Universidade Federal de Itajubá
Ciência da Computação

Banco de Dados II:
Trabalho Prático - Parte I

Álison Mesquita Maciel
Carlos Daniel Cipriano da Silva Ramos
Jackson Galdino da Silveira
João Pedro Malta Ardo
Matheus Martins Batista

Orientador: Profª. Dra. Vanessa Cristina Oliveira de Souza
Instituto de Matemática e Computação

Minas Gerais
2023

Álison Mesquita Maciel [1]
Carlos Daniel Cipriano da Silva Ramos [2]
Jackson Galdino da Silveira [3]
João Pedro Malta Ardo [4]
Matheus Martins Batista [5]
Vanessa Cristina Oliveira de Souza[6]

Resumo

Nesse trabalho, foi desenvolvida uma aplicação para consumir dados da API Internet Game Database (IGBD) através do PostgreSQL, e também otimizações e testes de performance para o banco de dados criado.

[1] Graduando em Ciência da Computação pela Universidade Federal de Itajubá – 2018011367 – E-mail: alisonmesquita98@hotmail.com
[2] Graduando em Ciência da Computação pela Universidade Federal de Itajubá – 2020032444 – E-mail: d2020032444@unifei.edu.br
[3] Graduando em Engenharia da Computação pela Universidade Federal de Itajubá – 2018010136 – E-mail: jacksongdas@unifei.edu.br
[4] Graduando em Ciência da Computação pela Universidade Federal de Itajubá – 2020029126 – E-mail: joaopedro.m.a@hotmail.com
[5] Graduando em Ciência da Computação pela Universidade Federal de Itajubá – 2019005687 – E-mail: matmb@unifei.edu.br
[6] Professora orientadora. Doutora em Computação Aplicada pelo Instituo Nacional de Pesquisas Espaciais. Docente na Universidade Federal de Itajubá – E-mail: vanessasouza@unifei.edu.br

Sumário

1 Introdução

Este trabalho visou desenvolver uma aplicação para consumir dados da API Internet Game Database (IGBD), que pertence à Twitch, uma subsidiária da Amazon que disponibiliza informações úteis sobre jogos, como gêneros, plataformas, críticas, enredo, personagens e modos de jogo.

Após a análise dos dados, utilizou-se somente aqueles que seriam interessantes para gerar as consultas: jogos, personagens, gêneros, plataformas, modos de jogos e companhias. Os dados foram fornecidos por interface e retornavam a requisição em formato JSON. Ademais, vale ressaltar que foi necessário tratá-los para inserir no banco de dados com Sistema de Gerenciamento de Banco de Dados (SGBD) relacional PostgreSQL.

1.1 Motivação

Conforme as diretrizes propostas na parte um do trabalho da disciplina de Banco de Dados II ofertada pela Universidade Federal de Itajubá, o grupo em questão ficou incumbido de selecionar e consumir os dados de uma API. Nessa perspectiva, a API escolhida foi a versão 4 da API do IGDB e realizou-se o processo de análise dos dados fornecidos.

A API do IGDB (Internet Game Database) é uma interface de programação de aplicativos que fornece acesso a um vasto repositório de informações relacionadas a jogos. Ela permite que desenvolvedores obtenham dados detalhados sobre jogos, incluindo informações sobre títulos, plataformas, gêneros, classificações, avaliações, imagens, vídeos e muito mais. A API do IGDB é projetada para facilitar a integração de dados de jogos em aplicativos, sites e serviços relacionados a jogos, fornecendo uma maneira eficiente de pesquisar, filtrar e recuperar informações precisas e atualizadas sobre uma ampla variedade de títulos de jogos. Com essa API, os desenvolvedores podem criar experiências personalizadas, construir recursos de recomendação, criar listas de desejos, exibir análises e criar ferramentas poderosas para a comunidade de jogadores.

2 Metodologia

Essa seção visa esclarecer o processo para desenvolvimento de um *script* de carga feito com a linguagem de programação Python e o padrão de arquitetura *Model View Controller (MVC)*.

