

RELATÓRIO FINAL

PROJETO SEMESTRAL - BANCO DE DADOS II

Geovany Carlos Mendes - 2018013746

Leonardo Vieira Ferreira - 2018009799

Thiago Silva Pereira - 2018008209

1. Introdução

Este é o relatório para o projeto final de banco de dados II, coordenado pela docente Vanessa Cristina Oliveira de Souza. Serão apresentados todos os resultados obtidos na construção da aplicação para consumo da API e dos testes com JMeter.

A API selecionada para o trabalho foi a do GitHub, que comporta informações sobre sua plataforma como: usuários, repositórios, licenças, Informes entre outros. A partir dele elaboramos o modelo relacional apresentado em 2.1 e todas as definições dispostas na metodologia. O modelo foi pensado para melhor dispor informações que podem ser consideradas úteis e essenciais para gerar um relatório sobre o tema, de maneira que a aplicação construída pudesse ter alguma aplicação prática.

A aplicação foi elaborada para utilização principalmente por empresas de TI. Durante o período de recrutamento é possível buscar por perfis do github que atendam as demandas da empresa, bem como aqueles que apresentam melhores rendimentos ou maior produtividade nos seus projetos. Funcionalidades futuras podem incluir integração com outros repositórios remotos, como o gitlab. Isso permitiria não só contemplar um número maior de usuários, mas também ser usado para obter relatórios dos funcionários que já atuam na empresa.

O relatório é disposto em:

- 2.0 Seção que realiza a apresentação da metodologia usada na construção da aplicação e do relatório Ad-Hoc, mencionando as tecnologias usadas, esquemas e modelos;
- 3.0 Seção onde se encontram baterias de testes para verificação das capacidades máximas de usuários e variação de latência nos bancos.
- 4.0 Seção que mostra como o relatório ad-hoc funciona.
- E por último, 5.0 é a Seção de conclusão do artigo, com os principais ganhos e dificuldades do grupo na execução do trabalho.

2. Metodologia

Tanto a aplicação para o consumo da API quanto para a construção do relatório Ad-Hoc foram feitas em javascript. O consumo da API foi feito através de scripts que realizavam chamadas através de um cliente http (Axios), tratavam os dados e persistiam os mesmos no banco de dados através de modelos, gerados pelo Sequelize no caso do PostgreSQL e Mongoose para o mongoDB. A aplicação, além de consumir os dados do Github, os fornece de acordo com a solicitação informada no front-end , através da construção de um objeto que determina os atributos para consulta ao ORM.

Usando de uma aplicação com essas propriedades, é possível mandarmos os parâmetros selecionados na geração do relatório, direto como um pedido para Back-end, que utiliza das funções do ORM selecionado para as operações com o banco.

As tecnologias e ferramentas utilizadas são listadas a seguir:

- ◆ **Sequelize:** Um ORM de JavaScript para realizar o mapeamento entre as tabelas do banco e os dados capturados;
- ◆ **Axios:** Uma biblioteca para realizar as requisições HTTP na API do Github;
- ◆ **Mongoose:** Um modelador de objetos para determinar as estruturas que seriam enviadas ao mongo e ter controle sobre o formato dos documentos;
- ◆ **Insomnia:** Um cliente HTTP para testar as requisições das aplicações criadas;
- ◆ **MongoDB compass:** Uma GUI (Graphical User Interface) para gerenciamento do Mongo.
- ◆ **MongoDB Atlas:** Um gerenciador para o MongoDB na nuvem.

Para a primeira aplicação, que realiza o consumo dos dados da API, foram gerados dois scripts para cada entidade importante no banco (Issues, Licenses, Repos e Users), uma para persistir os dados no Mongoddb e outro no Postgre.

2.1 Modelo Relacional

Com a aplicação caracterizada, o primeiro passo do projeto foi a construção de um modelo relacional, essencial para uma boa construção de um banco relacional como o Postgre, esse mesmo modelo também é utilizado como auxílio na construção do modelo orientado a documentos utilizado no mongoDB, porém agora com maior

liberdade em voltar a construção ao documentos que serão utilizados pela API. Na Figura 1 é possível visualizar o modelo.

O modelo relacional foi construído em volta dos repositórios e usuários como informações principais, já que essas duas tabelas representam a base do GitHub, a partir desse meio foi desenvolvida ligação com outras informações que complementam a aplicação, como os informes de cada repositório ou as licenças utilizadas. É interessante observar que o informe se comporta como uma entidade fraca de repositório, sendo seu id e escopo dependente do repositório. Outra peculiaridade que torna o modelo mais completo é a presença de atributos multivalorados, que podem vir a gerar novas tabelas em bancos relacionais, mas que podem ser tratados em um modelo de documentos.

