos em formato ".txt"separados por estado, cidade e cep, sendo o último particionado em cinco arquivos diferentes. Foram 77 arquivos compactados que totalizaram mais de 600 mil endereços brasileiros.

B. Comparação dos Métodos

O banco dados foi gerado e executado, utilizando uma instância do PostgreSQL [17] no Docker [18] em um notebook SIM 5110M Core i3-3110M 2.4GHz com 6GB de memória RAM e SSD de 240GB KINGSTON SUV300S com sistema operacional Pop!_OS 19.04.

As APIs foram desenvolvidas utilizando a linguagem de programação Python [19] e o Framework Flask [20].

Todos os testes de análise comparativa foram realizados localmente e de forma isolada. Primeiramente, utilizando a API REST e posteriormente a GraphQL, ambos os testes com a mesma instância do PostgreSQL [17].

Conforme mencionado na Subseção II-E, a avaliação foi realizada comparando o *overfetching* e a latência. As requisições foram feitas utilizando os três níveis da árvore de requisição (Figura 5), sendo o nível 0, consultas em estados; o nível 1, incluindo cidades e o nível 2, acrescentando os logradouros.

Em cada nível, houve um tamanho de bloco a ser requisitado, variando do menor para o maior, ou seja, no nível 0 (estados), 1 (cidades) e 2 (logradouros) foi possível obter os tamanhos de blocos:

- em 0: (id), (id, name), (id, name, acronym);
- em 1: (id), (id, idState), (id, idState, name);
- em 2 (cep), (cep, idState), (cep, idState, idCity), (cep, idState, idCity, ditrict), (cep, idState, idCity, district, publicPlace);
- e também variações de tamanhos mesclando os três níveis, ou seja, utilizado apenas "*name*"do nível 0 e 1, e o "*district*"(bairro) do nível 2.

As requisições foram realizadas utilizando os diversos tamanhos de blocos possíveis e a análise ponderou a quantidade de dados úteis e totais retornados pela API GraphQL em comparação a REST. O tempo foi calculado utilizando a biblioteca *timeit* do Python [19], sendo destacado pela acurácia. Para cada tamanho de bloco possível foi realizado 100 requisições para o cálculo do tempo médio, e os dados úteis e totais retornados foram mensurados utilizando as próprias informações do *header* (cabeçalho).

C. Resultados

Os resultados apresentados a seguir foram obtidos com três tamanhos de blocos possíveis (mínimo, médio e máximo) para cada escopo, sendo os três escopos:

- Local:
  - Mínimo: Obtendo apenas os nomes dos estados.
  - Médio: Obtendo os nomes e siglas dos estados.
  - Máximo: Obtendo todas informações dos estados.
- Fronteira:
  - Mínimo: Obtendo apenas os nomes dos estados e nomes das respectivas cidades.
  - Médio: Obtendo os nomes dos estados e os nomes e *ids* das respectivas cidades.
  - Máximo: Obtendo todas informações dos estados e suas cidades.
- Global:
  - Mínimo: Obtendo as siglas dos estados, nomes das cidades e os CEPs dos logradouros.
  - Mínimo: Obtendo todas informações dos estados, nomes e *ids* das cidades e os CEPs e barrios dos logradouros.
  - Máximo: Obtendo todas informações dos logradouros com suas cidades e estados.

Para todos os escopo foram definidos os parâmetros de pesquisa com *page* 0 e *limit* de 20.000 registros.

Devido a diversa quantidade de variações do tamanho do bloco, apenas os mecionados anteriormente foram adotados, sendo escolhido dessa forma para melhor distribuição e diferença em relação aos tamanhos e possíveis variações de tempo que poderiam ocorrer.

As requisições forem feitas utilizando um *script* python automatizado com o pacote *requests*. Os tempos foram cálculados considerando apenas a latência das requisições, ou seja, a ”chamada”de função GET, em REST, e POST, em GraphQL, do pacote *requests*. Os resultados do script foram armazenados em um arquivo *json* contendo uma lista de ”dicionários”com o conjunto de dados em formato *key-value pair*, descritos da seguinte forma:

- ”scope”: O escopo que foi utilizado para requisição (local, border (fronteira), global)
- ”blockSize”: o tamanho do bloco definido para requisição (min, med, max).
- ”size”: o tamanho efetivo do bloco retornado.
- ”time”: um array com os 100 períodos de tempo obtido com o temporizador da requisição.

A Figura 9 exemplifica o corpo do arquivo de resultados.

```
[
    {
        "scope": "local | border | global",
        "blockSize": "min | med | max",
        "size": "tamanho do bloco retornado",
        "time": [tempos das execuções]
    },
]
```

Figura 9. Estrutura do arquivo de resultados.

