MODELAGEM EFICIENTE: UM ORM PYTHON DESACOPLADO E FOCADO EM SIMPLICIDADE.

João Pedro Gomes de Souza
Graduando em Banco de Dados pela Fatec Bauru
joao.souza190@fatec.sp.gov.br

Orientador: Luis Alexandre da Silva
E Docente na Fatec Bauru
luis.silva51 @fatec.sp.gov.br

RESUMO

Este artigo descreve o desenvolvimento de um ORM (*Object-Relational Mapping*) em *Python*, com foco na simplicidade, no desacoplamento de frameworks específicos e na eficiência operacional. Motivado pela necessidade de ferramentas mais leves e adaptáveis, o sistema proposto busca facilitar a interação com bancos de dados relacionais por meio de uma estrutura flexível e de rápida configuração, em contraste com soluções tradicionais. Este artigo oferece uma alternativa enxuta, priorizando clareza de uso e baixo acoplamento. Ao adotar uma abordagem minimalista, o sistema viabiliza a construção de aplicações mais leves, flexíveis e ajustadas a diferentes cenários, contribuindo para o desenvolvimento de soluções mais controladas e orientadas às necessidades específicas de cada projeto. Neste contexto, o presente artigo detalha desde a concepção do ORM até sua implementação prática, abordando suas principais características, como a definição de modelos, os mecanismos de filtragem e cache, e os experimentos de desempenho realizados. Demonstrando ser viável que é viável construir uma ferramenta enxuta e funcional, sem renunciar à produtividade ou da clareza na manipulação de dados relacionais, oferecendo uma alternativa eficiente às soluções mais robustas e acopladas disponíveis no ecossistema *Python*.

1 INTRODUÇÃO

Sistemas de gerenciamento de dados enfrentam o desafio de integrar linguagens orientadas a objetos com bancos de dados relacionais, conhecido como *object-relational impedance mismatch*. Esse problema surge das diferenças paradigmáticas entre as abordagens, afetando o mapeamento de dados, o desempenho e a manutenibilidade dos sistemas. Com a popularização dos ORMs, a gestão dessas diferenças tornou-se um aspecto central no desenvolvimento de software, influenciando decisões de design e implementação.

Apesar da adoção generalizada dos ORMs, persistem desafios como a compatibilidade entre modelos e tabelas relacionais, o desempenho das transações e a complexidade de modelagem. Esses obstáculos frequentemente resultam em erros de design e limitações nos sistemas.

Este artigo propõe um ORM desenvolvido em Python, que busca simplificar esses problemas com uma solução leve, intuitiva e configurável. Sem depender de frameworks robustos como *Django* ou *SQLAlchemy*, o ORM oferece uma abordagem minimalista e eficiente, com suporte a conversão de dados em JSON, cache de consultas, e definição dinâmica de modelos e filtros. Seu design prioriza simplicidade,

flexibilidade e controle refinado, tornando-o ideal para sistemas que exigem alta performance e baixa latência.

2 FUNDAMENTAÇÃO TÉORICA

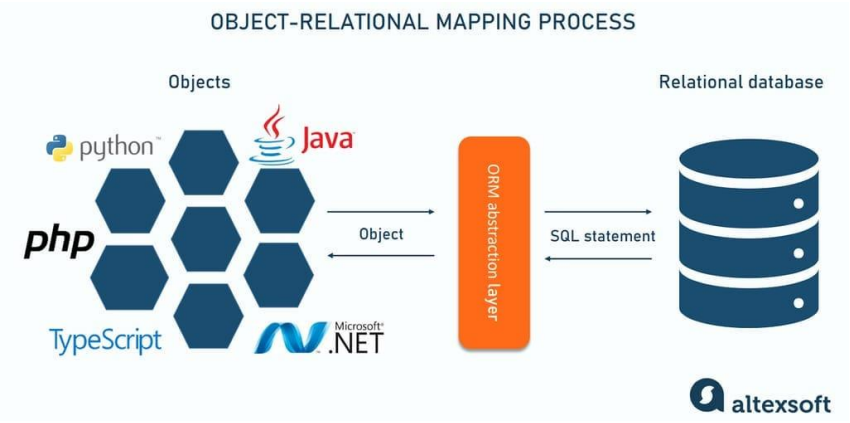
O desenvolvimento de sistemas modernos frequentemente esbarra no object-relational impedance mismatch, que descreve as dificuldades de alinhar conceitos de orientação a objetos com esquemas relacionais.

Barnes (2007) destaca que a representação de objetos do mundo real em estruturas relacionais aumenta a complexidade do sistema, afetando negativamente a performance e a manutenção. Enquanto a Programação orientada a objetos (POO), utiliza conceitos como herança e polimorfismo, bancos relacionais baseiam-se em tabelas, chaves e *joins*, dificultando a tradução entre esses modelos.

Os ORMs funcionam como uma camada de mapeamento entre objetos da aplicação e tabelas do banco de dados, permitindo que a lógica de negócios permaneça separada da lógica de persistência. Com isso, os desenvolvedores podem interagir com os dados utilizando objetos e seus métodos, ao invés de escrever diretamente instruções SQL, o que torna o código mais coeso e orientado a objetos (FOWLER, 2002).

Na Figura 1, observa-se a atuação do ORM como uma camada intermediária entre a lógica da aplicação e o sistema de armazenamento persistente. A ilustração evidencia como essa abstração permite a interação com os dados de forma transparente, dispensando o uso direto de instruções SQL por parte do desenvolvedor transparente.

Figura 1 Diagrama Objeto Relacional



Fonte: Object-Relational Mapping Tools: Pros, Cons and When to use.

Ferramentas como *SQLAlchemy* e *Django ORM* exemplificam abordagens distintas: enquanto o *Django ORM* prioriza a facilidade de uso e a rapidez no desenvolvimento, oferecendo uma experiência integrada ao ecossistema web, o *SQLAlchemy* se destaca por sua flexibilidade e controle mais detalhado, permitindo maior customização, porém com maior complexidade e uma curva de aprendizado mais acentuada (GEEKSFORGEEKS, 2024).