2.1 Coleta e tratamento de dados

Com base na documentação para desenvolvedores do IGDB API v4, o *script* utiliza a biblioteca Apicalypse para Python, esta fornece uma interface simplificada para construir consultas e enviar solicitações à API, permitindo aos desenvolvedores obter informações específicas dos jogos com facilidade. Os parâmetros abaixo foram respeitados para consumir os dados da API:

- Não fazer mais 4 requisições por segundo;
- Converter o JSON array retornado em um objeto (dicionário) pelo Python;
- 500 é o limite de retorno para cada requisição;
- Analisar se houve sucesso na requisição.

Ademais, com o sucesso das requisições foi necessário analisar quais *endpoints* consumir. A equipe optou pelos seguintes interfaces:

- Games;
- Genres;
- Plataform;
- Companies;
- Character;
- Companies.

Os pormenores do tratamento dos dados retornado para cada manipulação e inserção no banco estão comentados no *script*. Por fim, os códigos SQL desenvolvidos, o *script*, instruções de execução e documentação foram disponibilizados em repositório no Github (link no Capítulo 5).

2.2 Testes no Banco de Dados

O Apache JMeter foi utilizado para analisar a performance do banco criado, essa ferramenta de teste de desempenho de código aberto desenvolvida pelo projeto Apache. O JMeter é capaz de simular múltiplos usuários virtuais que enviam solicitações HTTP, FTP, JDBC, SOAP, entre outras, para um servidor alvo, permitindo a análise de métricas como tempo de resposta, taxa de transferência, latência e consumo de recursos.

3 Banco de Dados

Esta seção busca explicitar acerca da modelagem, tabelas, índices e funções criados para realizar o processo de carga no banco.

3.1 Modelo Entidade Relacionamento

A aplicação utiliza um banco relacional com o PostgreSQL no qual a modelagem encontra-se na Figura 1.

