TIAGO DE OLIVEIRA CARLOS

ESTUDO SOBRE O DESEMPENHO DE BIBLIOTECAS EM PYTHON PARA MAPEAMENTO OBJETO-RELACIONAL

Lavras-MG

2023

TIAGO DE OLIVEIRA CARLOS

ESTUDO SOBRE O DESEMPENHO DE BIBLIOTECAS EM PYTHON PARA MAPEAMENTO OBJETO-RELACIONAL

Monografia apresentada à Universidade Federal de Lavras, como parte das exigências do Curso de Ciência da Computação, para a obtenção do título de Bacharel.

Prof. Dr. Neumar Costa Malheiros
Orientador

Prof. Dr. Paulo Afonso Parreira Júnior
Coorientador

LAVRAS-MG

2023

Ficha catalográfica elaborada pelo Sistema de Geração de Ficha Catalográfica da Biblioteca Universitária da UFLA, com dados informados pelo(a) próprio(a) autor(a).

Carlos, Tiago de Oliveira.
Estudo sobre o desempenho de bibliotecas em Python para mapeamento objeto-relacional / Tiago de Oliveira Carlos. - 2023.
37 p.

Orientador(a): Neumar Costa Malheiros.
Coorientador(a): Paulo Afonso Parreira Júnior.
Tese (doutorado) - Universidade Federal de Lavras, 2023.
Bibliografia.

1. Python. 2. SQL. 3. Mapeamento Objeto-Relacional. I. Malheiros, Neumar Costa. II. Júnior, Paulo Afonso Parreira. III. Título.

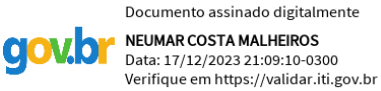
TIAGO DE OLIVEIRA CARLOS

ESTUDO SOBRE O DESEMPENHO DE BIBLIOTECAS EM PYTHON PARA MAPEAMENTO OBJETO-RELACIONAL

STUDY ON THE PERFORMANCE OF PYTHON LIBRARIES FOR OBJECT-RELATIONAL MAPPING

Monografia apresentada à Universidade Federal de Lavras, como parte das exigências do Curso de Ciência da Computação, para a obtenção do título de Bacharel.

APROVADA em 5 de Dezembro de 2023.
Dr. Denilson Alves Pereira UFLA
Dr. Ramon Gomes Costa UFLA



Prof. Dr. Neumar Costa Malheiros
Orientador

Prof. Dr. Paulo Afonso Parreira Júnior
Coorientador

LAVRAS-MG

2023

RESUMO

Desde seu surgimento, a técnica de mapeamento objeto-relacional (ORM) tem alcançado notável popularidade. Diversas bibliotecas surgiram, oferecendo diferentes implementações desta técnica, que vem se tornando um recurso padrão em grande parte dos *frameworks* de desenvolvimento Web. Assim, torna-se relevante uma análise da usabilidade dessa técnica, visando apresentar suas possíveis vantagens e desvantagens. Este trabalho teve como objetivo analisar o impacto da utilização dessas ferramentas no tempo de processamento de operações em bancos de dados em comparação com o acesso direto ao banco de dados. Utilizando a linguagem de programação Python, foram avaliados os tempos de execução do *framework* Django com sua ORM, da biblioteca SqlAlchemy junto ao *framework* Flask, e da biblioteca Psycopg2 para representar o acesso direto ao banco de dados, por meio de comandos SQL. Os resultados revelaram um aumento significativo no tempo de processamento ao utilizar uma ORM. O tempo extra foi justificado nas ferramentas de geração de consultas SQL e mapeamento de registros no banco de dados para objetos em Python. Foi possível concluir que, embora o aumento no tempo seja considerável, o uso de ORMs ainda é justificado, principalmente pelo desempenho apresentado por computadores e servidores modernos, e a existência de ferramentas capazes de oferecer apoio a aplicação, como sistemas de balanceamento de carga, que possibilitam o uso de diversas instâncias da mesma aplicação. Assim, é possível atingir um bom desempenho em aplicações que fazem uso de mapeadores objeto-relacional, porém sistemas de larga escala podem apresentar um custo de manutenção elevado, pois podem necessitar de um hardware mais robusto quando comparado à uma solução que utiliza do acesso direto ao banco de dados. O foco atual no desenvolvimento ágil favorece também o uso de *frameworks*, que permitem a criação rápida de sistemas através de entregas contínuas. Dessa forma, prioriza-se a criação de códigos de fácil leitura e manutenção, visto que o sistema está em constante evolução, e os requisitos do cliente podem variar a qualquer momento. Nesse contexto, o uso de código SQL dificulta o desenvolvimento, exigindo que a equipe apresente o conhecimento de uma linguagem extra, e apresentando consultas que necessitam ser atualizadas junto ao modelo de dados.

Palavras-chave: Python. Django. SqlAlchemy. Mapeamento Objeto-Relacional. SQL. Avaliação de desempenho.

ABSTRACT

Since its inception, the object-relational mapping (ORM) technique has gained a remarkable popularity. Several libraries have emerged, offering different implementations of this technique, which has become a standard resource in a significant portion of web development frameworks. Thus, a usability analysis of this technique becomes relevant, aiming to present its possible advantages and disadvantages. This study's main focus is to analyze the impact the use of these tools have on the processing time of database operations when compared to direct database access. Through the Python programming language, it was possible to evaluate the execution times of the Django framework and its ORM, the SqlAlchemy library with the Flask framework, and the Psycopg2 library representing access through SQL queries. The results revealed a significant increase in processing time when using an ORM. The additional time was justified in the tools for generating SQL queries and mapping data from the database to objects in Python. It was possible to conclude that, although the increase in time is considerable, the use of ORMs is still justifies, especially due to the performance offered by modern computers and servers, and the existence of tools capable of supporting the application, such as load balancing systems that enable the use of multiple instances of the same application. Thus, it is possible to attain good performance in applications that use object-relational mappers, however, large scale systems may have a higher maintenance cost, as they may require more robust hardware compared to a similar application that uses direct database access. The current focus on agile development also favors the use of frameworks, allowing for the rapid creation of systems through continuous deliveries. Therefore, the emphasis is on creating code that is easy to read and maintain, given that the system will be constantly evolving, and client requirements may change at any time. In this context, the use of SQL code hinders development, requiring the team to have knowledge of an additional language, that involves using queries that need to be updated along with the data model.

Keywords: Python. Django. SqlAlchemy. Object-Relational Mapping. SQL. Performance evaluation.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE QUADROS

SUMÁRIO

1 INTRODUÇÃO

O rápido avanço da tecnologia tem introduzido novos desafios, sendo o gerenciamento e processamento de grandes volumes de dados um dos mais críticos. A busca por soluções para esse desafio é impulsionada pelo interesse das grandes companhias em prever o comportamento dos consumidores, permitindo melhores decisões de negócio, que resultam em um aumento nos lucros (TOTVS, 2023).

Os bancos de dados relacionais são uma solução eficiente e confiável para o armazenamento de informação, utilizando diversas tabelas relacionadas entre si (ORACLE, 2023). Os dados armazenados nesses bancos de dados relacionais, porém, não podem ser manipulados diretamente por linguagens de programação orientadas a objetos. Com isso, surge a necessidade do uso do mapeamento objeto-relacional, normalmente definido como:

> Object-Relational Mapping (ORM), em português, mapeamento objeto-relacional, é uma técnica para aproximar o paradigma de desenvolvimento de aplicações orientadas a objetos ao paradigma do banco de dados relacional. O uso da técnica de mapeamento objeto-relacional é realizado através de um mapeador objeto-relacional que geralmente é a biblioteca ou framework que ajuda no mapeamento e uso do banco de dados. (FONSECA, 2019)

Assim, as bibliotecas de mapeamento objeto-relacional atuam como uma camada de abstração, facilitando a utilização de bancos de dados relacionais ao gerar consultas SQL e converter registros armazenados em tabelas no banco de dados para objetos na linguagem escolhida.

O uso de ORM pode proporcionar um desenvolvimento mais rápido de aplicações, porém, sua existência como uma camada entre a aplicação e o banco de dados pode introduzir um custo no desempenho da aplicação. Com isso, torna-se necessário comparar o impacto adicional no desempenho decorrente dessa camada de mapeamento, de modo a determinar se esse custo extra justifica a utilização dessa técnica.

Este trabalho tem como objetivo geral verificar o impacto negativo no desempenho relacionado ao uso de bibliotecas de mapeamento objeto-relacional. Seu objetivo específico é comparar o tempo necessário para realização de operações em sistemas gerenciadores de bancos de dados entre diferentes padrões de implementação de ORMs. É apresentado também as vantagens apresentadas pelas ferramentas disponíveis nos *frameworks* que acompanham as ORMs escolhidas.