No contexto do desenvolvimento de software, Pimenta (2017) aponta que os ORMs têm se consolidado como uma abordagem eficaz para atenuar as discrepâncias entre o paradigma orientado a objetos e o modelo relacional, ao proporcionarem mecanismos que automatizam e simplificam o processo de persistência de dados. O autor desta que essas ferramentas, quando bem aplicadas, contribuem para a organização estrutural do código, manutenção facilitada e aumento da produtividade em projetos complexos, proporcionando vantagens como:

a) **Produtividade e Abstração:** Uma das principais vantagens dos ORMs é a produtividade que proporcionam ao desenvolvedor. Ao eliminar a necessidade de escrever SQL diretamente, os ORMs permitem que o desenvolvedor se concentre mais na lógica de negócios e menos na manipulação de dados aumentando sua produtividade e reduzindo chances de erros;

b) **Facilidade na modelagem**: a capacidade de representar tabelas de um banco de dados relacional como classes, permitindo que os desenvolvedores aproveitem plenamente as vantagens da programação orientada a objetos. Isso inclui:

- Encapsulamento: Os dados das tabelas podem ser encapsulados em atributos privados ou protegidos, com acesso mediado por métodos públicos (*getters* e *setters*). Isso garante maior controle sobre como os dados são acessados e modificados;

- Herança: É possível modelar tabelas relacionadas utilizando herança de classes. Por exemplo, uma tabela `Usuario` pode ser uma classe base, e tabelas específicas como `Admin` e `Cliente` podem herdar dessa base, evitando duplicação de lógica e facilitando a manutenção;

- Reuso de Código: Métodos comuns, como validações ou operações específicas, podem ser definidos em uma classe base e reutilizados em subclasses, promovendo modularidade e consistência;

- Polimorfismo: O ORM permite o uso de relações polimórficas, em que um mesmo atributo ou método pode ser aplicado a diferentes classes relacionadas, simplificando operações que envolvem múltiplas tabelas com estrutura similar;

- Associações como Propriedades: Relações entre tabelas, como **um-para-muitos** e **muitos-para-muitos**, podem ser mapeadas diretamente como propriedades de classe, tornando-as intuitivas para navegação;

- Integração com Padrões de Projetos: ORMs permitem implementar facilmente padrões de design, como *Repository e Unit of Work*, um dos principais Padrões de projetos que buscam a separação de responsabilidades e facilitando testes e manutenção do código

- Persistência de Dados: Transforma objetos da aplicação em registros no banco de dados, mantendo a integridade entre os dois mundos;

- Abstração de Consultas: Permite construir consultas complexas utilizando sintaxes orientadas a objetos em vez de SQL puro;

- **Gerenciamento de Relacionamentos**: Representa relações entre tabelas (um-para-um, um-para-muitos, muitos-para-muitos) através de referências entre objetos.

- **Facilidade de Uso**: Por se integrarem diretamente à linguagem de programação, os ORMs tornam a manipulação de dados e o controle do fluxo mais intuitivos e ágeis, o que melhora significativamente a experiência do desenvolvedor.

Embora os ORMs ofereçam conveniência ao abstrair o acesso ao banco de dados, não constituem uma solução universal para todos os cenários. Em contextos com consultas complexas ou grandes volumes de dados, seu uso pode degradar o desempenho, gerando múltiplas consultas desnecessárias e dificultando otimizações — especialmente em aplicações de maior escala, nas quais a eficiência das interações com o banco de dados é crucial (AppJot, 2023).

Conforme apontado por Colley, Stainer e Asaduzzaman (2017), o uso de ORMs pode introduzir uma sobrecarga de desempenho significativa em comparação com a execução direta de SQL. A Figura 2 exemplifica essa questão, comparando tempos de execução com e sem ORM. Os dados indicam que operações de busca (com cláusula WHERE) e consultas simples (SELECT sem WHERE) são particularmente impactadas, sugerindo limitações dos ORMs em cenários que demandam alta eficiência no processamento de dados.

Figura 2 Média de impacto de um ORM

Task	*Source*	*Logical reads*	*Physical reads*	*Parse / compile (ms)*	*Elapsed (ms)*
Add	ORM	2	0	0	13
	Non-ORM	-	-	-	-
List	ORM	2	0	6	135
	Non-ORM	2	1	3	53
Edit	ORM	2	0	4	0
	Non-ORM	-	-	-	-
Search	ORM	1	4	4	290
	Non-ORM	2	0	1	118
Delete	ORM	19	0	7	19
	Non-ORM	-	-	-	-

Fonte: The Impact of Object-Relational Mapping Frameworks on Relational Query Performance, 2017

Essas vantagens tornam os ORMs ferramentas valiosas no desenvolvimento moderno, promovendo produtividade, organização e facilidade de manutenção do código. Entretanto, é importante reconhecer que a abstração oferecida pode acarretar

implicações em termos de desempenho, especialmente em cenários que exigem maior controle sobre as operações realizadas no banco de dados.

No ecossistema *Python*, muitos dos ORMs mais utilizados apresentam limitações que podem dificultar seu uso em cenários específicos. Por exemplo, o *Django ORM* é fortemente integrado ao próprio framework, o que limita seu uso em contextos fora dele. Em contraste, o *SQLAlchemy* oferece uma flexibilidade muito maior, porém isso vem acompanhado de uma curva de aprendizado mais íngreme, o que pode torná-lo excessivamente complexo para projetos menores ou menos exigentes (GETACHEW, 2023).

Diante desse panorama, o ORM proposto neste artigo científico busca equilibrar simplicidade, desempenho e flexibilidade. Trata-se de uma alternativa leve, desacoplada e eficiente, ideal para aplicações que demandam performance, integração fluida com a programação orientada a objetos (POO) e facilidade de uso.