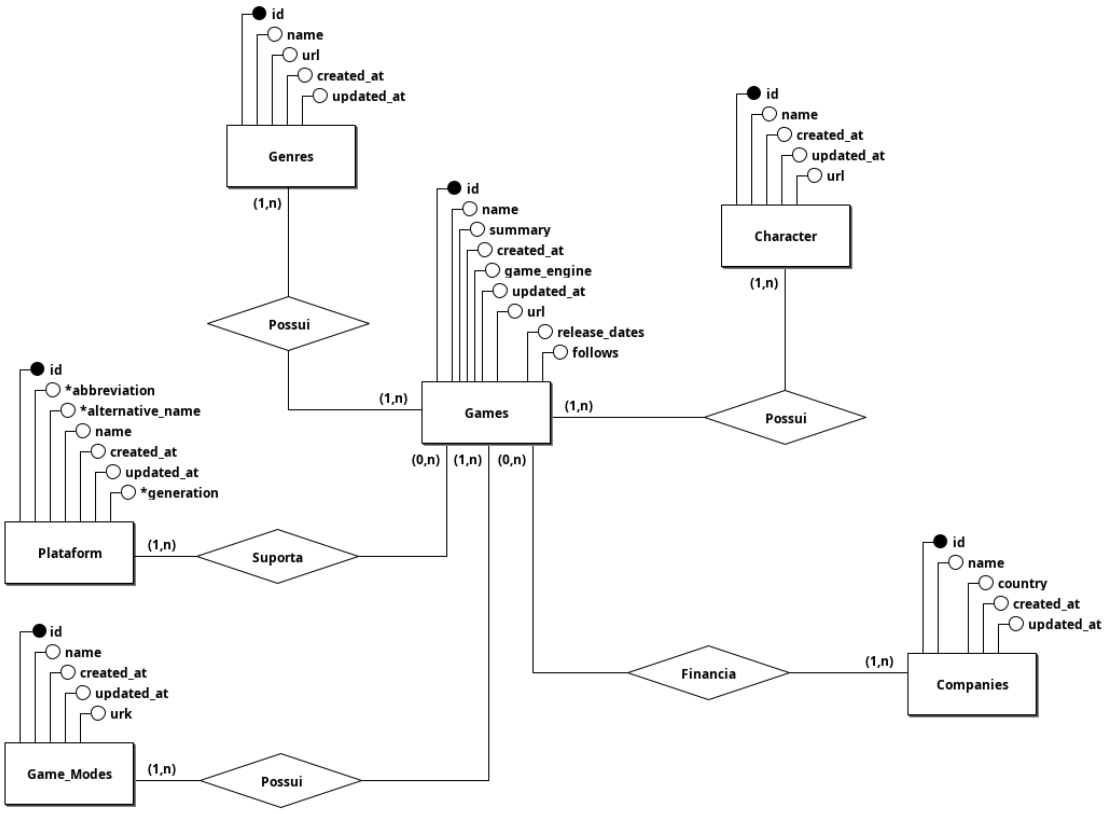

Figura 1 – MER da Aplicação

3.2 Criação das Tabelas

Segue o código SQL para a criação das tabelas principais no banco de dados:

```
CREATE TABLE plataform(
```

```
    id smallint PRIMARY KEY NOT NULL,
    name varchar(100) NOT NULL,
    created_at date NOT NULL,
    updated_at date NOT NULL,
    abbreviation varchar(50),
    alternative_name varchar(100),
    generation smallint
);

CREATE TABLE genres(
        id smallint PRIMARY KEY NOT NULL,
        name varchar(100) UNIQUE NOT NULL,
        url text NOT NULL,
        created_at date NOT NULL,
        updated_at date NOT NULL
);

CREATE TABLE games_modes(
    id smallint PRIMARY KEY NOT NULL,
    name varchar(40) UNIQUE NOT NULL,
    created_at date NOT NULL,
    updated_at date NOT NULL,
    url text NOT NULL
);

CREATE TABLE companies(
    id int PRIMARY KEY NOT NULL,
    name text NOT NULL,
    country text,
    created_at date NOT NULL,
        updated_at date NOT NULL
);

CREATE TABLE games(
    id int PRIMARY KEY NOT NULL,
    name text NOT NULL,
    created_at date NOT NULL,
    updated_at date NOT NULL,
    url text NOT NULL,
```

```
    summary text,
    game_engines text,
    follows int,
    release_date date
);

CREATE TABLE character(
    id int PRIMARY KEY NOT NULL,
    name text NOT NULL,
    created_at date NOT NULL,
    updated_at date NOT NULL,
    url text NOT NULL
);
```

Além disso, também criou-se as tabelas de relacionamento com o SQL abaixo:

```
CREATE TABLE game_plataform(
    id_game int NOT NULL,
    id_plataform smallint NOT NULL,
    FOREIGN KEY (id_game) REFERENCES games(id),
    FOREIGN KEY (id_plataform) REFERENCES plataform(id)
);

CREATE TABLE game_company(
    id_game int NOT NULL,
    id_company int NOT NULL,
    FOREIGN KEY (id_game) REFERENCES games(id),
    FOREIGN KEY (id_company) REFERENCES companies(id)
);

CREATE TABLE game_genre(
    id_game int NOT NULL,
    id_genre smallint NOT NULL,
    FOREIGN KEY (id_game) REFERENCES games(id),
    FOREIGN KEY (id_genre) REFERENCES genres(id)
);

CREATE TABLE game_character(
    id_game int NOT NULL,
    id_character int NOT NULL,
    FOREIGN KEY (id_game) REFERENCES games(id),
```

```
    FOREIGN KEY (id_character) REFERENCES character(id)
);


CREATE TABLE game_gamemode(
    id_game int NOT NULL,
    id_game_mode smallint NOT NULL,
    FOREIGN KEY (id_game) REFERENCES games(id),
    FOREIGN KEY (id_game_mode) REFERENCES games_modes(id)
);
```

3.3 Grupos de Usuários

Foram criados as seguintes roles para acessar o banco, cada uma com suas permissões específicas:

- Usuário: possui permissão para consultar em todas as tabelas do banco;
- Desenvolvedor: pode realizar as operações CRUD apenas para os registros de todas as tabelas do banco;
- Administrador: possui acesso total, pode realizar operações CRUD, manipular as tabelas, conceder privilégios e criar novos usuários ou roles.

```
CREATE ROLE usuario;
CREATE ROLE dev;
CREATE ROLE dba;

GRANT SELECT ON ALL TABLES IN SCHEMA public TO usuario;
GRANT SELECT, INSERT, DELETE, UPDATE ON ALL TABLES IN
    SCHEMA public TO dev;
GRANT ALL ON DATABASE "IGDB" TO dba;

CREATE USER martinsDBA WITH PASSWORD 'fabosmati';
CREATE USER martinsDEV WITH PASSWORD 'fabosmati';
CREATE USER martinsUSER WITH PASSWORD 'fabosmati';

GRANT usuario TO martinsUSER;
GRANT dev TO martinsDEV;
GRANT dba TO martinsDBA;
```

```
GRANT USAGE ON SCHEMA public TO martinsUSER;
GRANT USAGE ON SCHEMA public TO martinsDEV;
GRANT USAGE ON SCHEMA public TO martinsDBA;
```

3.4 Índices

Para melhorar o desempenho das consultas e otimizar o acesso aos dados, foi criado o índice sobre o atributo 'Data de Lançamento dos Jogos' e índices compostos sobre os relacionamentos 'Plataforma e Games', 'Gênero e Games', 'Modo de Jogo e Games'. Segue o código SQL utilizado na criação dos índices:

```
CREATE INDEX release_game_date
ON games USING btree (release_date);

CREATE INDEX idx_game_plataform
ON game_plataform (id_game, id_plataform);

CREATE INDEX idx_game_genres
ON game_genre (id_game, id_genre);

CREATE INDEX idx_game_game_mode
ON game_gamemode (id_game, id_game_mode);
```

Vale salientar que os índices criados foram previamente testados com a ferramenta *EXPLAIN ANALYZE* do PostgreSQL, durante o teste outros índices foram retirados pois não estavam sendo utilizados durante as consultas. Concomitantemente, encontrou-se consultas que serão úteis na segunda parte deste trabalho e utilizam esses índices (códigos de manipulação do banco estão disponíveis no repositório remoto).

3.5 Funções

Para realizar a contagem dos registros em cada tabela criou-se uma função no banco. A Figura 2 mostra o número de registro de cada tabela.

Criação da função em SQL para consultar o número de registros:

	table_name name	count_rows integer
1	game_genre	70676
2	games	60000
3	game_plataform	52017
4	companies	45772
5	game_gamemo...	30990
6	character	16501
7	game_company	4129
8	game_character	3528
9	plataform	200
10	genres	23
11	games_modes	6

Figura 2 – Contagem de Registros

```
CREATE OR REPLACE FUNCTION count_rows(tablename text)
RETURNS integer AS
    $$
    DECLARE
            result integer;
    BEGIN
            EXECUTE 'SELECT count(*) FROM' ||
            tablename INTO result;
            RETURN result;
    END;
    $$ language plpgsql;

SELECT table_name, count_rows(table_name)
FROM information_schema.tables
WHERE table_schema NOT IN
    ('pg_catalog', 'information_schema')
ORDER by count_rows(table_name) DESC
```

Um ponto a se destacar sobre quantidade de registros é que, a princípio, o grupo decidiu popular o banco com todos os 230 mil registros de jogos fornecidos pela API. No entanto, durante a realização do teste de desempenho, chegou-se ao consenso de que 60 mil registros seriam suficientes.

4 Resultados

A seção a seguir busca elucidar sobre os resultados obtidos com o teste de performance realizado com o JMeter.

4.1 Configuração

Antes de realizar os teste foi necessário realizar a configuração do JMeter, das limitações de conexão do banco e do Driver de conexão utilizado.

4.1.1 JMeter

O JMeter foi configurado de modo que seu plano de teste utilizasse o Java Database Connectivity (JDBC) para fazer as requisições ao banco. Configuramos o JDBC para não limitar o número de conexões ativas, utilizou-se o valor padrão estipulado, 10000ms, para o aguardo da resposta da requisição e todas as *threads* são iniciadas ao mesmo tempo durante os testes.