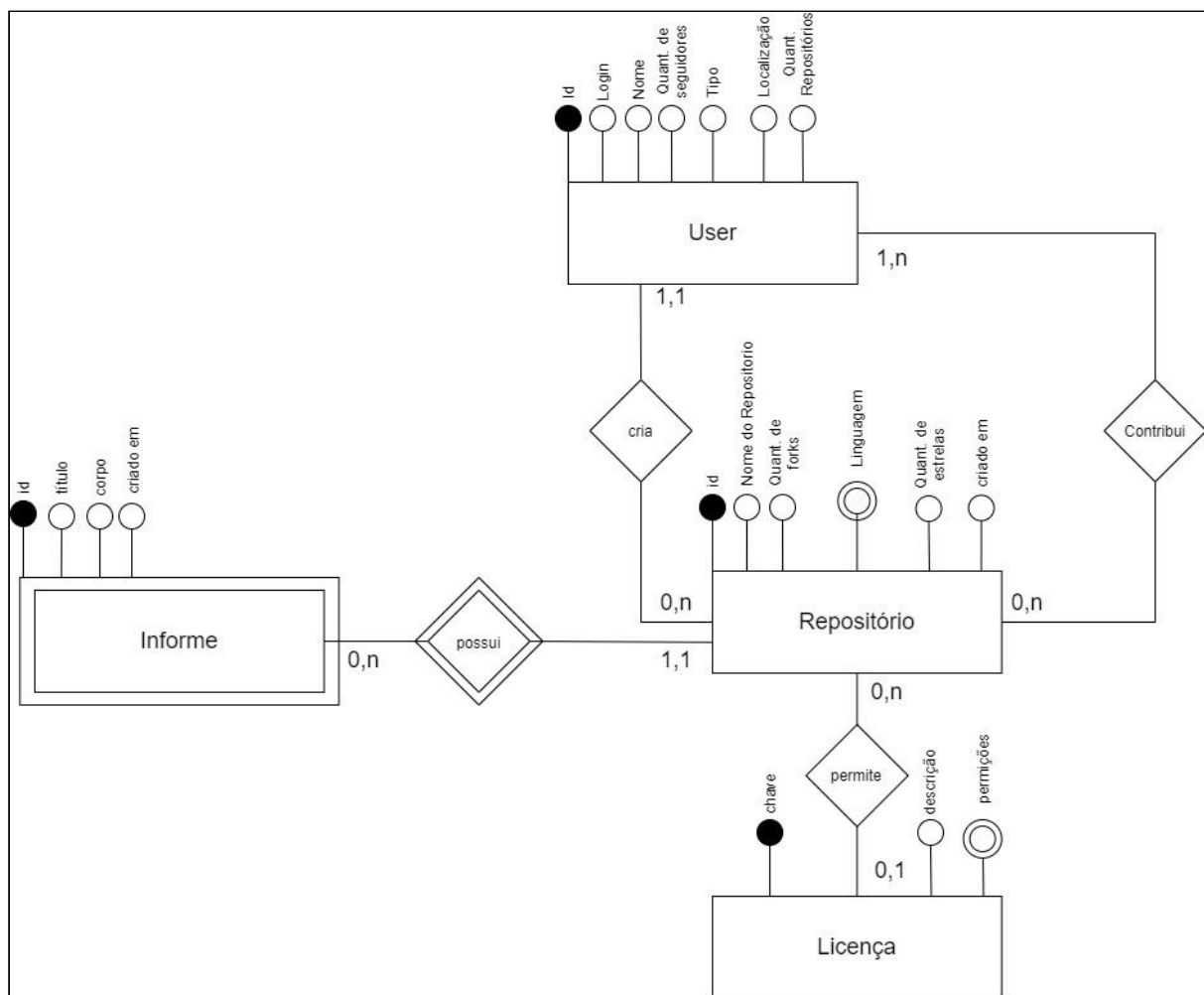


Figura 01: Modelo Relacional desenvolvido para consumo da API

2.2 Tabelas e Coleções

Com a criação do modelo relacional, agora é possível criar o modelo lógico e as tabelas no Postgre, as quais são apresentadas com a contagem dos dados colhidos na Figura 2.

total_issues bigint	1	3987
total_licenses bigint	1	13
total_permissions bigint	1	57
total_repositories bigint	1	33623
total_users bigint	1	1000
total_users_repos bigint	1	20465

Figura 02: Count das tabelas do utilizadas e populadas no Postgre

Para o banco no Mongo, o modelo orientado a documentos gerado pode ser visualizado nas Figura 3 a Figura 7, e na Figura 8 a contagem da população das coleções.

```
"Issue": {
  "title": "String",
  "body": "String",
  "created_at": "Date"
}
```

Figura 03: Modelo JSON tabela Issue

```
"License": {
  "key": "String",
  "description": "String",
  "permissions": [ ]
}
```

Figura 04: Modelo JSON tabela License

```
"Repository": {
  "full_name": "String",
  "language": "String",
  "forks": "Int",
  "created_at": "Date",
  "licenseid": "String",
  "issues": [ ]
}
```

Figura 05: Modelo JSON tabela Repository

```
"User": {
  "login": "String",
  "name": "String",
  "followers": "Number",
  "type": "String",
  "location": "String",
  "public_repos": [ ],
  "subscriptions": [ ]
});
```

Figura 06: Modelo JSON tabela User

```
"Permission": {
  "name": "String"
}
```

Figura 07: Modelo JSON tabela Permission





Collection Name	Documents	Avg. Document Size	Total Document Size	Num. Indexes	Total Index Size	Properties
issues	2,942	1.1 KB	3.2 MB	1	76.0 KB	
licenses	13	466.2 B	5.9 KB	2	44.0 KB	
repositories	48,154	232.5 B	10.7 MB	1	468.0 KB	
users	1,100	7.2 KB	7.7 MB	2	60.0 KB	

Figura 08: Count das coleções do utilizadas e populadas no mongoDB

2.3 Grupos de Usuários

Como o objetivo do projeto é descrito pela criação de um relatório ad-hoc para leitura e filtro de informações de uma API, é estudado que para o consumo de uma API do GitHub seria apenas importante para gerência de usuários que exista uma *role*, 'classe' de privilégios no Postgre, para suprir as necessidades da aplicação, e outra *role* que cumpra os papéis de uma dba para a gerência do banco. Os códigos utilizados para a criação desses papéis no Postgre são vistos nas Figuras 9 e 10 abaixo.

```
/*Criando role DBA BD*/
CREATE ROLE dba_github_database;
GRANT ALL PRIVILEGES ON DATABASE github_database TO dba_github_database WITH GRANT OPTION;
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA gitdatabase TO dba_github_database WITH GRANT OPTION;
GRANT ALL PRIVILEGES ON SCHEMA gitdatabase TO dba_github_database WITH GRANT OPTION;
CREATE USER geovany WITH PASSWORD 'b2NvcnZvZGlzc2VudW5jYW1haXM=';
GRANT dba_github_database TO geovany;
ALTER ROLE dba_github_database WITH CREATEROLE;
ALTER ROLE geovany CREATEROLE;
```

Figura 9: Código usado para a criação da role de DBA e um usuário geovany com as permissões da role.

```
--SCRIPT PARA O TRABALHO:
/*Criando role aplicação*/
CREATE ROLE application;
GRANT INSERT, SELECT, UPDATE, DELETE ON ALL TABLES IN SCHEMA gitdatabase TO application;
GRANT USAGE ON SCHEMA gitdatabase TO application;
CREATE USER app WITH PASSWORD 'VrCv%(6;9&';
GRANT application TO app;
```

Figura 10: Código usado para criar a role da aplicação e um usuário app com as permissões da role.