3 METODOLOGIAS E DESENVOLVIMENTO

A metodologia adotada combina abordagens qualitativas e quantitativas, garantindo uma análise das necessidades do sistema, bem como a validação de sua eficiência por meio de experimentos e testes comparativos. O desenvolvimento foi baseado em um processo iterativo e incremental, permitindo ajustes contínuos conforme novas descobertas forem feitas ao longo da implementação. As principais etapas envolvem:

a) **Análise de requisitos**: Levantamento das necessidades do projeto, definição das principais funcionalidades e comparação com soluções existentes;
b) **Projeto da arquitetura**: Definição da estrutura do ORM, incluindo seu modelo de funcionamento, interfaces e integração com diferentes bancos de dados;
c) **Implementação:** Desenvolvimento do núcleo do ORM, focando na abstração mínima necessária para simplificar operações comuns sem comprometer o desempenho;
d) **Testes e validação**: Realização de testes unitários, benchmarks de desempenho e comparações com ORMs já consolidados no mercado;
e) **Documentação**: Elaboração de guias de uso e referências técnicas para facilitar a adoção da ferramenta por desenvolvedores.

Para compreender as principais limitações e desafios dos ORMs existentes, foi realizada uma análise detalhada de ferramentas amplamente utilizadas, como o *Django ORM* e o *SQLAlchemy*, incluindo *benchmarks* preliminares, identificou limitações relacionadas à complexidade de configuração/uso e à falta de flexibilidade:

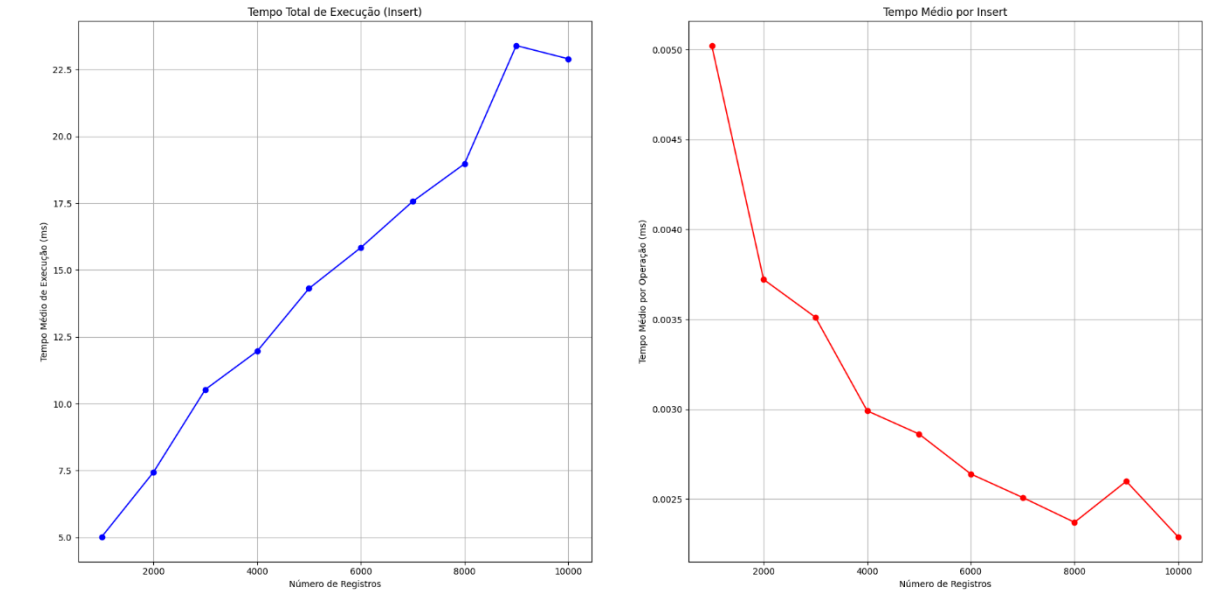
- **Django ORM:** Apresenta forte acoplamento ao *framework* web *Django*, dificultando seu uso fora desse ecossistema e limitando a flexibilidade para consultas avançadas ou integração com bancos de dados não relacionais;

- **SQLAlchemy:** Embora muito poderoso, possui uma curva de aprendizado acentuada e exige configuração detalhada, o que pode torná-lo excessivamente complexo para aplicações menores ou para desenvolvedores que buscam uma solução mais direta.

A avaliação do desempenho operacional foi realizada por meio de experimentos comparativos. O objetivo central foi observar a performance do *SQLAlchemy*, uma ferramenta consolidada, e compará-la diretamente com o ORM proposto neste artigo sob condições idênticas. Para assegurar a reprodutibilidade e eliminar variáveis externas, os testes foram executados utilizando o mesmo SGBD (*PostgreSQL*) operando dentro de containers *Docker*. Esta abordagem garante que cada execução ocorra em um ambiente isolado e recém-criado, eliminando assim quaisquer efeitos residuais de cache do SGBD entre os testes e mitigando variações decorrentes de flutuações nos recursos da máquina. Foram executadas as mesmas operações (inserção e exclusão) em ambos os ORMs, processando lotes de 1.000 até 10.000 registros, permitindo analisar a eficiência do ORM desenvolvido frente a uma linha de base estabelecida. Os testes do ORM proposto se encontra na seção resultados. Detalhes completos dos testes estão disponíveis no repositório do projeto.