4.1.2 Especificações da Máquina

Também, as especificações da máquina utilizada são:

- Processador - Intel Xeon E5-2650 v3 (10/20) @ 3.000GHz;
- Memória RAM - 16GB (12GB destinados para JMeter)
- SSD - 256GB NX SERIES M2 3GB/segundo leitura e escrita
- Sistema Operacional - Manjaro Linux x86_64 (6.1.31-2-MANJARO)
- Placa de Vídeo - RX5700XT 8GB (VRAM)

4.1.3 Banco de Dados

As alterações realizadas no arquivo de configuração do PoistgreSQL 15 *postgresql.conf* foram:

- Quantidade máxima de usuário - *max_connections* = 500;
- Quantidade de memória dedicada para *buffer* - *shared_buffers* = 2GB

4.2 Consulta Eleita

Para realizar os testes com JMeter optou-se pela consulta abaixo que retornava 59961150.0 bytes.

```
SELECT * FROM games g
    JOIN game_plataform gp ON gp.id_game = g.id
    JOIN plataform p ON gp.id_plataform = p.id
    JOIN game_genre gg ON gg.id_game = g.id
    JOIN genres gr ON gg.id_genre = gr.id
```

4.3 Requisições x Latência com 30 *Threads*

Para sintetizar as métricas, a tabela abaixo representa a quantidade de requisições e a latência para um número fixo de 30 threads.

Requisições	Latência Média (ms)
100	12301
200	12602
300	12850
400	12955
500	13026

Conforme o gráfico presente na Figura 3, a latência média do processo aumenta conforme a quantidade de requisições é elevada, entretanto percebe-se que o acréscimo de atraso é menor a cada incremento.

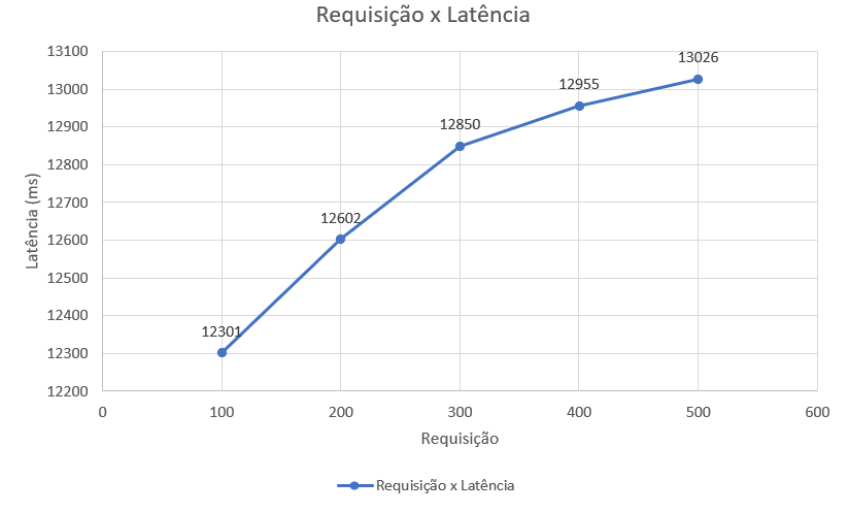

Figura 3 – Gráfico dos Resultados Requisições x Latência

Este resultado demonstra que atingido certo limiar de requisições, a latência tende convergir para um valor máximo, em que uma quantia de requisições superiores não aumenta

significativamente a latência média. Concomitantemente, estima-se que o banco de dados suporte um grande número de requisições para a consulta testada.

4.4 Usuário x Latência com 1 Requisição fixa

Conforme as diretrizes, o teste foi executado 30 vezes e com os valores obtidos faz-se a média das latências obtidas. Para sintetizar as métricas, a tabela abaixo representa a quantidade de usuários simultâneos e a latência.