A técnica de mapeamento objeto-relacional pode ser implementada de diversas maneiras, sendo categorizada em diversos padrões diferentes, baseados em sua sintaxe. Dois dos principais padrões são representados neste trabalho, com o Django representando o padrão Active Record e o SqlAlchemy representando o padrão Data Mapper.

A escolha da linguagem Python, assim como do framework Django e da biblioteca SqlAlchemy, foi baseada em diversos critérios. O principal aspecto foi em relação à popularidade da linguagem, que é destacada como a principal linguagem do ano de 2023 de acordo com Cass (2023), que justifica essa popularidade da linguagem pela sua versatilidade. Outro aspecto foi em relação à disponibilidade de implementações populares de ORMs para ambos os padrões Data Mapper e Active Record, com o framework Django e a biblioteca SqlAlchemy sendo amplamente utilizados.

Com este estudo foi possível observar variações de desempenho entre as ORMs com base nas operações realizadas em um banco de dados. A diferença em relação ao acesso direto ao banco de dados foi considerável, com as bibliotecas ORM exigindo em torno de duas vezes o tempo para grande parte das operações realizadas. Ainda assim, as bibliotecas de mapeamento objeto-relacional ainda apresentam amplos casos de uso, sendo capazes de garantir ganhos consideráveis de produtividade, o que pode dificultar a decisão quanto ao uso dessas bibliotecas. Quanto aos padrões analisados, as diferenças encontradas foram limitadas à sintaxe dos comandos, visto que as consultas geradas pelas duas bibliotecas são compostas pelas mesmas instruções.

Este trabalho está organizado como a seguir. O Capítulo 2 aborda o referencial teórico, com os conceitos básicos necessários para a compreensão deste trabalho, além de trabalhos relacionados. No Capítulo 3, são definidos os testes realizados, assim como o modelo de dados utilizados no banco de dados. O Capítulo 4 descreve os resultados obtidos, e promove a discussão quanto ao desempenho das bibliotecas avaliadas. Por fim, o Capítulo 5 apresenta a conclusão do trabalho, com as considerações finais e discussão sobre trabalhos futuros.

2 REFERENCIAL TEÓRICO

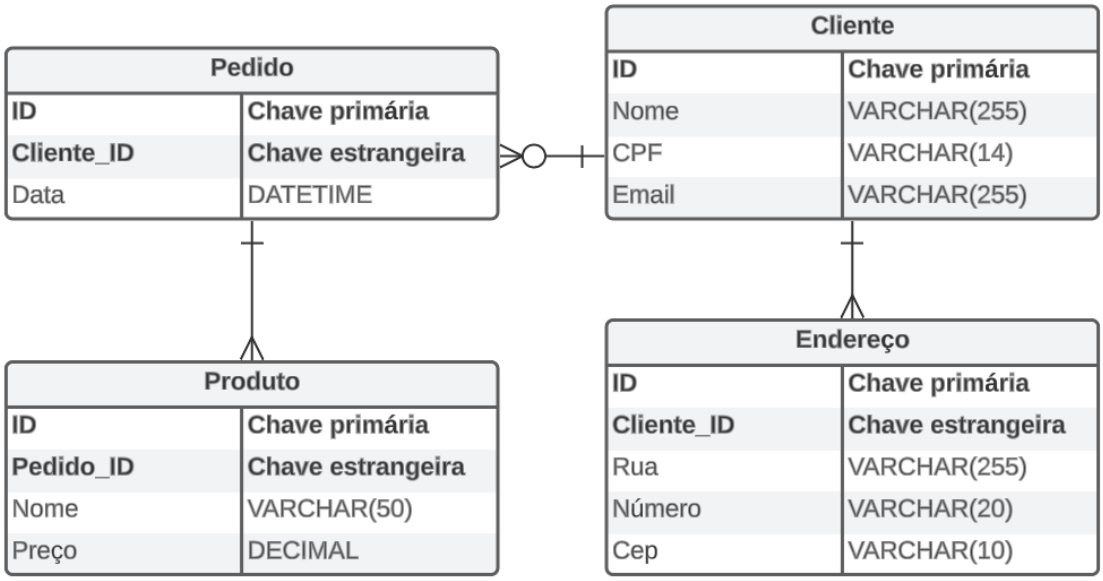
Neste capítulo, serão introduzidos os conceitos básicos necessários à compreensão deste trabalho. Serão discutidos tópicos sobre bancos de dados relacionais (Seção 2.1), modelos para implementação de ORMs (Seção 2.2) e divisão do mapeamento objeto-relacional em padrões (Seção 2.3).

2.1 Bancos de Dados

Um banco de dados é um conceito ligado não só à área de computação, sendo definido de maneira generalizada como uma coleção de dados relacionados, com esses dados representando fatos conhecidos, que podem ser registrados e que possuem um significado implícito (ELMASRI; NAVATHE, 2015, p.4).

Os bancos de dados podem ser divididos em vários modelos, e, segundo uma pesquisa realizada pelo Stack Overflow (2023), o modelo relacional permanece como o mais popular. Conforme Sumathi e Esakkirajan (2007), este modelo utiliza uma coleção de estruturas lógicas, no formato de tabelas, para representar dados e o relacionamento entre eles. A Figura 2.1 apresenta um diagrama UML que representa as tabelas de um . As tabelas devem apresentar uma coluna que atua como chave primária, ilustrado na Figura 2.1 como a primeira coluna de cada tabela, que recebe o nome da própria tabela, seguido do sufixo *ID*. Para definir os relacionamentos entre tabelas, deve-se adicionar uma coluna em uma das tabelas, referenciando a chave primária de outra tabela, como é o caso da coluna *Pedido_ID* na tabela *Produto* da Figura 2.1. Nesse contexto, a coluna que faz referência à chave primária na tabela *Pedido* é chamada de chave estrangeira.

Figura 2.1 – Representação simplificada de tabelas de um banco de dados relacional



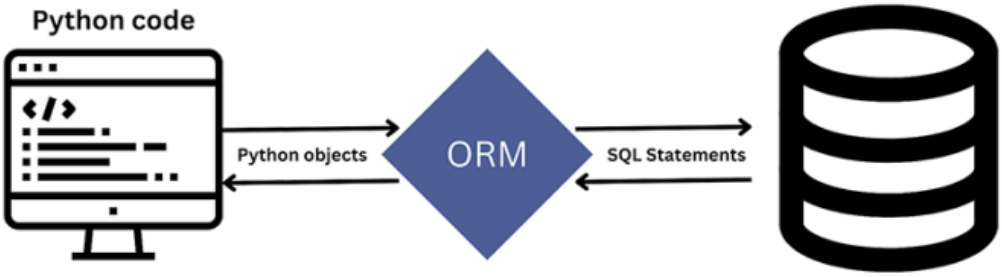
Fonte: Do autor (2023).

O acesso a dados armazenados em bancos de dados relacionais é feito através da Linguagem de Consulta Estruturada (SQL – Structured Query Language). Essa linguagem atua como um padrão entre diferentes implementações do modelo relacional, fornecendo comandos para a manipulação de registros.

2.2 Mapeamento Objeto-Relacional

O Mapeamento Objeto Relacional (ORM – Object-Relational Mapping) consiste no uso de uma camada de abstração entre um Sistema de Gerenciamento de Banco de Dados (SGBD) e uma aplicação desenvolvida em uma linguagem de programação orientada a objetos, como apresentado na Figura 2.2. Implementações dessa técnica não são limitadas apenas à simples tradução de entradas de um banco de dados para objetos da linguagem escolhida, apresentando também capacidade de geração de consultas SQL para acesso a essas entradas, geração de tabelas baseadas nos objetos criados, entre diversas funcionalidades.

Figura 2.2 – Representação simplificada de uma ORM em Python



Fonte: HAKIM (2023).

Com a difusão cada vez maior do paradigma orientado a objetos, em um mercado dominado por bancos de dados relacionais, as bibliotecas de mapeamento objeto-relacional se tornam cada vez mais necessárias no processo de desenvolvimento (DEVMEDIA, 2023). Como um dos focos de bibliotecas ORM é a simplificação do desenvolvimento, elas são normalmente utilizadas através de *frameworks*, que compartilham esse propósito.

Certos *frameworks*, conhecidos como "full stack frameworks", possuem todas as ferramentas necessárias para a criação de uma aplicação Web completa, fornecendo uma interface para o usuário, uma camada de processamento de dados, e, em certos casos, até mesmo um sistema gerenciador de banco de dados, se tornando uma escolha clara tanto para projetos simples quanto complexos.