2.3 Índices

Durante a otimização do banco foram criados diversos índices baseados nos filtros que pensado ser os mais recorrentemente usados, assim criando a necessidade de uso mais rápido e eficiente proveniente dos índices, mesmo com o custo de memória. Todos os índices criados são listados na tabela 01.

Tabela 01: Tabela de índices criados para o Postgre

	schemaname name	tablename name	indexname name	tablespace name	indexdef text
1	gitdatabase	repositories	index_full_name	[null]	CREATE INDEX index_full_name ON gitdatabase.repositories USING btree (full_name)
2	gitdatabase	users	index_user_login	[null]	CREATE INDEX index_user_login ON gitdatabase.users USING btree (login)
3	gitdatabase	users	index_user_name	[null]	CREATE INDEX index_user_name ON gitdatabase.users USING btree (name)
4	gitdatabase	users	index_user_local	[null]	CREATE INDEX index_user_local ON gitdatabase.users USING btree (location)

Índices como o nome de usuário e nome do repositório são essenciais, uma vez que esse é o principal artefato fornecido pelo GitHub e portanto os mais pesquisados. No caso do full_name do repositório, ainda é possível usá-lo para encontrar o dono do repositório com mais facilidade em alguns casos, já que compõem de 'criador/repositório'.

Outras alternativas de busca interessante e que também podem ser recorrentes, é um filtro de usuários por localidade ou busca por login, já que alguns usuários podem não ter nome.

2.4 Triggers

Um trigger pensado para facilitar as atualizações futuras quanto a adição e remoção de funcionários, foi criado para o incremento ou decremento do número de repositórios públicos criados por determinado usuário. A Figura 11 representa o código usado na criação do trigger.

```
CREATE OR REPLACE FUNCTION att_num_repos() RETURNS TRIGGER AS $$ BEGIN
IF (TG_OP = 'INSERT') THEN
UPDATE gitdatabase.users SET qt_repos = qt_repos + 1 WHERE login = NEW.owner;
ELSEIF (TG_OP = 'DELETE') THEN
UPDATE gitdatabase.users SET qt_repos = qt_repos - 1 WHERE login = OLD.owner;
END IF;
RETURN NULL;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER t_att_num_repos AFTER INSERT OR DELETE ON gitdatabase.repositories FOR EACH ROW
EXECUTE PROCEDURE att_num_repos();
```

Figura 11: Código utilizado na construção do trigger.

3. Testes com JMeter

Com os bancos modelados e construídos, foram realizados testes no JMeter, em uma máquina local, com o objetivo de verificar seu potencial. Para isso foi construído uma *query* de alto custo (Figura 12) para realizar consultas no banco, alternando o número de usuários com acesso simultâneo, para identificar o limite máximo aceito pelo banco. Em um segundo teste, com a mesma *query*, porém agora com o número de usuários fixos e alternando o número de requisições feitas por eles, o objetivo é medir a capacidade do banco em atender esses pedidos.

```

SELECT login, name, count(r) AS participating_repos FROM (
(SELECT u.id, u.login, u.name, r.id AS R FROM gitdatabase.users u
JOIN gitdatabase.repositories r ON r.owner = u.id)
UNION (SELECT u.id, u.login, u.name, rs.repoid AS R FROM gitdatabase.users u
JOIN gitdatabase.repository_subscriptions rs ON rs.userid = u.id)
) AS con
WHERE id > (SELECT owner FROM gitdatabase.repositories
WHERE stargazers_count > 10
AND forks >10
AND owner NOTNULL
AND language = 'Java'
AND licenseid = 'mit')
GROUP BY (login, name)

```

Figura 12: Query SQL para requisições usadas para testes com o banco PostgreSQL. “Login, nome e quantidade de repositórios a qual participa ou é dono de todos os usuários cujo o ID é maior que o ID do usuários que corresponde ao dono do repositório que contém mais de 10 estrelas, mais de 10 forks, linguagem principal java e id de licença igual a ‘mit’

As principais informações da primeira bateria de testes com o postgres podem ser visualizadas na Tabelas 2 (Nº de usuários, taxa de sucesso, vazão e latência média). Observando os resultados é possível construir os gráficos das Figuras 13 e 14, baseada na latência e vazão x número de usuários. Com essas informações podemos visualizar que o número máximo de usuários simultâneos é um valor entre 500 e 750.

Ressalta-se que os testes foram realizados localmente na máquina de um dos integrantes, que possui sistema operacional 64bits, windows 10 e processador intel core i5 2.5GHz.

Tabela 2: Resultados dos testes para usuários conectados simultaneamente no Postgres.