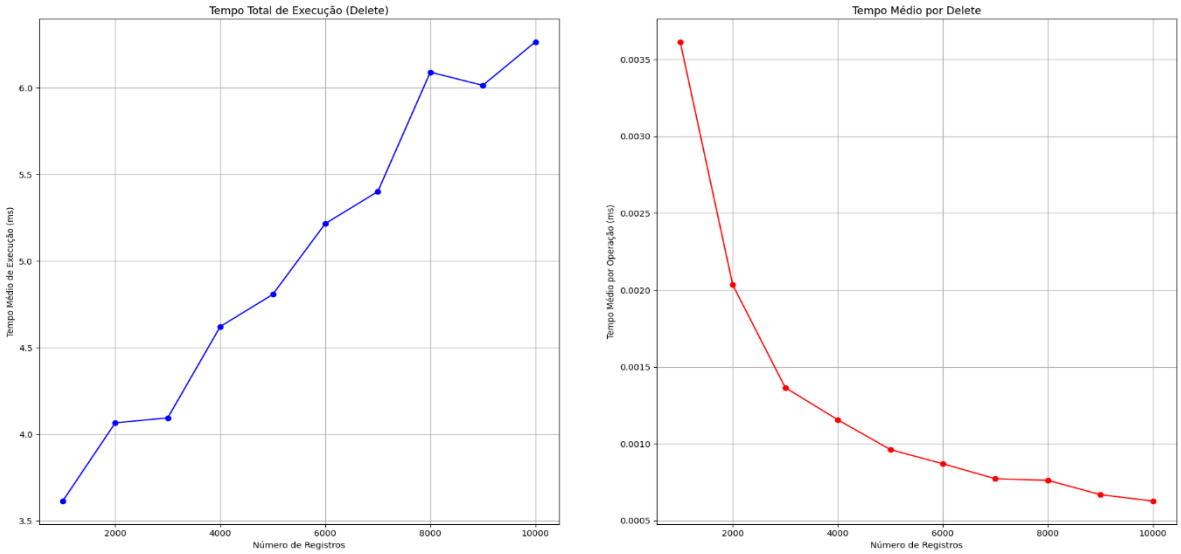
As Figuras 3 (Inserção) e 4 (Exclusão) ilustram o desempenho do *SQLAlchemy* neste ambiente controlado. Em ambos os cenários, verifica-se que, embora o tempo total de execução aumente com o número de registros, o tempo médio por operação diminui significativamente. Tal comportamento indica que o *SQLAlchemy*, em conjunto com o *PostgreSQL*, aplica otimizações eficazes, como processamento em lote ou gerenciamento otimizado de transações, resultando em maior eficiência por operação à medida que o volume de dados cresce.

Figura 3 SQLAlchemy Benchmark de insert



Fonte: O autor, execução completa disponível no código de benchmarks do projeto.

Figura 4 SQLAlchemy Benchmark de delete



Fonte: O autor, execução completa disponível no código de benchmarks do projeto.

Esses resultados obtidos servirão como base comparativa para a avaliação do desempenho do ORM proposto neste artigo. A partir deles, é possível observar os impactos e o desempenho de um ORM amplamente consolidado no ecossistema tecnológico, como o *SQLAlchemy*, em diferentes cenários de carga. Essa análise inicial fornece um referencial sólido que permitirá, em etapas posteriores, contrastar as soluções adotadas pelo ORM desenvolvido neste artigo , evidenciando suas particularidades, vantagens e possíveis limitações em relação às abordagens tradicionais.

Com base nos pontos negativos identificados nos ORMs *Python* existentes, o projeto proposto foi elaborado com foco em superar essas limitações, oferecendo uma abordagem mais flexível, leve e intuitiva. O foco principal está na simplicidade da configuração e no baixo acoplamento com o restante da aplicação, eliminando a necessidade de arquivos de configuração extensos ou mapeamentos complexos.

Adicionalmente, o ORM conta, em sua versão inicial, com suporte aos principais sistemas gerenciadores de banco de dados relacionais — *PostgreSQL*, *MySQL* e *SQLite* — garantindo ampla compatibilidade e versatilidade para diferentes ambientes de desenvolvimento e produção.

Além disso, o ORM visa simplificar o uso em projetos de pequeno e médio porte, como alternativa a soluções mais complexas. Propõe-se uma curva de aprendizado suave e interface direta, incluindo conversão nativa para *JSON* e modelos dinâmicos, tornando-o flexível tanto para aplicações simples quanto para cenários que exigem maior controle.

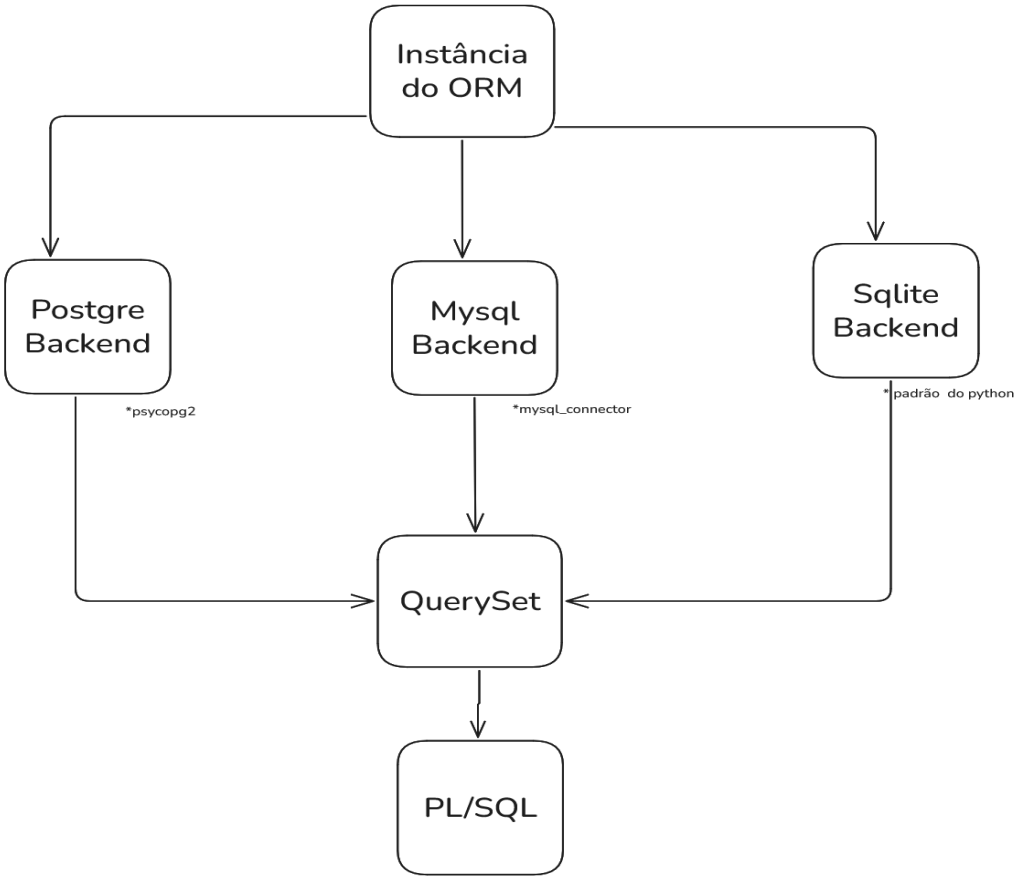
A implementação do ORM seguiu metodologias ágeis, com entregas incrementais e validações contínuas, sempre orientada pelas melhores práticas de desenvolvimento de software, dentre essas metodologias, destacam-se:

a) **Estrutura Modular**: Arquitetura modular com responsabilidades bem definidas, facilitando manutenção, expansão e reutilização de componentes
b) **Orientação a Testes (TDD)**: *Test-Driven Development* **(**TDD) adotado para garantir robustez, com testes cobrindo conexões a múltiplos SGBDs, otimização de consultas, validação de cache e benchmarks de desempenho.
c) **Versionamento**: Versionamento com *Git* e *GitHub* para controle rigoroso de alterações e facilitação da colaboração entre desenvolvedores.
d) **Gerenciamento de Dependências e Versionamento**: *Poetry* utilizado para gerenciar dependências e gerar o pacote distribuível, garantindo fácil instalação, controle de versões e integração em outros projetos *Python*.

No projeto, optou-se por utilizar bibliotecas nativas de SGBDs (como *psycopg2* para *PostgreSQL* e *mysql-connector-python* para MySQL) em vez de uma solução genérica como o *Open Database Connectivity* (ODBC). Essa decisão foi guiada por requisitos específicos de desempenho, funcionalidade e simplicidade de implementação.

A figura 5 demonstra um diagrama de como o ORM implementa uma camada de abstração que engloba tanto os diferentes SGBDs quanto suas respectivas bibliotecas de conexão nativas. Essa abordagem permite que o ORM gerencie as particularidades de cada banco, garantindo uma interface unificada para o acesso e manipulação dos dados, além de facilitar a manutenção e a escalabilidade do sistema.

Figura 5 Diagrama de entidades do ORM



Fonte: O autor.

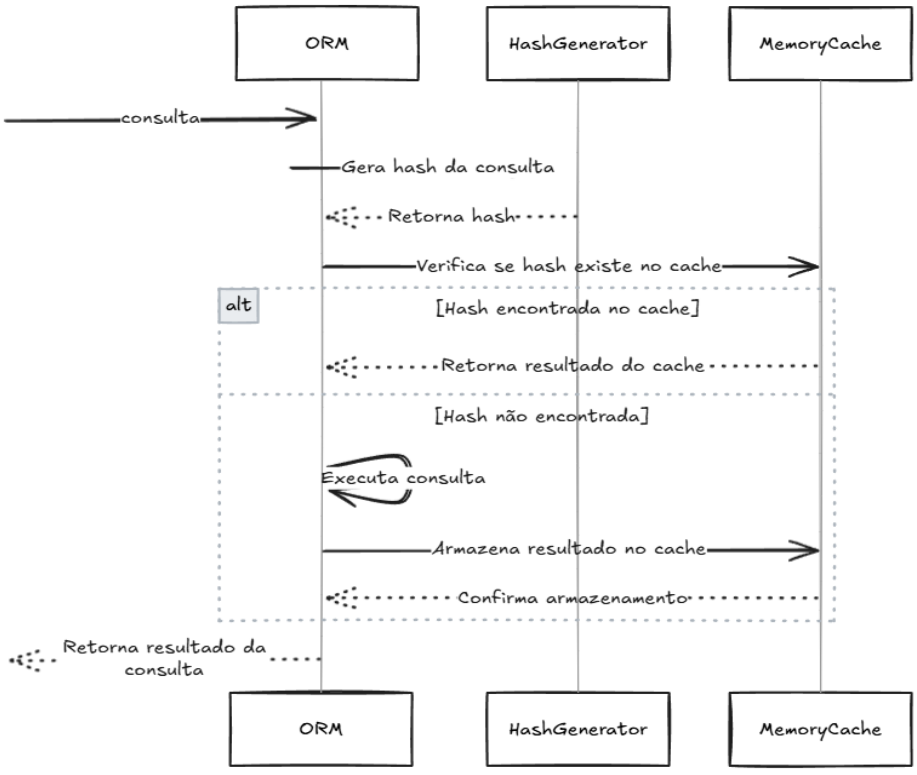
Considerando que cada SGBD adota abstrações próprias e mecanismos distintos de comunicação, o uso direto de bibliotecas nativas mostrou-se mais apropriado, proporcionando maior controle sobre as operações e uma integração mais eficiente com a camada de abstração proposta.

A adoção de uma arquitetura modular e desacoplada também possibilita a integração de um mecanismo de cache mais eficiente diretamente na camada de abstração do ORM. Esse sistema de cache tem como principal finalidade otimizar o desempenho das consultas, reduzindo acessos desnecessários ao banco de dados e, consequentemente, melhorando o tempo de resposta da aplicação.

Para isso, propõe-se a utilização de uma camada única de cache em memória, com funcionalidades configuráveis pelo usuário. Os principais aspectos desse mecanismo estão descritos a seguir:

a) **Tamanho do cache:** Na instância do ORM, o usuário poderá passar o valor máximo que será suportado no cache, esse valor pode ser tanto por número de consultas ou Megabytes de dados;
b) **Camada de Cache na Sessão:** Para otimizar consultas repetidas, o ORM implementa um cache em memória ativo durante a sessão do usuário. Conforme ilustrado no diagrama de sequência da Figura 6, ao receber uma consulta, o ORM gera um *hash* único para ela e verifica sua presença no cache. Se o *hash* for encontrado (cache *hit*), o resultado previamente armazenado é retornado diretamente, evitando o acesso ao banco de dados. Caso contrário (cache *miss*), a consulta é executada no banco, seu resultado é armazenado no cache associado ao *hash* e, então, entregue ao solicitante. Este cache de sessão é automaticamente invalidado ao término da sessão para liberar recursos.