Ao operar como uma camada extra na aplicação, bibliotecas ORM podem causar uma diminuição no desempenho, o que pode variar entre uma perda de poucos milissegundos até mesmo vários segundos, a depender do *framework* utilizado e, principalmente, da quantidade de dados sendo manipulados pela camada. Neste contexto, o desempenho do mapeamento objeto-relacional se tornou foco de diversos estudos.

Um estudo conduzido por Zmaranda et al. (2020) apresenta uma comparação entre três das principais ORMs do *framework* .NET, na qual *Entity*, *nHibernate* e *Dapper* tem sua performance analisada nas operações de CRUD (*Create, Read, Update, and Delete*). Uma das conclusões apresentadas neste estudo foi que nenhuma das ORMs analisadas se destacou em todos os aspectos, com o *nHibernate*, por exemplo, se sobressaindo apenas nas operações de busca e inserção. Uma possibilidade apresentada na pesquisa é a utilização de diferentes bibliotecas ORMs para diferentes operações, sendo possível utilizar a melhor alternativa para

cada operação. Entretanto, essa abordagem poderia introduzir complexidade desnecessária ao código, além de exigir do desenvolvedor conhecimento da sintaxe de múltiplas bibliotecas. A conclusão desse estudo ressalta como estratégia mais eficaz a escolha de uma única ORM, com base na operação mais frequente da aplicação.

A principal perda de desempenho, porém, pode não ser relacionada à ORM utilizada, mas sim ao uso inadequado das funções fornecidas por ela. Essa foi a base do estudo realizado por Chen et al. (2014), no qual foi proposto um *framework* capaz de detectar e priorizar padrões que apresentam perda de desempenho em sistemas que fazem uso de ORM. O artigo tem um foco em dois problemas: uso excessivo de dados e múltiplos acessos ao banco de dados dentro de laços. O estudo conclui apresentando um possível ganho médio de 35% no tempo de resposta do sistema após a correção dos sistemas analisados.

2.3 Padrões Disponíveis para ORMs

É comum a divisão de bibliotecas de mapeamento objeto-relacional em diferentes padrões, baseados na sintaxe empregada por cada implementação. Neste trabalho, foram escolhidos *frameworks* com ORMs que abrangem os dois padrões mais populares, Data Mapper e Active Record.

O padrão Data Mapper, conforme definido por Fowler et al. (2007), é uma camada de *software* que separa os objetos em memória daqueles salvos no banco de dados. Cada tabela do banco de dados é representada por uma classe, geralmente denominada *Entity*, e é acompanhada por outra, conhecida como Repository, responsável por executar operações no banco de dados.

Na Figura 2.3, é apresentado um exemplo simplificado de uma leitura na biblioteca SqlAlchemy, que utiliza o padrão Data Mapper. A classe *Filho*, definida na linha 2, representa a tabela *Filho* presente no banco de dados, que contém apenas colunas para o id e nome. Na linha 11 é definido o método de acesso que busca e retorna todos os filhos presentes no banco de dados através do objeto *repository*, definido na linha 9. É importante ressaltar que, em uma aplicação bem estruturada, a definição de modelos de dados e as funções de acesso às informações seriam separadas em arquivos diferentes.

Figura 2.3 – Exemplo de leitura do padrão Data Mapper

```
1  # Definição do modelo
2  class Filho(Base):
3      id = Column(Integer, primary_key=True)
4      nome = Column(String(255))
5
6  # Acesso aos dados:
7  app = Flask(__name__)
8  db = SQLAlchemy(app)
9  repository = db.session
10
11 def readFilho():
12     return repository.query(Filho).all()
```

Fonte: Do autor (2023).

Este foco em modularização, ainda que torne a codificação mais complexa, garante um código mais legível e de mais fácil manutenção. Sendo recomendado para projetos maiores e mais complexos, este padrão está presente em *frameworks*, como Hibernate, no Java, Entity Framework, na linguagem C#, e também na biblioteca SqlAlchemy, junto ao *framework* Flask, na linguagem Python.

O padrão Active Record, por sua vez, tem como objetivo facilitar o acesso ao banco de dados, consolidando a camada de acesso ao banco de dados nos objetos representantes das tabelas, geralmente denominados de *model*. Na Figura 2.4, é apresentado um exemplo de uma operação de acesso ao banco de dados utilizando o Django. Na linha 2 é definida a classe que representa a tabela *Filho*, que herda os métodos de acesso ao banco de dados da classe *models*, fornecida pelo Django. Na linha 7, é realizado o acesso ao banco de dados, e retornados todos os filhos presentes. Como o padrão não apresenta um objeto específico para o mapeamento, como o *Repository*, presente no Data Mapper, a *Model* deve ser idêntica à tabela que representa.

Figura 2.4 – Exemplo de leitura do padrão Active Record

```
1  # Model
2  class Filho(models.Model):
3      id = models.IntegerField(Integer, primary_key=True)
4      nome = models.CharField(max_length=255)
5
6  def readFilho():
7      return Filho.objects.all()
```

Fonte: Do autor (2023).

Fowler et al. (2007) recomenda o uso do padrão Active Record em aplicações simples, devido principalmente à necessidade de alinhar o objeto à tabela sendo representada, o que pode dificultar a representação de lógicas de negócio complexas e mudanças na estrutura do banco de dados. Ainda assim, esse padrão é implementado em *frameworks* amplamente utilizados, como Ruby on Rails, Laravel e Django. No Quadro 2.1, é apresentado um resumo comparando as principais diferenças entre os padrões Active Record e Data Mapper.

Quadro 2.1 – Comparativo entre os padrões Active Record e Data Mapper

Aspecto	**Data Mapper**	**Active Record**
Separação de responsabilidades	Define uma separação rigorosa entre as representações das tabelas do banco de dados e funções de manipulação de dados	Combina as operações de manipulação de dados no objeto responsável pela representação dos dados.
Complexidade	Utiliza uma sintaxe mais complexa, necessitando da instanciação e utilização de um objeto extra, que atua como a camada entre a aplicação e o banco de dados.	Apresenta uma sintaxe mais simples, necessitando apenas a criação dos modelos representantes das tabelas do banco de dados. As funções para acesso aos registros do banco de dados são fornecidas por padrão pelos modelos criados.
Cenário de uso	É recomendado para aplicações maiores e mais complexas por possuir uma maior modularização, resultando em um código mais legível	É recomendado para aplicações simples, por dificultar a representação de lógicas de negócio complexas, e unir a responsabilidade de acesso ao banco de dados à representação dos dados.
Principais representantes	Presente no SqlAlchemy, Hibernate e Entity Framework	Presente no Django, Laravel e Ruby on Rails

Fonte: Do autor (2023).

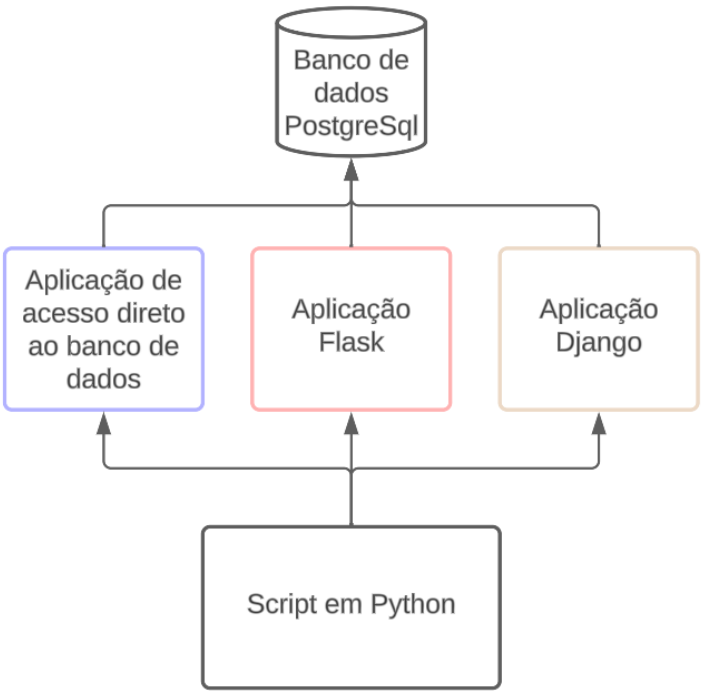
3 METODOLOGIA

Neste capítulo, é descrita a metodologia empregada no estudo para atingir os objetivos estabelecidos.

3.1 Cenários de Avaliação

Para avaliar o desempenho das bibliotecas ORM foram desenvolvidas três aplicações em Python: uma sem uso de *frameworks*, outra empregando Django (Active Record) e a terceira utilizando Flask (Data Mapper). Todas as aplicações acessam o mesmo banco de dados, com o mesmo modelo. A relação entre as aplicações é apresentada no diagrama da Figura 3.1.