Nº DE USUÁRIOS	SUCESSO (%)	LATÊNCIA MÉDIA(ms)	Vazão
10	100.00%	726	6.5/seg
100	100.00%	7396	9.0/seg
250	100.00%	19834	8.7/seg
500	100.00%	23820	11.7/seg
750	68.27%	25987	13.8/seg

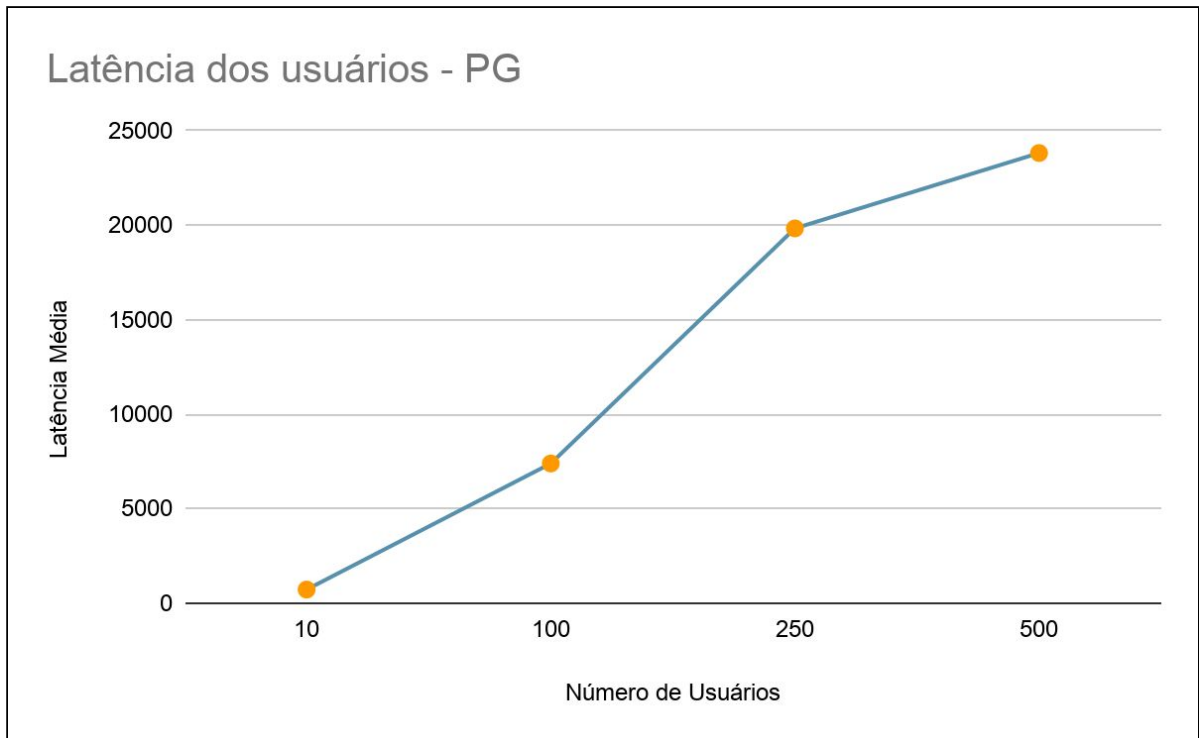


Figura 13: Gráfico de Latência x Usuários simultâneos, dos resultados apresentados no JMeter com o Postgre.

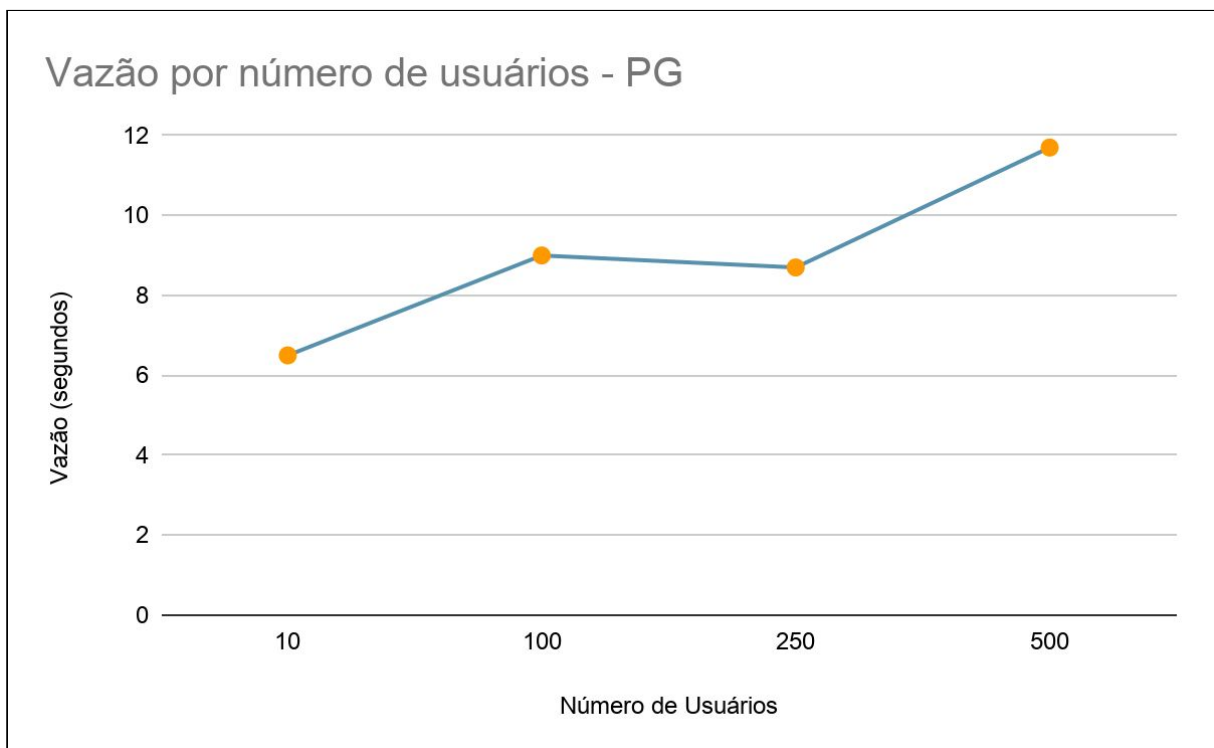


Figura 14: Gráfico de Vazão x Usuários simultâneos, dos resultados apresentados no JMeter com o Postgre.