Figura 6 Diagrama da consulta em cache



Fonte: O autor

O maior benefício dessa abordagem é a melhoria significativa na performance, já que a leitura de dados diretamente da memória é muito mais rápida do que a execução de uma consulta no banco de dados em disco. O cache em memória irá reduzir o tempo de resposta para o usuário, especialmente em casos em que múltiplas requisições semelhantes são feitas durante a mesma sessão. Ao eliminar a necessidade de fazer a mesma consulta repetidamente ao banco de dados, o ORM consegue oferecer um desempenho mais eficiente, especialmente em sistemas com alto volume de leituras ou consultas que não necessitam de dados atualizados a cada requisição.

4. RESULTADOS

O exemplo a seguir destaca aspectos essenciais do funcionamento do *PylightORM* e sua integração com o banco de dados, evidenciando sua simplicidade, flexibilidade e eficiência. Devido às limitações deste artigo, nem todas as funcionalidades implementadas puderam ser exploradas.

A implementação do ORM se inicia pela definição dos modelos de dados. Cada entidade de domínio é representada como uma classe Python que herda de *Model*, permitindo que o mapeamento objeto-relacional ocorra de maneira automática.

A figura 7 demonstra a definição da entidade Pessoa, que contém campos com diferentes tipos de dados primitivos:

Figura 7 Criação do Modelo.

```
class Pessoa(Model):

        id = IDField(auto_increment=True)

        nome = CharField(length=50)

        numero = IntegerField(not_null=True)

        data = DateField(not_null=True, default="2021-10-10")

        ativo = BooleanField(not_null=True, default=True)
```

Fonte: O autor.

Cada tipo de dado do modelo é mapeado para um tipo de coluna no banco de dados através dos Fields:

a) **IDField:** Define um campo inteiro de chave primária auto incrementável**;**
b) **CharField**: Representa dados textuais, com limitação de tamanho;
c) **DateField**: Específica o campo de data;
d) **BooleanField**: Define campos que armazenam valores verdadeiro/falso;
e) **IntegerField**: Mapeia números inteiros.

Após definir os modelos, o próximo passo é configurar o ORM com os parâmetros de conexão ao banco de dados. O código da figura 8 ilustra essa configuração utilizando o *PostgreSQL* como sistema gerenciador. São definidos o tipo de SGBD (*backend*), endereço do host, porta, nome do banco, usuário e senha. Esses dados permitem que o ORM estabeleça a comunicação com o banco para executar operações como criação de tabelas, inserções e consultas:

Figura 8 Instanciando o ORM.

```
from main.main import SwifitORM

from main.session import Session


orm = Pylight(
    backend="postgres",
    host="localhost",
    port=5432,
    database="dbteste",
    user="postgres",
    password="postgres",
)
```

Fonte: O autor.

Após configurar a conexão com o banco, o código a seguir demonstra como o ORM utiliza uma instância de *Session* para criar automaticamente a tabela correspondente ao modelo definido. Esse processo gera dinamicamente o comando SQL necessário, com base na estrutura da classe Pessoa:

Figura 9 Criando a tabela.

```
with Session(orm) as sessao:
    sessao.create_table(Pessoa)
```

Fonte: O autor.

O código abaixo da Figura 10, mostra como criar e inserir um novo registro no banco utilizando o método *create* da própria classe modelo. Esse método retorna uma instância do objeto com os valores definidos, pronta para ser adicionada à sessão e persistida no banco:

Figura 10 Criação de registro no modelo

```
nova_pessoa = Pessoa.create(
    nome="UsuarioTeste",
    data="2021-10-10",
    ativo=True,
    numero=10
)
sessao.add(nova_pessoa, commit=True)
```

Fonte: O autor.

O parâmetro *commit=True* garante que a transação seja finalizada e os dados sejam salvos imediatamente no banco.

A figura 11 demonstra a alteração de um atributo diretamente no objeto em memória. Em seguida, utiliza-se o método update para refletir essa alteração no banco de dados. Isso evidencia a separação entre o estado do objeto local e o persistido:

Figura 11 Alteração de objeto em memória

```
nova_pessoa.nome = "UsuarioAtualizado"

print(nova_pessoa.nome)

sessao.update(nova_pessoa, commit=True)
```

Fonte: O autor.

A alteração do objeto em memória evidencia a separação entre o estado local e o estado persistido no banco de dados. Essa característica é essencial em ORMs modernos, pois permite que o desenvolvedor modifique os atributos de um objeto sem que essas mudanças sejam imediatamente refletidas na base de dados. O envio explícito de um comando, como o update, é necessário para que o estado atualizado seja persistido. Isso confere maior controle transacional, permitindo operações mais seguras e previsíveis no gerenciamento dos dados.

O ORM também facilita a recuperação de dados, seja através de consultas completas ou por filtros específicos. Na figura 12, é demonstrado um exemplo de como recuperar todos os registros da tabela associada ao modelo Pessoa. O método *select_all* retorna uma lista de objetos de Pessoa, permitindo que cada registro seja manipulado como uma instância da classe modelo:

Figura 12 Extração de registros do modelo.

```
pessoas = sessao.select_all(Pessoa)
for pessoa in pessoas:
    print(pessoa.nome)
    print(pessoa.data)
```

Fonte: O autor.

Onde em cada valor do loop de repetição *for*, o usuário terá um objeto do modelo Pessoa já com todos seus métodos e atributos.

O ORM também permite a realização de buscas específicas por meio do método *Find*. Quando parâmetros são passados diretamente para a função, os registros são localizados com base em valores fornecidos para os campos correspondentes, como no exemplo da figura 13:

Figura 13 Extração de registros com filtros

```
joao_silva = session.find(

    Pessoa, nome="joao silva", ativo=True)
```

Fonte: O autor.