Figura 3.1 – Diagrama das aplicações utilizadas



Fonte: Do autor (2023).

As aplicações em Django e Flask apresentam *endpoints* POST para cada operação CRUD, assim como *endpoints* diferentes para as tabelas que apresentam ou não chaves estrangeiras. Esses *endpoints* recebem o número de repetições a serem executadas, e retornam o tempo medido para cada operação. Um *script* foi elaborado para executar essas aplicações, uma por vez, realizar as requisições para os *endpoints* disponibilizados pelos *frameworks* e armazenar os valores obtidos em arquivos CSV.

A aplicação que executa consultas SQL através do psycopg2 opera de maneira distinta, apenas medindo o tempo das operações diretamente no banco de dados e armazenando os valores obtidos em um arquivo CSV. Não existe comunicação entre o *script* principal e a aplicação de acesso direto ao banco de dados, sendo o *script* responsável apenas pela execução da aplicação.

3.1.1 Modelo de Dados

A criação das tabelas no banco de dados foi realizada pelo Django, que também precisou criar tabelas para suas diversas funções, como as tabelas para o sistema de autenticação e para registro de suas migrações. A representação dos modelos no código em Python segue o paradigma de programação orientado a objetos, possibilitando assim o uso de conceitos como classes abstratas e herança, conforme ilustrado na Figura 3.2. A classe *Pessoa*, declarada como abstrata nas linhas 6 e 7, existe apenas no código em Python, não resultando na criação de uma tabela correspondente no banco de dados.

Figura 3.2 – Utilização de uma classe abstrata no Django

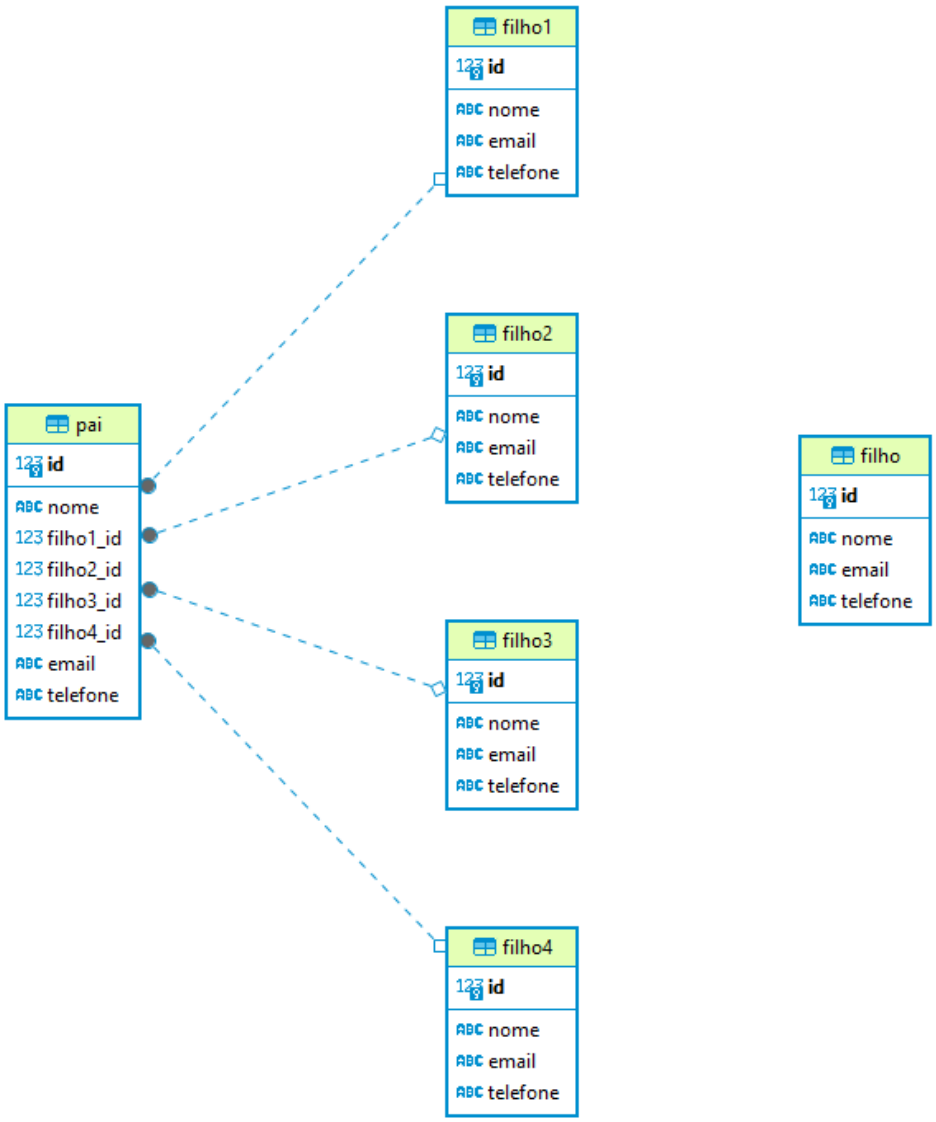
```
1  class Pessoa(models.Model):
2      nome = models.CharField(max_length=255)
3      email = models.CharField(max_length=255)
4      telefone = models.CharField(max_length=255)
5
6      class Meta:
7          abstract = True
8
9  class Pai(Pessoa):
10     id = models.IntegerField(primary_key=True)
11
12     filho1 = models.ForeignKey('Filho1', on_delete=models.DO_NOTHING)
13     filho2 = models.ForeignKey('Filho2', on_delete=models.DO_NOTHING)
14     filho3 = models.ForeignKey('Filho3', on_delete=models.DO_NOTHING)
15     filho4 = models.ForeignKey('Filho4', on_delete=models.DO_NOTHING)
16
17     class Meta:
18         db_table = "pai"
19
20 class Filho(Pessoa):
21     id = models.IntegerField(primary_key=True)
22
23     class Meta:
24         db_table = "filho"
```

Fonte: Do autor (2023).

Os testes tiveram um foco em duas tabelas distintas, uma tabela *pai* que, além dos atributos herdados da classe *pessoa*, apresenta quatro chaves estrangeiras, e uma tabela *filho*, que não possui chave estrangeira ou relacionamento com as outras tabelas. Essa divisão visava apresentar custo de performance associado à técnica de *Eager Loading*, que pode resultar em um significativo custo de desempenho, visto que carrega registros de múltiplas tabelas relacionadas, após o uso de operações de junção (*join*) ou através de múltiplas operações seguidas. Assim, uma operação de leitura que era esperada buscar em apenas uma tabela do banco de dados pode evoluir para uma operação que envolve várias tabelas diferentes, e, no pior dos casos, pode até mesmo envolver todas as tabelas de um banco de dados.

No diagrama apresentado na Figura 3.3, encontra-se a representação das tabelas utilizadas, onde está representada uma tabela *pai* relacionada a quatro tabelas diferentes, e uma tabela *filho* sem nenhum relacionamento. A modelagem utilizada limita um pai a apenas quatro filhos, cada um armazenado em tabelas diferentes, de modo a verificar a otimização na geração de consultas SQL das bibliotecas ORM. As operações na tabela *pai* interagem com cinco registros diferentes, manipulando um registro na tabela pai e quatro registros em diferentes tabelas filho, numeradas de um à quatro. Assim, é esperado que essas operações apresentem um tempo equivalente ou inferior a cinco operações na tabela *filho,* que interagem com apenas um registro. A possibilidade das bibliotecas ORM apresentarem uma relação de tempos inferior a cinco vezes entre as tabelas *pai* e *filho* existe, pois parte do tempo gasto em operações que manipulam dados está na comunicação entre a aplicação e o banco de dados, e em certas operações é possível realizar menos acessos ao banco de dados para manipular registros na tabela *pai* e tabelas relacionadas.

Figura 3.3 – Diagrama das tabelas utilizadas



Fonte: Do autor (2023).

3.2 Medição do Tempo de Execução de Consultas

Como o objetivo deste trabalho é a comparação dos tempos utilizados por processamentos das ORMs, o tempo utilizado nas chamadas dos *endpoints* é completamente desconsiderado. Assim, foi necessário realizar as medições dentro da própria aplicação, com os tempos obtidos sendo retornados como resposta da requisição para o *script* que realizou a chamada, para então ser gravado em um arquivo CSV.

Quanto ao tempo de acesso direto ao banco de dados, foi utilizado um *script* secundário, visto que a biblioteca Psycopg2 foi utilizada sem nenhum *framework* que permitisse a criação

de *endpoints*. Essa diferença, porém, não apresenta nenhuma discrepância nos resultados obtidos, uma vez que o tempo gravado só corresponde ao tempo necessário para a comunicação com o banco de dados e o processamento da consulta que foi definida.