Para a segunda bateria de teste, visto os resultados da primeira, o número de usuários simultâneos foi fixado em 100, alternando o número de requisições a cada teste. Os resultados podem ser visualizados na Tabela 3, e em representação gráfica nas Figura 15 e 16.

Tabela 3: Resultados dos testes para requisições sequenciais com cada um dos 96 usuários conectados simultaneamente (Postgre).

Nº DE REQUISIÇÕES	SUCESSO (%)	LATÊNCIA MÉDIA(ms)	Vazão
10	100.00%	4155	19.2/seg
100	100.00%	4685	20.3/seg
250	100.00%	4791	18.9/seg
500	100.00%	4781	20.7/seg
750	100.00%	4729	21/seg
1000	100.00%	4659	21.4/seg

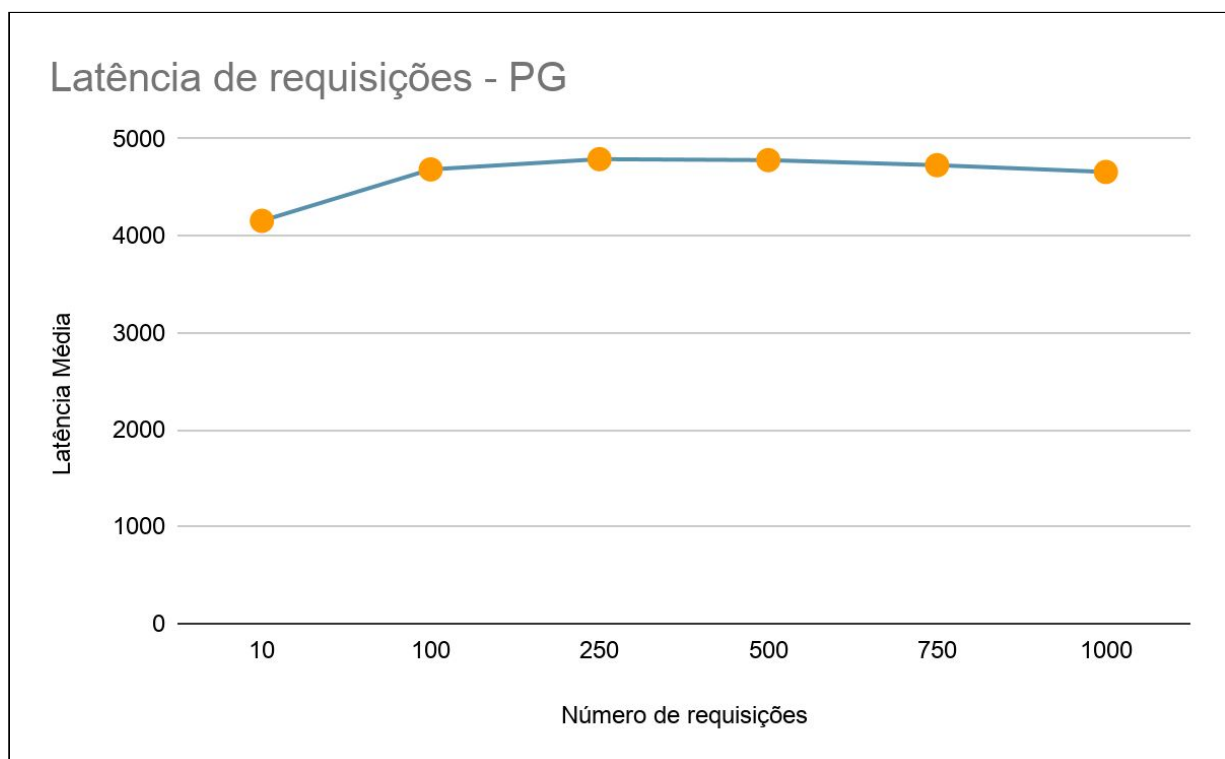


Figura 15: Gráfico de Latência x Nº Requisições por usuário, dos resultados apresentados no JMeter para o Postgres.