O cáculo do tamanho de bloco total e útil foi realizado considerando as resposta das requisições GraphQL como sendo sempre de ”tamanho útil”, e com esse resultado é obtido a proporção de overfetching que o REST possui superior ao GraphQL através da diferença entre ambos. Analisando os resultados obtidos na Tabela I, observa-se que o comportamento do REST possui maior overfetching quando são requisitados menores conjuntos de dados das APIs independente do escopo. Ressalta-se também tal comportamento no escopo global, em a requisição solicita todas entidades disponíveis (estados, cidades, cep), havendo uma redução razoável a medida que aprofundamos a árvore de requisição (Figura 5).

Tabela I
PROPORÇÃO DE OVERFETCHING EM REST SUPERIOR AO GRAPHQL.

Escopo	Tipo	Resposta da arquitetura (bytes)		% (≈)
		REST	GraphQL	
local	min	1966	365	81.43
	med	1966	590	69.99
	max	1966	705	64.14
fronteira	min	1092575	7213	99.34
	med	1092575	9789	99.10
	max	1092575	13466	98.77
global	min	4232448	347057	91.8
	med	4232448	906716	78.58
	max	4232448	2274673	46.26

Devido ao ambiente não controlado e compartilhado, algumas discrepâncias de latência surgiram. A Figura 10 exibe o pior caso de outlier encontrado na execução da API REST (escopo de fronteira). Enquanto que na Figura 11 apresenta o pior caso para a arquitetura GraphQL onde o tamanho de bloco foi do tipo máximo.

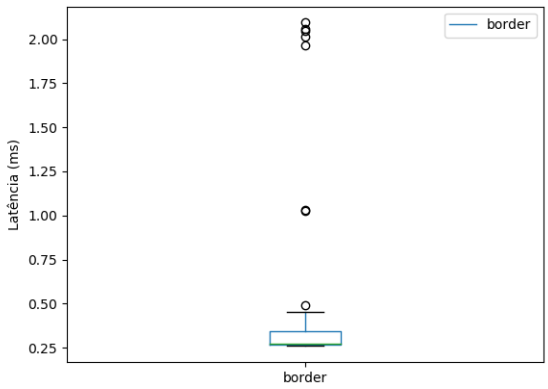

Figura 10. Outliers do escopo de fronteira para API REST.

Ressalta-se que em todos tamanhos de blocos: mínimo, médio e máximo, a API REST gerou resultados semelhantes.

Os outliers foram tratados utilizando a regra 1,5xIQR (Interquartile range), que informa de que modo os valores médios estão espalhados, utilizando a Equação 3, onde q_1 é o primeiro quartíl e q_3 é o terceiro quartíl.

$$iqr = q_3 - q_1 \tag{3}$$

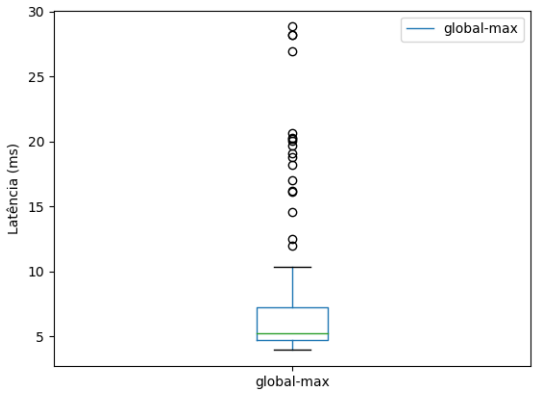

Figura 11. Outliers do escopo global e tamanho de bloco do tipo máximo para API GraphQL.

Tabela II
PROPORÇÃO DE OVERFETCHING EM REST SUPERIOR AO GRAPHQL.

Escopo	Tipo	Tempo médio da arquitetura (ms)	
		REST	GraphQL
local	min	0.00699157929033343	2.774390276240301
	med	0.00699157929033343	2.844395469267839
	max	0.00699157929033343	2.791358945252202
fronteira	min	0.30514050403318155	2.812832926094339
	med	0.30514050403318155	2.760971944828525
	max	0.30514050403318155	2.6593246800774875
global	min	3.2016676112147078	3.726284877153895
	med	3.2016676112147078	5.119934367011191
	max	3.2016676112147078	5.351672762182805

Desse modo, foi possível obter os resultados de latência conforme apresentados na Tabela II. É observado que ambas arquiteturas possuem a latência diretamente proporcional a complexidade do escopo. Alguns comportamentos podem ser destacados, como: a similaridade entre as latências de escopo local e de fronteira no GraphQL, possivelmente devido ao corpo da requisição serem equivalentes; e a semelhança entre os resultados obtidos em GraphQL para o mesmo tipo de tamanho (min, med, max), que talvez esteja interligado a estabilidade para acesso ao banco de dados.

V. DISCUSSÃO

Este trabalho investigativo projetou, desenvolveu e testou uma API REST e outra API GraphQL, com o intuito de comparar o desempenho de ambas, no tocante à eficiência, especificamente à categoria comportamento em relação ao tempo, latência, e à categoria utilização de recursos, *overfetching*.

Os resultados da pesquisa têm implicações significativas para a tomada de decisão com relação à arquiteturas de TI que venham a ser implementadas com a linguagem de programação Python e o banco de dados PostgreSQL, bem como acessem endereços em bases de dados georreferenciados.