Já é esperado que essa biblioteca tenha um desempenho melhor, visto que as ORMs escolhidas utilizam as mesmas operações fornecidas por ela para comunicação com um banco de dados PostgreSQL, além de realizar diversos processamentos adicionais. Ainda assim, a medição dos tempos de acesso direto ao banco de dados é essencial para compreensão do desempenho das ORMs, apresentando o tempo mínimo possível para cada operação, o que permite uma comparação de quanto tempo adicional é utilizado por cada operação quando se usa alguma ORM.

Para avaliação do tempo de execução foram realizadas 10 repetições de um ciclo de testes, com os resultados apresentados sendo uma média dos tempos obtidos, com um intervalo de confiança de 90%. Quanto às operações em si, foram feitas várias operações em sequência, de modo a mitigar a variância nos resultados obtidos, e, também, oferecer uma oportunidade de análise da escalabilidade dos sistemas analisados.

O ciclo de testes consistia na realização sequencial de operações de criação, leitura, edição e exclusão, em incrementos específicos. Inicialmente, foram realizadas mil execuções para cada operação, onde eram criados mil registros, que eram lidos, editados, e, por fim, excluídos. Eram registrados os tempos, que representavam todas as mil execuções de cada operação, e em seguida o processo era realizado novamente, com duas mil repetições, e, por fim, o ciclo era finalizado após a realização da medição do tempo utilizando quatro mil repetições. É importante ressaltar que, durante o experimento, não foi realizado o flush do cache do sistema gerenciador de bancos de dados utilizado (PostgreSql), portanto, os tempos obtidos podem se mostrar inferiores aos encontrados em um sistema real. Porém, como os tempos obtidos são comparados apenas entre si, a utilização do cache não impacta negativamente os resultados obtidos.

3.3 Configuração dos Frameworks e Bibliotecas

Nesta seção, são apresentadas as diferenças nas implementações entre os *frameworks* e a biblioteca utilizados, com um foco especial na operação de leitura, que difere em parte dos outros testes por requerer operações adicionais para seu funcionamento.

3.3.1 Django

O *framework* Django fornece uma variedade de ferramentas para auxílio no desenvolvimento de aplicações. Uma delas, a geração de tabelas, foi escolhida para a modelagem do banco de dados a ser usado nos testes. O sistema utilizado pelo Django utiliza um conceito de *migrations* para criar e alterar o banco de dados, sendo capaz de criar ou alterar o banco de dados com base no que é definido no modelo da aplicação. Conforme definido na documentação do *framework* Django (2023), o primeiro comando a ser realizado, definido como *makemigrations*, é responsável por registrar as mudanças realizadas, análogo a um *commit* de um sistema de versionamento, enquanto o comando *migrate* fica responsável por aplicar essas mudanças no banco de dados.

Conforme apresentado na Figura 3.4, para a medição do tempo foi utilizado o método time do Python, que retorna o tempo atual, permitindo o cálculo da diferença entre o tempo inicial e final da operação. Foi criado um *endpoint* POST para cada operação e tabela a ser testada, recebendo a quantidade de operações a serem realizadas, e retornando o tempo obtido, já convertido para milissegundos. O registro dos tempos obtidos é realizado através de outro *script*, que inicia a aplicação, e obtém os tempos ao realizar chamadas aos *endpoints* fornecidos.

A operação de leitura necessitou de uma atenção especial devido a uma otimização detectada em relação a chaves estrangeiras, onde os registros relacionados à chave estrangeira definida são obtidos apenas quando seus valores são utilizados. Assim, se tornou necessária a interação com os valores obtidos, representada na Figura 3.4 pela impressão na tela dos valores dos filhos. Para manter os resultados comparáveis, nos testes com apenas uma tabela, o valor do nome obtido também é impresso na tela, ainda que o método utilizado retorne a entrada desejada mesmo que ela não seja utilizada.

Figura 3.4 – Endpoint para medição do tempo de leitura do Django

```
@api_view(['POST'])
def readPai(request):
    data = json.loads(request.body)
    rep = data.get('rep', None)
    print("rep: ", rep)

    st = time.time() * 1000
    for i in range(rep):
        res = Pai.objects.get(id=i)
        print(res.filho1.nome,
              res.filho2.nome,
              res.filho3.nome,
              res.filho4.nome)
    et = time.time() * 1000

    return HttpResponse(et - st)
```

Fonte: Do autor (2023).

3.3.2 Flask

O *framework* Flask, por padrão, apresenta uma plataforma de desenvolvimento mais simples que o Django. Escolhido por sua popularidade e grande compatibilidade com a biblioteca SqlAlchemy, o *framework* possibilitou a criação de *endpoints* POST, permitindo a criação dos testes de forma similar aos realizados no Django. A biblioteca utilizada, Flask-SqlAlchemy (2023), aponta em sua documentação que não altera o funcionamento ou a utilização da biblioteca SqlAlchemy, atuando apenas como uma extensão de suporte para a mesma, e garantindo assim uma representação precisa da ORM.

A leitura da tabela *pai* apresentou o mesmo problema identificado no Django, onde os registros relacionados são recuperados do banco de dados apenas quando são utilizados, assim, como é possível perceber na Figura 3.5, foi realizada a impressão na tela dos dados obtidos, similar àquela realizada no *framework* Django. O método que realiza a impressão, embora acrescente um tempo considerável à operação de leitura, é consistente em ambos os *frameworks*, assegurando, desse modo, igualdade nos testes.

Figura 3.5 – Endpoint para medição do tempo de leitura no Flask

```
1  @app.route('/readPai', methods=['POST'])
2  def readPai():
3
4      data = request.get_json()
5      rep = data['rep']
6      print("rep: ", rep)
7
8      st = time.time() * 1000
9      for i in range(rep):
10         res = db.session.query(Pai).get(i)
11         print(res.filho1.nome,
12               res.filho2.nome,
13               res.filho3.nome,
14               res.filho4.nome)
15     et = time.time() * 1000
16
17     return jsonify(et - st), 200
```

Fonte: Do autor (2023).

3.3.3 Psycopg2

Em contraste às ORMs apresentadas, a biblioteca Psycopg2 executa comandos SQL definidos pelo desenvolvedor, e armazena o resultado obtido em uma tupla, independentemente da tabela que foi acessada. A biblioteca foi utilizada sem um *framework*, simulando um cenário comum, em que uma aplicação abdica das abstrações fornecidas por ORMs e *frameworks* em busca do desempenho máximo disponível.

Como os comandos devem ser definidos manualmente, foi necessário a criação de *queries* SQL que apresentam um bom desempenho, e que forneçam os mesmos resultados dos comandos gerados pelas ORMs. A operação de leitura com chaves estrangeiras apresentou a maior diferença entre as queries, pois os *frameworks* geram comandos diferentes buscando cada um dos filhos, enquanto a query utilizada emprega operações de *JOIN* para obter todos os dados desejados em apenas um *SELECT*.

Outro detalhe quanto a operação de leitura foi a adição da mesma impressão na tela presente nos *frameworks*, como é possível perceber na Figura 3.6. Diferentemente dos *frameworks*, a operação de busca utilizada retorna todos os valores relacionados, porém, a adição da impressão é necessária para garantir a coerência entre os testes.

Diferente das operações realizadas pelo Django ou SqlAlchemy e Flask, o tempo registrado para o Psycopg2 representa apenas o acesso ao banco de dados e recuperação das informações. A geração de consultas SQL não acontece, sendo utilizada uma consulta pré-definida, e o mapeamento dos dados recuperados não é realizado. Assim, o tempo obtido representa apenas o acesso ao banco de dados, e a diferença entre os tempos para as outras bibliotecas representa o tempo exigido pelas operações realizadas pela ORM.

Figura 3.6 – Endpoint para medição do tempo de leitura no Psycopg2

```
st = time.time() * 1000
for i in range(x):
    cursor.execute
    (
    f'SELECT * FROM pai
    LEFT OUTER JOIN filho1 AS filho1_1 ON filho1_1.id = pai.filho1_id
    LEFT OUTER JOIN filho2 AS filho2_1 ON filho2_1.id = pai.filho2_id
    LEFT OUTER JOIN filho3 AS filho3_1 ON filho3_1.id = pai.filho3_id
    LEFT OUTER JOIN filho4 AS filho4_1 ON filho4_1.id = pai.filho4_id
    WHERE pai.id = {i} LIMIT 1'
    )
    res = cursor.fetchone()
    print(res[9], res[13], res[17], res[21])
et = time.time() * 1000
linha.append([et - st])
```

Fonte: Do autor (2023).