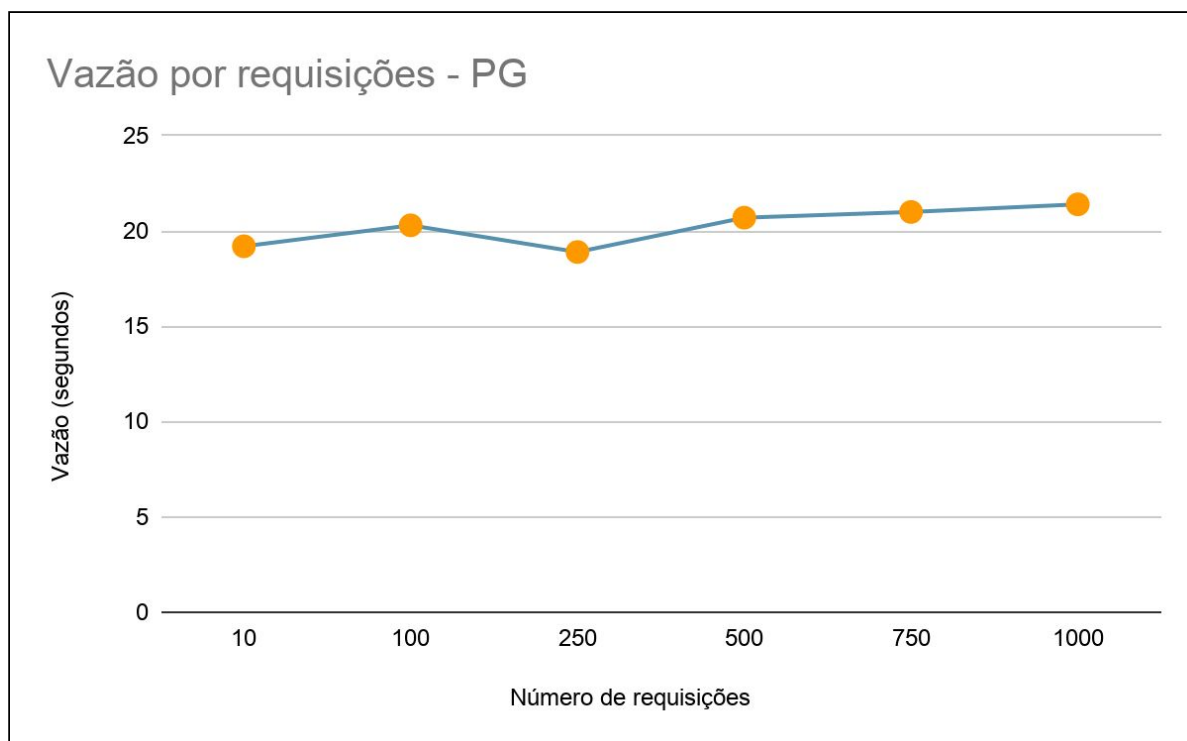


Figura 16: Gráfico de Vazão x N° Requisições por usuário, dos resultados apresentados no JMeter para o Postgres.

Os resultados demonstraram grande estabilidade do postgres a lidar com um alto número de requisições por usuário, mostrando grande estabilidade tanto na latência quanto na vazão.

Realizando as duas baterias de testes, porém agora para o banco do mongo, novas configurações são feitas. As conexões do JMeter com o mongo exigem maiores recursos e ainda não é muito eficiente, fazendo com que exista a necessidade de uma consulta mais simples, a qual foi dada com a seguinte query: “primeiro repositório com 'star_gazers_count' entre 50 a 60”. Esse os resultados para a primeira bateria de testes podem ser vistos na Tabela 4 e gráficos das Figuras 17 e 18.

Tabela 4: Requisições sequenciais para cada um dos 96 usuários conectados simultaneamente.

Nº DE REQUISIÇÕES	SUCESSO (%)	LATÊNCIA MÉDIA(ms)	Vazão
10	100.00%	19	11.5/seg
100	100.00.%	221	90.2/seg
1000	100.00.%	2482	254.3/seg
1250	100.00%	3091	246.1/seg
1500	100.00%	3365	272.3/seg
1750	95.89%	3752	248.7/seg

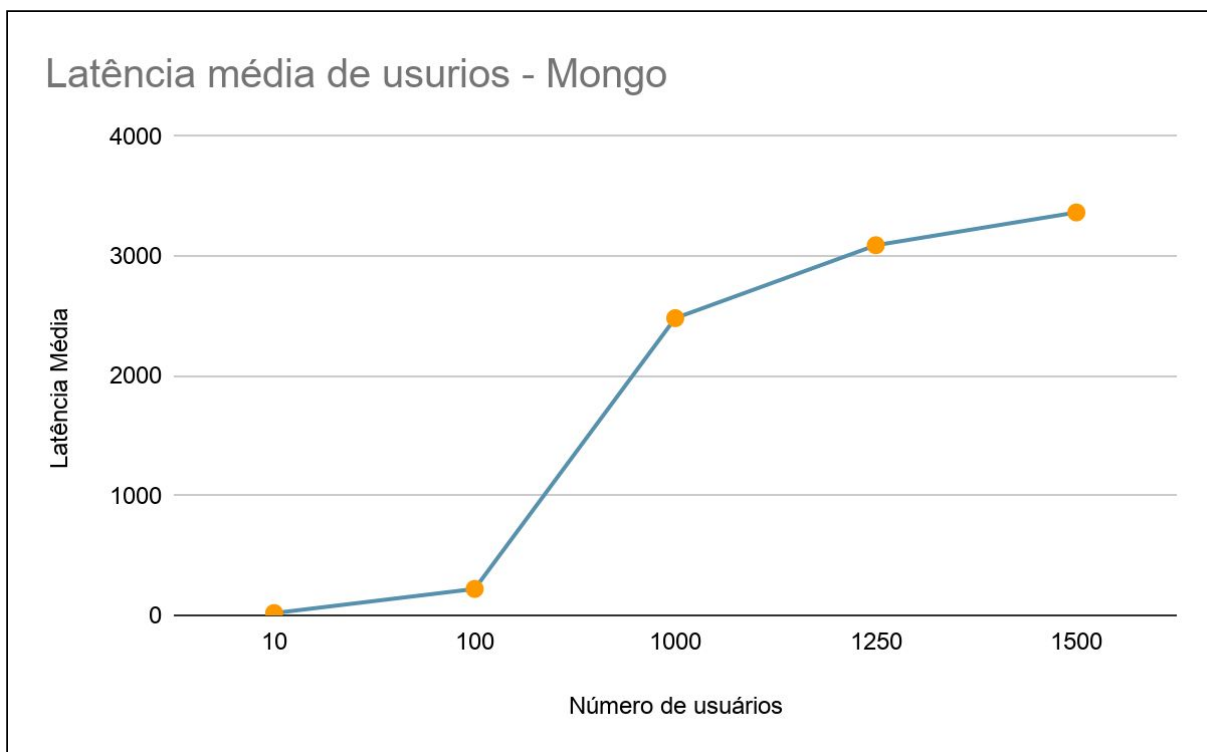


Figura 17: Gráfico de Latência x Usuários simultâneos, dos resultados apresentados no JMeter com mongo.