Nesse caso, busca-se por registros cujo nome seja "joao silva" e que estejam ativos. Para consultas mais elaboradas, o método *Find* também aceita filtros compostos, utilizando classes auxiliares como *In* e *Like*. Essas classes abstraem operações comuns do SQL e permitem construir consultas mais detalhadas, como demonstrado no exemplo da figura 14 abaixo:

Figura 14 Extração de registros com filtros compostos

```
usuarios_joao = sessao.find(

    Pessoa, filters=[

        Like({"nome": "joao"})

        In({"id": [1, 2, 3]})

     ])
```

Fonte: O autor.

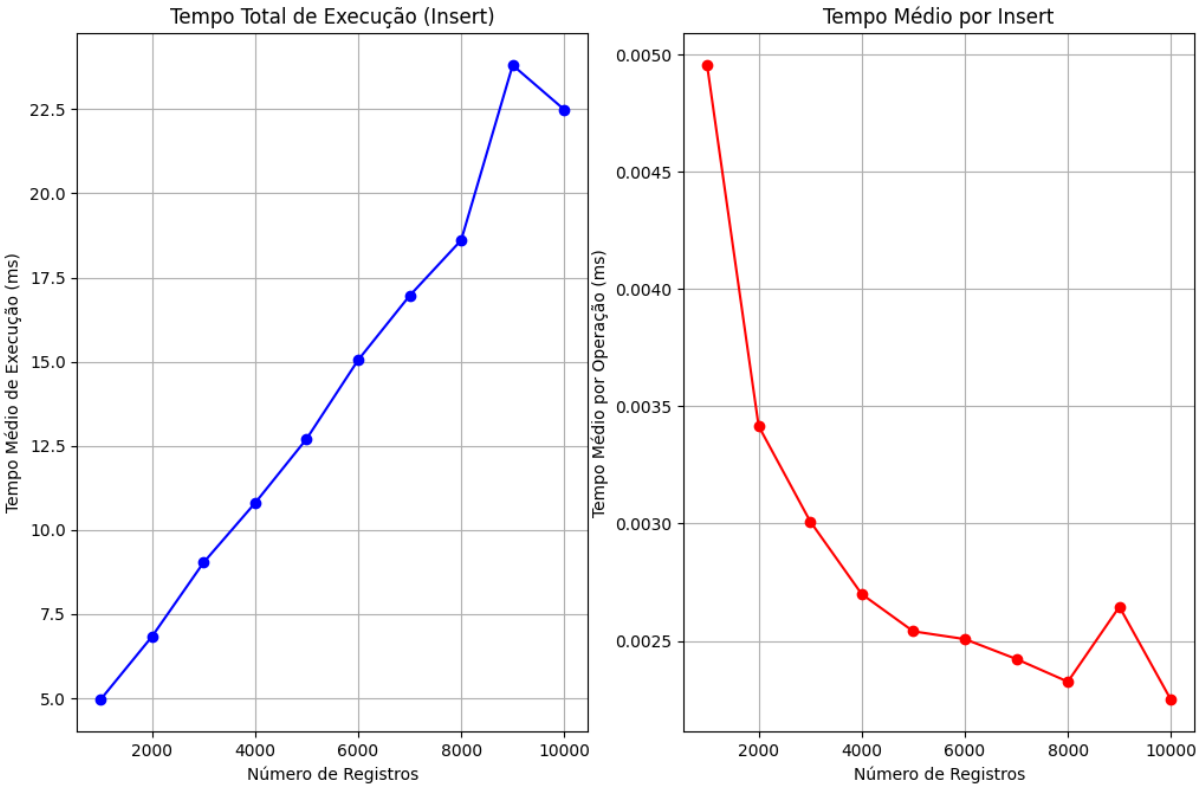
O sistema de filtros foi desenhado e nomeado da melhor maneira para representar e para permitir expressões SQL comuns sem que o desenvolvedor precise escrever SQL manualmente, abaixo segue os demais filtros presentes no ORM:

a) **Eq**: Filtro por igualdade;
b) **In**: Filtro para inclusão em uma lista de valores;
c) **Like**: Filtro para buscas parciais (similar ao operador *LIKE* do SQL);
d) **RangeFilter**: Filtro para selecionar registros cujo valor esteja dentro de um intervalo definido (por exemplo, entre duas datas ou números);
e) **NotEq**: Filtro para valores diferentes do especificado, ou seja, exclusão de registros com determinado valor;
f) **Composite**: Filtro que permite combinar múltiplas condições usando operadores lógicos como *AND* e *OR*, possibilitando consultas mais elaboradas e flexíveis;
g) **IsNull**: Filtro para verificar se um campo possui valor nulo (*NULL*), útil para buscas por ausência de dados.

Para avaliar o desempenho do *PylightORM*, foi conduzido um conjunto de experimentos seguindo a mesma metodologia previamente aplicada ao *SQLAlchemy*. Essa abordagem visa assegurar condições equivalentes de teste, permitindo uma análise comparativa fidedigna entre os dois ORMs. As operações analisadas — inserção, atualização, leitura e exclusão — foram executadas sobre uma tabela contendo 10.000 registros, com medições coletadas a cada 1.000 operações realizadas. A infraestrutura experimental utilizou containers PostgreSQL reinicializados a cada rodada, a fim de garantir um ambiente controlado, livre de