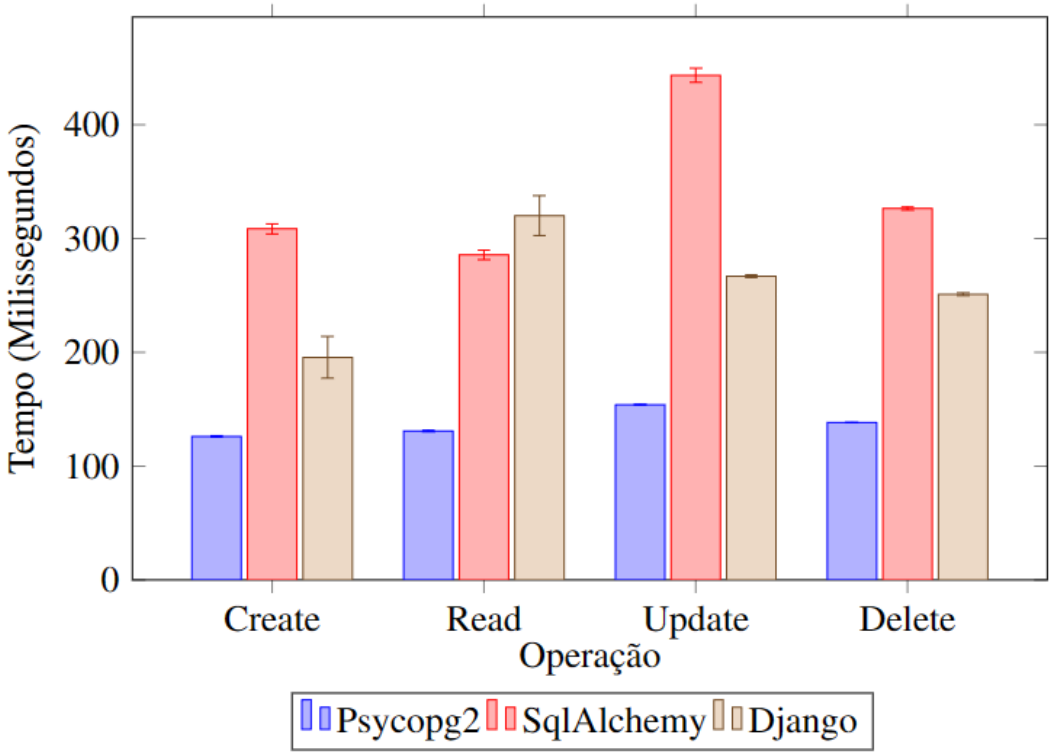
4 RESULTADOS E DISCUSSÃO

Nesta seção, serão apresentados os resultados obtidos, junto a uma análise dos tempos de execução registrados para as bibliotecas de mapeamento objeto-relacional e a biblioteca de acesso direto ao banco de dados. Serão discutidos os benefícios de ORMs, assim como casos de uso para os tipos de bibliotecas analisadas.

4.1 Avaliação do Desempenho

Os resultados, conforme apresentado na Figura 4.1, indicam uma vantagem significativa para a biblioteca Psycopg2 nas operações da tabela *filho*, onde o tempo gasto é, na maioria das operações, metade do tempo utilizado pelos *frameworks*. No entanto, é crucial ressaltar que o tempo exibido pelo Psycopg2 representa apenas o tempo de acesso ao banco de dados, visto que a biblioteca utiliza consultas SQL pré-definidas, e não trata os dados recebidos, que são armazenados em estruturas de dados genéricas.

Figura 4.1 – Tempos de execução de mil operações na tabela filho

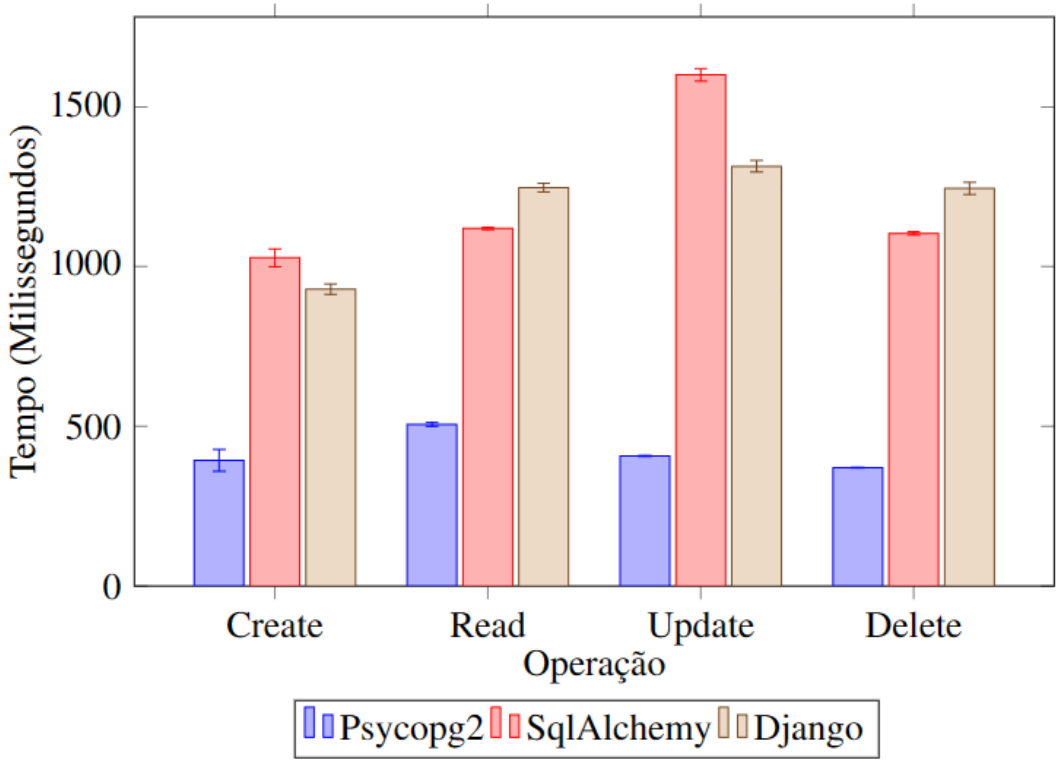


Fonte: Do autor (2023).

Na maioria das operações, observa-se que o tempo adicional necessário para a geração e mapeamento realizado pelas bibliotecas de ORM é considerável, frequentemente

ultrapassando o dobro do tempo total apresentado pelo acesso direto ao banco de dados. A operação com menor aumento em relação ao acesso direto foi o *create* na tabela *filho*, que, quando realizado pelo Django com seu padrão Active Record, exibiu um aumento de apenas 55% em relação ao acesso direto ao banco de dados, conforme apresentado na Figura 4.1. Já o maior aumento foi relacionado à operação update, realizada pela ORM de padrão Data Mapper SqlAlchemy, atualizando um registro na tabela *pai*, que apresentou um aumento de 293% em relação ao Psycopg2, conforme ilustrado na Figura 4.2.

Figura 4.2 – Tempos de execução de mil operações na tabela pai



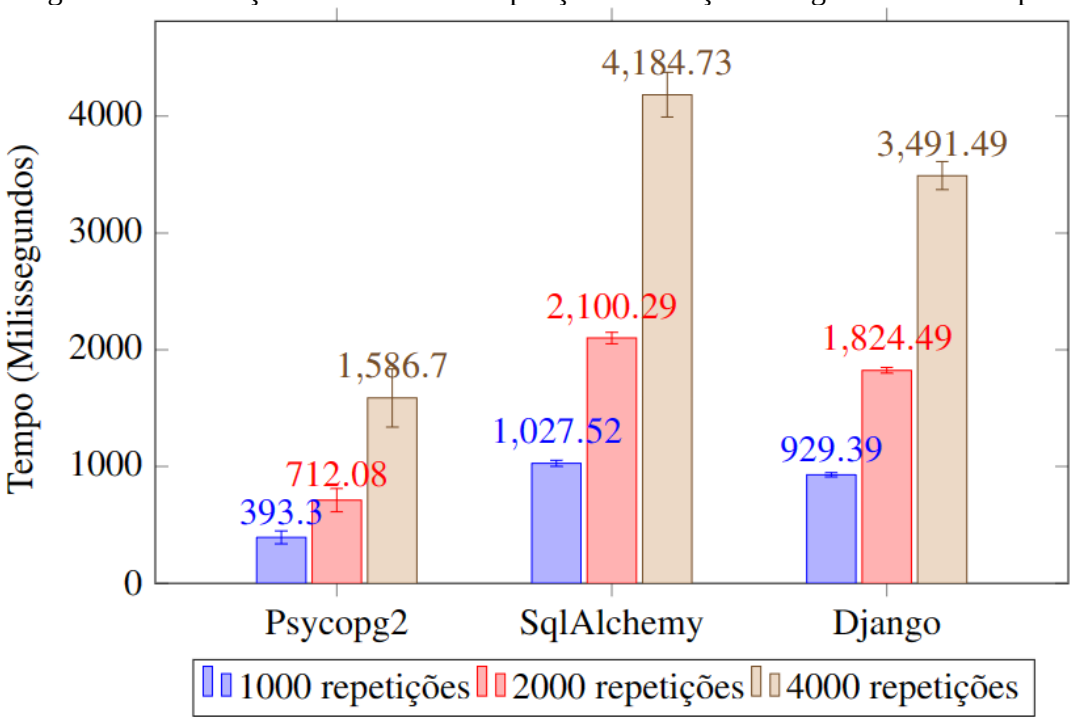
Fonte: Do autor (2023).