Usuários	Latência Média (ms)
30	6716
40	9139
50	14913
60	16992
70	20043

Conforme o gráfico presente na Figura 4, quanto mais usuários ativos realizando requisições, maior será a latência envolvida no processo. Analisando a tabela acima, o valor limite encontrado foi de 70 usuários fazendo requisições ao mesmo tempo. Logo, qualquer quantidade maior que essa resultará em uma porcentagem de erro nos testes.

Nos casos em questão onde temos o banco de dados operando no seu limite, a comunicação com o banco pode se tornar instável e novas conexões podem não ser estabelecidas. Durante os testes realizados encontramos alguns erros, estes nos auxiliaram no processo para descobrir os limites do banco e serão detalhados nas próximas seções.

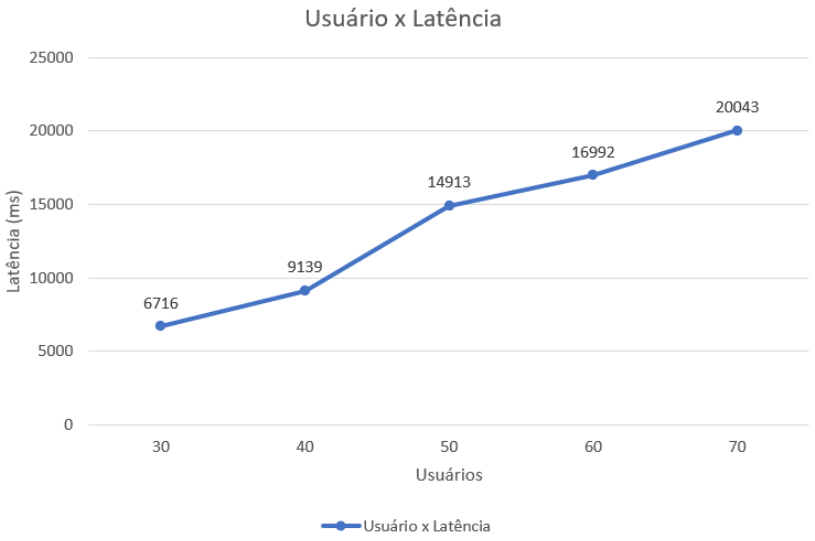

Figura 4 – Gráfico dos Resultados Usuário x Latência

4.5 *PQSLException: An I/O error occurred while sending to the backend*

O erro indica que ocorreu um erro de E/S (entrada/saída) ao enviar dados para o banco de dados, o contexto em questão indica que pode ter ocorrido alguma interrupção na conexão entre a aplicação e o banco, alguma falha ou indisponibilidade na rede no momento da requisição. Além disso, abaixo temos alguns cenários que podem enfrentar o mesmo erro:

- Problemas de capacidade: Se o banco de dados estiver sobrecarregado ou com recursos insuficientes, ele pode não conseguir lidar com a quantidade de requisições sendo enviadas pela aplicação.
- Configuração incorreta: Configurações incorretas no lado do cliente ou servidor podem levar a problemas de E/S. Isso pode incluir configurações de tempo limite (timeout), tamanhos de buffer ou outras configurações relacionadas à comunicação entre a aplicação e o banco.

4.6 *SQLException: Error preloading the connection pool*

O erro indica que ocorreu um problema ao realizar o pré-carregamento (preloading) do pool de conexões do banco de dados. O pool de conexões é uma coleção (ou conjunto) de conexões que o próprio banco cria antecipadamente, visando melhorar a eficiência e o desempenho do banco, reduzindo a latência na comunicação em questão. O erro pode acontecer nas situações abaixo:

- Falha na conexão com o banco de dados: Não foi possível estabelecer a conexão com o banco de dados por conta de algum problema de rede, falta de permissões ou configurações no banco.
- Número máximo de conexões excedidas: O número máximo de conexões permitidas pelo banco (ou pelo próprio pool) foi atingido, sendo assim, temos um bloqueio ao tentar estabelecer novas conexões.

5 Material desenvolvido

Os materiais produzidos estão disponíveis nos *links* abaixo:

- Código Fonte: script_carga_igdb
- Vídeo Explicativo: Youtube