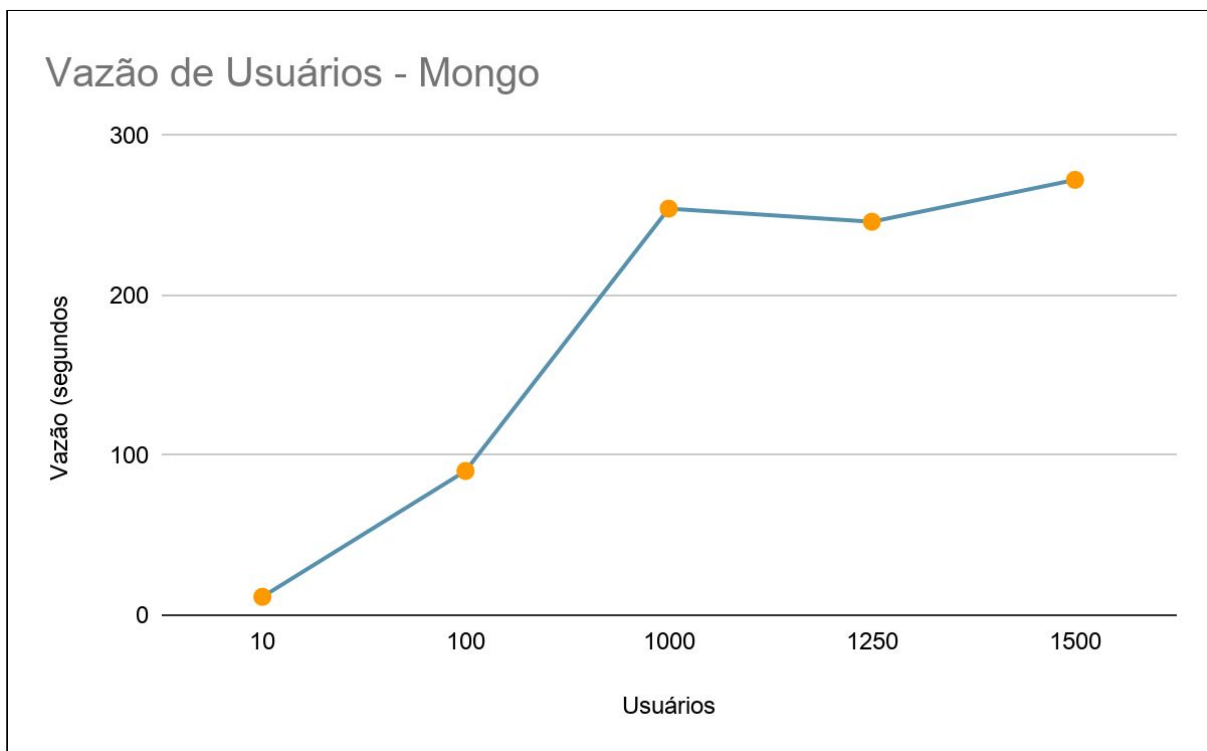


Figura 18: Gráfico de Vazão x Usuários simultâneos, dos resultados apresentados no JMeter com mongo.

Observando as respostas dos usuários para o mongo, percebe-se uma maior capacidade de atender a usuários simultâneos. Todavia, com a segunda bateria de

testes, observou-se a incapacidade do JMeter, com o mongo, em atender a uma grande taxa de requisições por usuários, “*crashando*” a ferramenta. Propondo uma melhor visualização gráfica, os testes então são validados usando 100 usuários fixos e alterando as requisições de acordo com a Tabela 5, gerando também o gráfico das Figuras 19 e 20.

Tabela 5: Requisições sequenciais para cada um dos 96 usuários conectados simultaneamente.

Nº DE REQUISIÇÕES	SUCESSO (%)	LATÊNCIA MÉDIA(ms)	vazão
10	100.00%	424	202.7/seg
25	100.00%	393	219/seg
50	100.00%	333	282.3/seg
60	100.00%	306	308.1/seg
70	100.00%	303	315.2/seg
80	100.00%	304	315/seg
90	90.59%	3039	21.7/seg