A comparação dos resultados entre as bibliotecas revela uma conclusão semelhante à encontrada por Zmaranda et al. (2020) em sua comparação de ORMs no *framework* .NET, na qual nenhuma ORM se destacou como a melhor em todas as operações. Os tempos variam de acordo com a operação sendo realizada e com a tabela sendo acessada e modificada, com o Django obtendo o melhor tempo na exclusão da tabela *filho*, enquanto o SqlAlchemy apresenta um tempo menor na exclusão da tabela *pai*. A criação e atualização também têm resultados diferentes entre as tabelas, com a diferença entre os *frameworks* diminuindo.

A Figura 4.3 destaca a diferença nos tempos para cada biblioteca em relação ao número de repetições, especificamente para a operação de criação de registros na tabela *pai*. Existe um padrão perceptível entre os tempos obtidos, visto que dobrar o número de operações resulta em

um aumento similar no tempo resultante. Esse padrão é consistente em todas as operações e tabelas analisados, não se limitando apenas à operação de criação.

Figura 4.3 – Variações no número de repetições na criação de registros na tabela pai



Fonte: Do autor (2023).

Os resultados se aproximam do que era esperado, dobrando junto ao número de execuções, porém ainda é possível detectar uma certa variação. A operação de criação na tabela *pai*, quando realizada pela biblioteca Psycopg2, apresentou a menor variação nos tempos entre diferentes repetições, com um incremento de apenas 81% nos tempos obtidos entre 1000 e 2000 repetições. Essa mesma operação também apresentou a maior diferença observada, com um aumento de 122% entre 2000 e 4000 repetições.

Os tempos obtidos, utilizando mil repetições, são representados na Tabela 4.1, com os menores tempos entre as bibliotecas ORM destacados, em verde para a tabela *pai*, e em amarelo para a tabela *filho*. Como as operações na tabela *pai* manipulam registros em diferentes tabelas, o aumento presente no tempo era esperado. A expectativa era um aumento máximo de cinco vezes ao comparar os tempos entre as tabelas *pai* e *filho*, com a possibilidade de tempos menores a depender da otimização presente nas consultas SQL geradas. O aumento obtido foi próximo ao esperado, com o Django apresentando um tempo cerca de 4.65 vezes maior para operações na tabela *pai* quando comparado ao tempo de operações na tabela *filho*, e o SqlAlchemy apresentando um aumento de cerca de 3.6 vezes no mesmo contexto. O acesso direto ao banco

de dados, utilizando comandos SQL otimizados, apresentou apenas cerca de 3.2 vezes mais tempo para operações na tabela *pai* em relação à tabela *filho.*

Tabela 4.1 – Diferença entre tempos da tabela pai e filho para mil repetições

Operação	**Psycopg2**		**SqlAlchemy**		**Django**	
	Filho	**Pai**	**Filho**	**Pai**	**Filho**	**Pai**
Create	126.34ms	457.54ms	310.63ms	1005.64ms	195.68ms	929.39ms
Read	128.25ms	545.23ms	275.19ms	1138.28ms	320.21ms	1247.34ms
Update	154.41ms	407.23ms	443.05ms	1667.40ms	266.98ms	1314.36ms
Delete	139.09ms	371.56ms	330.55ms	1101.80ms	251.13ms	1244.91ms

Fonte: Do autor (2023).

A operação *update* do Django apresentou a maior diferença de tempo entre as tabelas, necessitando cerca de 5 vezes mais tempo para realizar a exclusão na tabela *pai* quando comparada com o tempo de exclusão de um único registro na tabela filho. Já a operação de criação na tabela *filho*, realizada pelo SqlAlchemy apresentou o menor aumento de tempo entre as ORMs, necessitando de 3.23 vezes mais tempo em comparação à mesma operação na tabela *filho*. A menor diferença entre tempos foi representada pelo acesso direto ao banco de dados, com a biblioteca Psycopg2, visto que sua operação *update* necessitou de apenas 2.64 vezes mais tempo em comparação à manipulação de registros na tabela *filho*.

4.1.1 Especificidades da Implementação

As bibliotecas ORM oferecem diversos métodos para manipulação dos dados no banco de dados. Os métodos específicos utilizados podem ter um impacto direto na otimização das consultas geradas, como foi evidenciado pelos métodos de registro de dados fornecidos pelo Django. O método *save* é responsável pela criação de novos registros no banco de dados e a atualização de registros existentes, em casos onde o objeto fornecido ao método tem um identificador (id) definido. Essa dupla responsabilidade pode resultar em situações onde o id do objeto está definido, porém não existe uma entrada correspondente no banco de dados, fazendo assim, com que sejam geradas duas consultas, a primeira responsável pela atualização de um registro existente, e a segunda encarregada de criar um novo registro, caso a atualização não

obtenha sucesso. Portanto, optar pelo método *save* em lugar do método *create* para salvar novas entradas pode resultar em diferenças consideráveis no tempo necessário para a operação, como indicado na Tabela 4.2.

Tabela 4.2 – Comparação entre os métodos de inserção de dados do Django

Tabela	*Create*	*Save*
Filho	195.68ms	377.95ms
Pai	929.39ms	1780.38ms

Fonte: Do autor (2023).

A maioria dos métodos fornecidos pelas ORMs incorporam otimizações, como é o caso de operações de busca no banco de dados, que recuperam entradas conforme são utilizadas. Certas funções são otimizadas de modo a gerar consultas diferentes das esperadas, como é o caso da atualização e exclusão do SqlAlchemy, que utilizam das funções *query* e *filter* que realizam a busca por um registro. No entanto, caso os dados da entidade solicitada não sejam acessados ou alterados, é possível fazer uso dos métodos update ou delete sem que seja gerada uma *query* de busca no banco de dados, resultando em uma consulta bem otimizada, com um único acesso ao banco de dados. Existe, porém, um custo necessário para a verificação da necessidade de buscar ou não a entrada no banco de dados, o que pode explicar o tempo a mais gasto pelo SqlAlchemy em relação ao Django na atualização de registros.

O Django, com seu padrão Active Record, utiliza métodos mais intuitivos, permitindo a exclusão e edição sem a necessidade de métodos relacionados a consultas no banco de dados, através da criação de um objeto com apenas o *id* a ser deletado para o uso do método *delete*, ou um objeto com as novas informações para ser atualizado através do método *save*. Mesmo que o resultado seja o mesmo, com as modificações realizadas em apenas uma consulta, o código apresentado pelo Django se mostra mais legível.

4.2 Benefícios das Bibliotecas ORM

O principal propósito de uma ORM é atuar como uma camada de abstração, simplificando o acesso ao banco de dados e auxiliando no desenvolvimento de aplicações. Essa abstração, embora introduza um aumento considerável no tempo, permite ao desenvolvedor focar na lógica da aplicação em desenvolvimento, delegando a otimização de consultas à ORM. Assim, o tempo necessário para a criação e otimização de consultas SQL pode ser direcionado

para a otimização da lógica utilizada em operações mais complexas, resultando em uma aplicação refinada e, provavelmente, desenvolvida em menos tempo.

É importante ressaltar, porém, que existe uma curva de aprendizado para cada biblioteca disponível. Além disso, certas ORMs apresentam funções únicas ou implementam funções de maneira não convencional, como evidenciado pelo método *save* do Django. Em comparação, o acesso direto ao banco de dados exige um conhecimento mais profundo em relação ao necessário por bibliotecas de mapeamento objeto-relacional. Por outro lado, ao fazer uso da linguagem SQL, o desenvolvedor pode usar seu conhecimento em diferentes linguagens de programação e bancos de dados. O mesmo princípio pode ser aplicado a bibliotecas, que possuem similaridades suficientes para garantir ao desenvolvedor facilidade ao realizar uma transição entre eles.