Diante do exposto, conclui-se que:

- Uma implementação confiável do estilo de arquitetura REST em uma API apresentou uma resposta mais rápida do que uma API criada com a tecnologia GraphQL.
- Os tamanhos das respostas de uma API GraphQL são consideravelmente menores do que os tamanhos das respostas geradas por uma API que segue as restrições do estilo da arquitetura REST.
- Existe algum comportamento interessante em relação a estabilidade da latência no GraphQL, sendo seus resultados muito próximos.
- Curiosamente, um caso foi identificado quanto ao *overfetching* em REST nos escopos com tamanho de tipo máximo. Embora as requisições para ambas as arquiteturas estejam solicitando o maior conjunto de dados disponíveis, o REST ainda apresenta, no melhor caso, um *overfetching* de 46.26% superior ao GraphQL, comportamento não identificado mas talvez ligado à duplicidade de informações no corpo da resposta em REST.

Alguns desafios e desvantagens a serem considerados nesta abordagem:

- Poderiam ser aplicados testes com variações no tamanho do *limit*. Isso aumentaria o conjunto de endereços, englobando não somente o Brasil. Desse modo, seria possível expandir o campo de busca. Embora o *limit* tenha sido definido em 20.000 endereços, a quantidade de estados brasileiros, por exemplo, possuem bem menos que isso.
- Necessidade de executar os testes realizados, em ambiente controlado. Na execução local, existe um constante compartilhamento de recursos e muitas vezes processos que são alocados e desconhecidos. Um ambiente controlado seria ideal para realizar o cálculo da latência, por exemplo.
- Ressalta-se que, mesmo tendo sido aplicada uma função para reduzir os outliers, ainda pode haver muito ruído nos dados de latência, pelos mesmos motivos destacados no item anterior.

A partir deste estudo, trabalhos futuros poderão explorar outros tipos de testes de software. Por exemplo, testes de carga e stress, para comparar como ambas as arquiteturas reagem à requisições constantes sem ociosidade de tempo. Simultaneamente, verificar o uso dos recursos computacionais que cada uma utiliza para dar uma resposta.

REFERÊNCIAS

[1] CEP Aberto. CEP Aberto - Códigos de Endereçamento Postal (CEP) geolocalizados de todo o Brasil. [Online]. Available: https://www.cepaberto.com

[2] Google. Google Maps Platform - Places. [Online]. Available: https://cloud.google.com/maps-platform/places/?hl=pt-br

[3] R. Nourjou and J. Thomas, "System architecture of cloud-based web gis for real-time macroeconomic loss estimation," in *Proceedings of the 5th ACM SIGSPATIAL International Workshop on Mobile Geographic Information Systems*, ser. MobiGIS '16. New York, NY, USA: ACM, 2016, pp. 56–63. [Online]. Available: http://doi.acm.org/10.1145/3004725.3004731

[4] W. Jiang and E. Stefanakis, “A restful api for the extended what3words encoding,” *ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. IV-4, pp. 97–104, 2018. [Online]. Available: https://www.isprs-ann-photogramm-remote-sens-spatial-inf-sci.net/IV-4/97/2018/
[5] ESRI. ArcGIS REST API. [Online]. Available: https://developers.arcgis.com/rest/
[6] MapQuest. MapQuest REST API. [Online]. Available: https://www.api-rest.com/apis/api-463-mapquest-technology
[7] w3w. what3words API. [Online]. Available: https://www.api-rest.com/apis/api-463-mapquest-technology
[8] M. Masse, *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. ”O’Reilly Media, Inc.”, 2011.
[9] A. Ritsilä, “Graphql: The api design revolution,” 2018.
[10] International Organization for Standardization. (2003) ISO/IEC TR 9126-3:2003. [Online]. Available: https://www.iso.org/standard/22891.html
[11] ——. (2011) ISO/IEC 25010:2011. [Online]. Available: https://www.iso.org/standard/35733.html
[12] GraphQL. GraphQL is the better REST. [Online]. Available: https://www.howtographql.com/basics/1-graphql-is-the-better-rest/
[13] C. Guillen-Drija, R. Quintero, and A. Kleiman, “Graphql vs rest: una comparación desde la perspectiva de eficiencia de desempeño.” 11 2018.
[14] OSM. OpenStreetMap. [Online]. Available: https://www.openstreetmap.org/about
[15] Here Technologies. Location Services - Places. [Online]. Available: https://www.here.com/products/location-based-services/poi-tools
[16] TomTom. Maps - Search API. [Online]. Available: https://maps.tomtom.com/onlinesearch/
[17] PostgreSQL. PostgreSQL:The World’s Most Advanced Open Source Relational Database. [Online]. Available: https://www.postgresql.org/
[18] DockerHub. Postgres. [Online]. Available: https://hub.docker.com/_/postgres
[19] Python. Python. [Online]. Available: https://www.python.org/
[20] Flask. Flask web development, one drop at a time. [Online]. Available: http://flask.pocoo.org/