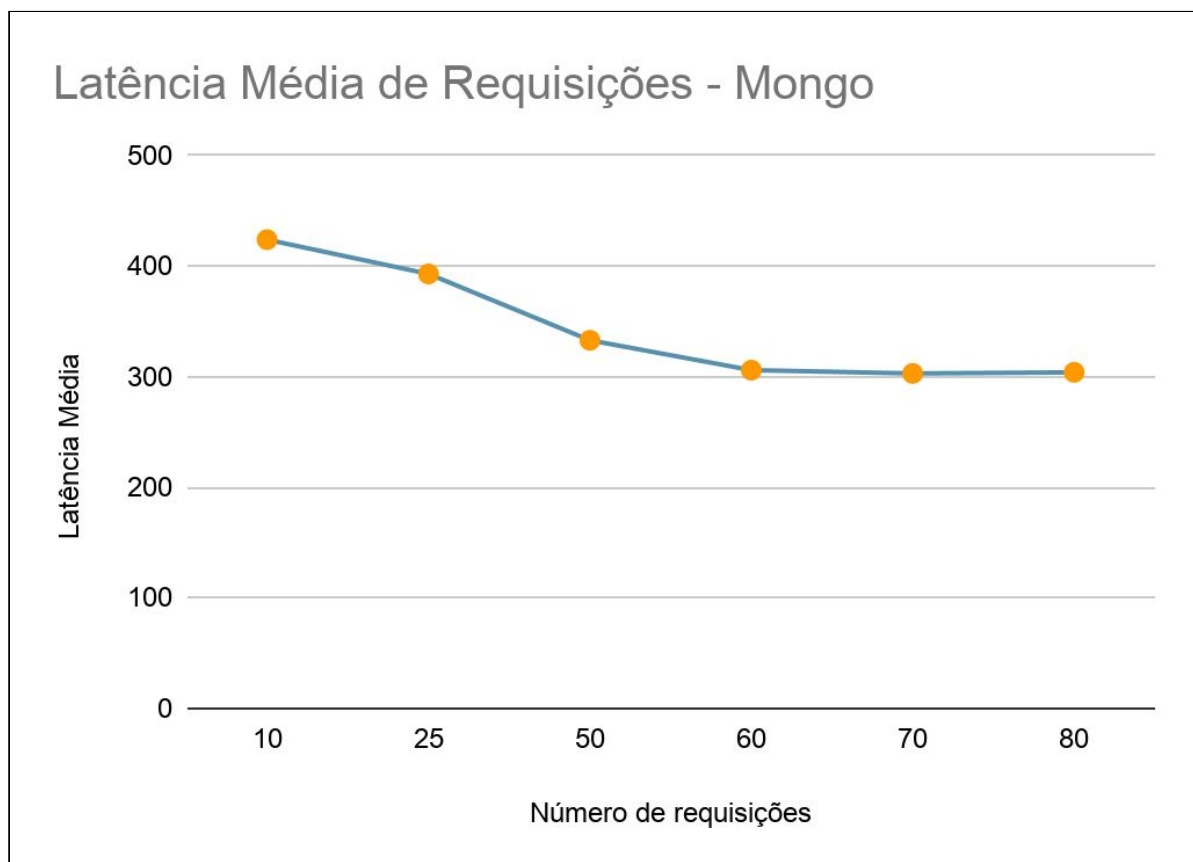


Figura 19: Gráfico de Latência x Nº Requisições por usuário, dos resultados apresentados no JMeter para o (mongo).



Figura 20: Gráfico de vazão x N° Requisições por usuário, dos resultados apresentados no JMeter para o (mongo).

Tirando uma conclusão dos testes, percebe-se uma capacidade notável do mongo a responder requisições simultâneas, porém com grande variação em latência e taxa de erros à medida que mais requisições são feitas. Já o Postgre foi o contrário, forneceu menos capacidade de requisições simultâneas, mas maior estabilidade de latência e taxa de erros.

4. Relatório Ad-hoc

Para a geração do relatório Ad-hoc, foi construído um formulário JavaScript no Front-end da aplicação, o qual compõe dos principais filtros e atributos dos quais podem ser gerados a tabela resposta do relatório. O usuário é capaz de selecionar a entidade principal da tabela ('Users' ou 'Repositories'), os atributos a quais querem que apareça, e até filtros para cada tipo de atributo. Nas Figuras 21 e 22 é possível ver o formulário específico de cada entidade.

AD-HOC GIT REPORT

REPORT FORM

Name
NAME

Table
Users

Min Followers 0 **Type** User **Min Repos** 0

Attributes
login name followers type location public_repos

MAKE A REPORT

Figura 21: Formulário para relatório de Usuários.

REPORT FORM

Name
NAME

Table
Repositories

Languages
TYPE A LANGUAGE → Java
JavaScript
Clojure

Min Stars 0 **Min Forks** 0

Creation date begin
mm / dd / yyyy

Creation date end
mm / dd / yyyy

Licenses
TYPE A LICENSE → agpl-3.0

Owner
OWNER

Figura 22: Formulário para relatório de Repositórios.

Com a seleção feita, o botão “make a report” passa-se os parâmetros para a construção da query no Back-end, o qual utilizará o ORM para realizar a consulta ao especificada no banco. Com a resposta em mãos, o Front-End automaticamente

gera um relatório de resposta em formato de tabela (Exemplo Figura 23), a qual é paginada de 150 em 150 dados que podem ser passados pelos botões “next” e “previous”. Essa divisão de páginas contribui para a eficiência do tempo de resposta e no tamanho da carga recebida por vez.

Result						
id	login	name	followers	type	location	public_repos
604	evs	Elliot Schoemaker	11	u	Brisbane, Australia	3
609	weepy	weepy	114	u	London	85
614	wunki	Petar Radošević	74	u	Netherlands	6
619	shanesveller	Shane Sveller	48	u	Chicago, IL	73
622	peks	Martin Zibert	12	u	Slovenia	8
630	skddc	Sebastian Kippe	110	u	Earth	86
634	natacado	Paul Paradise	34	u	Seattle, WA	34
639	mountain	Mingli Yuan	225	u	Beijing	51
643	olivM	Olivier Mourlevat	39	u	Paris, France	78

PREVIOUS

NEXT

Figura 23: Formulário para relatório de Repositórios.

5. Conclusão

A execução do projeto apresentou dificuldades inicialmente devido ao limite de requisições que a API estabelece. Uma aplicação não autenticada possui um número máximo de requisições muito baixo e para contornar esse problema cada membro do grupo gerou um token no site do Github, fornecendo ele para aplicação através de variáveis ambiente. Deste modo, mesmo que algum usuário esgotasse suas requisições, era possível utilizar outro. Ainda sim, em certos momentos a API bloqueou as requisições pelo IP da máquina utilizada. Ainda durante a captura de dados, fomos surpreendidos com certos tamanhos de dados que não esperávamos, como o nome de um repositório com mais de 200 caracteres. Nenhuma consulta utilizada no relatório comprometeu o desempenho da aplicação, mesmo com um número muito grande de variáveis envolvidas. Por mais que os dados estejam sendo limitados no banco e nos modelos, é preciso tratá-los assim que chegam da API para evitar qualquer problema nas operações realizadas, como a inserção de um grande número de registros em uma única execução. Aprendemos a tratar um número grande de dados, uma vez que agora sabemos que a validação em uma camada superior permite evitar erros de forma mais adequada e garantir melhor alocação de recursos do nosso banco. Mencionando também os testes do JMeter, os resultados ajudaram a entender novas noções de capacidade e eficiência dos dois bancos, mongo e Postgre, ainda assim, a fraca capacidade da máquina utilizada nos testes pode ter interferido com os resultado e em alguns momentos causou problemas de “*crash*” trabalhosos.

GitHub com os códigos -

<https://github.com/geovanymds/github-api-report>