Durante o desenvolvimento deste trabalho foram usadas diversas ferramentas fornecidas pelo Django, com o recurso mais importante sendo a geração e atualização de tabelas baseadas nos modelos definidos na aplicação. Essa função permite alterações rápidas, seja para pequenas correções ou para mudanças mais amplas no modelo do banco de dados. A criação e alteração do modelo do banco de dados através de *scripts* SQL demandam consideravelmente mais tempo do que necessário para definir os atributos desejados no código e executar o comando de migração.

Outra função disponível intrinsecamente relacionada ao fato de uma ORM atuar como uma camada intermediária entre a aplicação e o banco de dados é a capacidade de alterar o sistema gerenciador de bancos de dados sendo utilizado sem a necessidade de reescrever partes do código. Alguns *frameworks* fornecem um banco de dados em memória, que pode ser utilizado durante o desenvolvimento, sem impactar no funcionamento da aplicação em seu ambiente final, onde será utilizado um banco de dados convencional. Isso pode agilizar bastante o tempo de desenvolvimento de aplicações.

4.3 Cenários de Uso

Bibliotecas de acesso direto ao banco de dados, embora apresentem um menor tempo em relação às bibliotecas ORM disponíveis nos *frameworks*, apresentam cenários de uso bem mais limitados. Com a performance apresentada pelos dispositivos atuais, a perda de desempenho imposta pela ORM pode ser insignificante para certos tipos de aplicações. Mesmo

em aplicações com um enorme fluxo de usuários, é possível, por exemplo, fazer uso de sistemas de balanceamento de carga para distribuir as solicitações recebidas entre várias instâncias diferentes do sistema, e garantir um sistema rápido e estável.

Ainda assim, existem cenários em que o uso de acesso direto é justificado, como em sistemas de desempenho limitado, como é o caso em sistemas embarcados, ou em sistemas com restrições temporais, que necessitam de sistemas bem otimizados. Já as ORMs podem ser utilizadas em uma grande variedade de cenários, devido às diversas implementações disponíveis. Entre os *frameworks* analisados, o Django, com uma ampla gama de ferramentas disponíveis por padrão, pode ser utilizado para criar sistemas rapidamente, enquanto o SqlAlchemy pode ser usado no desenvolvimento de sistemas com dados complexos, ou que exigem uma otimização adicional, pois apresenta um foco no uso de consultas SQL personalizadas.

5 Conclusão

Os resultados deste estudo revelam um custo considerável ligado ao uso de *frameworks* e bibliotecas de mapeamento objeto-relacional, com a maior parte das operações necessitando o dobro do tempo utilizado pelo acesso direto ao banco de dados. Outro ponto importante está relacionado à diferença entre as ORMs, onde o Django apresentou resultados consideravelmente melhores que o SqlAlchemy nas operações na tabela *filho*, que é mais simples, enquanto na tabela *pai*, ainda que o Django continuou apresentando melhores tempos em metade das operações realizadas, sua vantagem é menor.

Certas operações no SqlAlchemy apresentaram uma sintaxe confusa, que pode justificar a diferença de tempo em relação ao Django. Certas operações, como a exclusão, fazem uso do método de busca no banco de dados, porém a operação não é realizada caso os dados não sejam utilizados, o que gera um custo adicional, visto que é necessário verificar o uso dos valores sendo deletados, em comparação ao método do Django, que fornece uma operação de exclusão diretamente em uma instância de um registro na tabela, que pode ter apenas o *id* definido.

Os padrões apresentados pelas ORMs não apresentam uma ligação à implementação escolhida para os métodos discutidos, visto que definem apenas a existência de um objeto que define métodos de comunicação com o banco de dados no caso do Data Mapper, ou a união desses métodos nos objetos representativos das tabelas do banco de dados. Assim, o fator mais provável por ser responsável pelo desempenho ruim apresentado pelo SqlAlchemy se torna o uso recomendado para a biblioteca, que é em bancos de dados complexos. Com esse foco em operações complexas, a ORM pode não apresentar uma boa otimização para a geração de consultas, visto que os métodos padrões normalmente são substituídos por consultas customizadas, para a realização de operações complexas.

A ferramenta de criação de tabelas fornecida pelo Django teve papel crucial no desenvolvimento do trabalho, possibilitando mudanças rápidas no modelo de dados utilizado, que podiam ser rapidamente adaptadas ao SqlAlchemy, graças ao uso de herança utilizado nos modelos de dados. O uso de *endpoints* HTTP também foi útil, permitindo definir e testar cada operação separadamente, fazendo também com que a aplicação utilizada na avaliação de desempenho fosse mais próxima de um caso real, ainda que o tempo medido desconsiderou o tempo utilizado para acesso aos *endpoints*. Não foi possível definir a melhor biblioteca em todos os aspectos, mas sim apresentar situações favoráveis para sua escolha, e o mesmo pode ser dito ao acesso direto ao banco de dados, que pode ser a melhor opção em certas situações.

Com os resultados obtidos sobre o tempo de processamento, é possível escolher o *framework* com melhor tempo para a operação mais utilizada na aplicação. No entanto, com modelos de dados e operações mais complexas, o tempo utilizado pelos *frameworks* pode variar. Um exemplo pode ser percebido na operação de exclusão, onde o *framework* com melhor tempo é baseado na existência de chaves estrangeiras no registro sendo alterado, com o Django sendo mais rápido ao excluir um registro na tabela *filho*, e o SqlAlchemy apresentando um melhor tempo de exclusão de múltiplos registros na tabela *pai*. Assim, a escolha de um *framework* em questão de tempo de processamento deve levar em consideração diversos aspectos complexos, como modelo do banco de dados, principais operações sendo realizadas, ferramentas fornecidas pelo *framework*, ou até mesmo qual banco de dados será utilizado.

REFERÊNCIAS

CASS, S. **The Top Programming Languages 2023**. 2023. Disponível em: <https:spectrum.ieee.org/the-top-programming-languages-2023>. Acessado em 12/12/2023.

CHEN, T.-H. et al. Detecting performance anti-patterns for applications developed using object-relational mapping. In: **Proceedings of the 36th international conference on software engineering.** New York, NY, USA: Association for Computing Machinery, 2014. (ICSE 2014), p. 1001–1012. ISBN 9781450327565. Disponível em: <https://doi.org/10.1145/2568225.2568259>.

DEVMEDIA. **Técnicas de mapeamento objeto relacional - Revista SQL Magazine 40.** 2023. Disponível em: <https://www.devmedia.com.br/tecnicas-de-mapeamento-objeto-relacional-revista-sql-magazine-40/6980>. Acessado em 16/11/2023.

DJANGO. **Migrations**. 2023. Disponível em: <https://docs.djangoproject.com/en/4.2/topics/migrations>. Acessado em 13/10/2023.

ELMASRI, R.; NAVATHE, S. B. **Fundamentals of Database Systems.** 7. ed. USA: Pearson, 2015.

FLASK-SQLALCHEMY. **Flask-SqlAlchemy Documentation.** 2023. Disponível em: <https://flask-sqlalchemy.palletsprojects.com/en/3.1.x/>. Acessado em 13/10/2023.

FONSECA, E. **O que é ORM?** 2019. Disponível em: <https://www.treinaweb.com.br/ blog/o-que-e-orm/>. Acessado em 16/11/2023.

FOWLER, M. et al. **Patterns of Enterprise Application Architecture.** USA: Addison Wesley, 2002.

HAKIM, A. A. **Exploring Django ORM.** 2023. Disponível em: <https://www.linkedin.com/pulse/exploring-django-orm-anas-al-hakim/>. Acessado em 05/11/2023.

ORACLE. **O que é um banco de dados relacional (RDBMS)?** 2023. Disponível em: <https://www.oracle.com/br/database/what-is-a-relational-database/>. Acessado em 16/11/2023.

STACK OVERFLOW. **2023 Developer Survey.** 2023. Disponível em: <https://survey.stackoverflow.co/2023>. Acessado em 16/11/2023.

SUMATHI, S.; ESAKKIRAJAN, S. **Fundamentals of Relational Database Management Systems.** 1. ed. USA: Springer, 2007.

TOTVS. **Big Data: o que é, como funciona e como aplicar?** 2023. Disponível em:<https://www.totvs.com/blog/inovacoes/big-data/>. Acessado em 16/11/2023.

ZMARANDA, D. et al. Performance comparison of crud methods using net object relational mappers: A case study. **International Journal of Advanced Computer Science and Applications,** The Science and Information Organization, v. 11, n. 1, 2020. Disponível em: <http://dx.doi.org/10.14569/IJACSA.2020.0110107>.