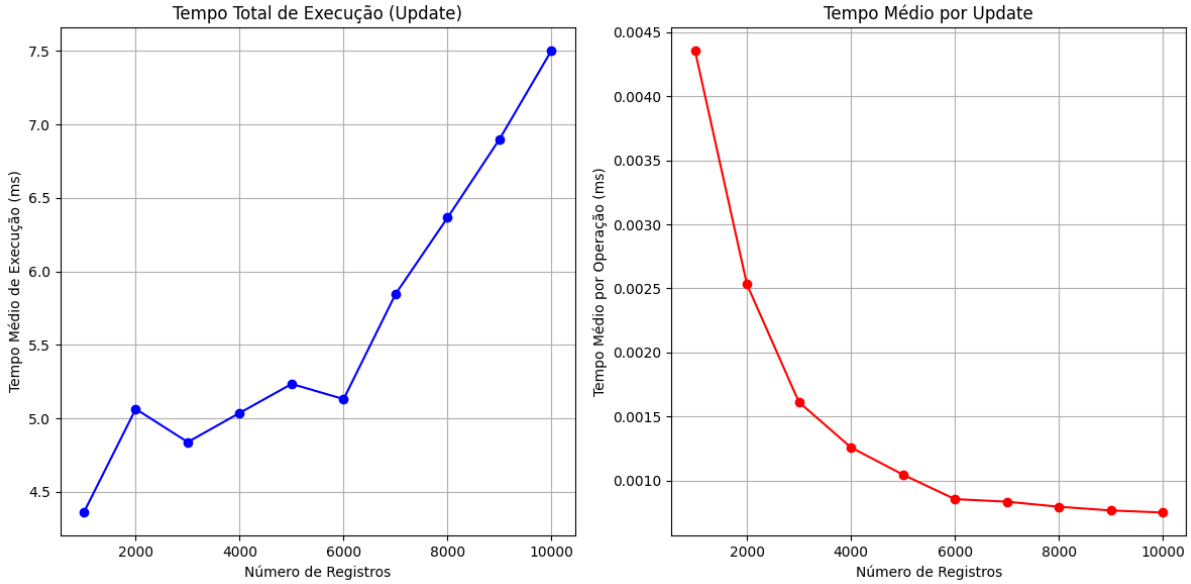
interferências externas como cache residual ou variações nos recursos computacionais. As Figuras 15 e 16 apresentam os resultados obtidos no PylightORM:

Figura 15 PylightORM Benchmark de Insert



Fonte: O autor, execução completa disponível no código de benchmarks do projeto.

Figura 16 PylightORM Benchmark de Delete



Fonte: O autor, execução completa disponível no código de benchmarks do projeto

Os testes indicaram que o *PylightORM* apresenta desempenho consistente e competitivo, mesmo com uma arquitetura mais simplificada. Por exemplo, para um volume de 8000 registros, a operação de inserção com *PylightORM* registrou um tempo médio por operação de aproximadamente 0.0025ms (resultando em um tempo total de cerca de 24ms). Comparativamente, o *SQLAlchemy* mostrou-se mais eficiente neste cenário de inserção, com um tempo médio por operação de cerca de 0.0023ms (tempo total aproximado de 19ms).

Ao analisar as operações de exclusão para os mesmos 8000 registros, utilizando os dados atualizados, o *PylightORM* apresentou um tempo médio por operação de aproximadamente 0.0008ms (tempo total de cerca de 6.4ms). Neste caso, o *SQLAlchemy* demonstrou um desempenho ligeiramente superior, com um tempo médio por exclusão de aproximadamente 0.00075ms (tempo total de cerca de **6.1ms**). Estes pontos específicos ilustram a vantagem do *SQLAlchemy* tanto em inserções quanto em exclusões.

Em suma, os benchmarks comparativos indicaram que, embora o SQLAlchemy tenha apresentado maior eficiência nas operações de inserção e uma leve vantagem nas exclusões nos testes finais, o PylightORM demonstrou um desempenho geral robusto e competitivo. As diferenças observadas são relativamente pequenas, especialmente quando se considera a longa trajetória e otimização do SQLAlchemy no ecossistema Python. Portanto, o PylightORM se posiciona como uma alternativa promissora e viável, oferecendo resultados sólidos mesmo diante de uma referência tão estabelecida no mercado.

5. CONSIDERAÇÕES FINAIS

Este artigo apresentou o desenvolvimento de um ORM minimalista e otimizado para Python, focado em simplicidade, eficiência e flexibilidade. O ORM oferece uma interface intuitiva, com bom desempenho e fácil integração com bancos de dados relacionais.

O artigo detalhou aspectos centrais do ORM, como sua abordagem intuitiva para a definição de modelos e a integração fluida com a camada de dados. Embora as operações fundamentais de persistência (incluindo criação, leitura, atualização e exclusão - CRUD) tenham sido implementadas e sirvam como base funcional, o ORM desenvolvido engloba outras funcionalidades relevantes que, por limitações de escopo e extensão deste artigo, não foram abordadas em detalhe. A utilização direta de classes Python para representar as tabelas do banco de dados, conforme explorado, simplifica notavelmente as operações de manipulação de dados, como inserção e consulta, abstraindo parte da complexidade inerente a ORMs mais robustos e tradicionais.

Os resultados obtidos reforçam que é possível construir um ORM enxuto, eficiente e funcional, capaz de atender às necessidades de projetos que demandam maior leveza e controle sem renunciar à produtividade. O projeto encontra-se preparado para evoluir, com perspectivas promissoras de expansão, incluindo suporte a migrações, execução de consultas mais complexas, e integração com soluções *Python* que utilizam cotidianamente SQL.

6. REFERÊNCIAS

COLLEY Derek. STANIER Clare, ASADUZZAMAN Md. **The Impact of Object-Relational Mapping Frameworks on Relational Query Performance**. 2017. Disponível em: https://www.derekcolley.co.uk/paper2.pdf Acesso em: 25 nov. 2024.

ALEXSOFT. **OBJECT-RELATIONAL MAPPING PROCESS**. 2023. Disponível em: https://www.altexsoft.com/blog/object-relational-mapping-tools . Acesso em: 26 nov. 2024.

BARNERS, Jeffrey. **Object-Relational Mapping as a Persistence Mechanism for Object-Oriented Applications**. 2007. Disponível em: https://digitalcommons.macalester.edu/mathcs_honors/6/ Acesso em: 26 nov. 2024.

PYTHON SOFTWARE FOUNDATION. **Python documentation**. Disponível em: https://docs.python.org/. Acesso em: 27 nov. 2024.

SQLALCHEMY. **SQLAlchemy documentation**. Disponível em: https://docs.sqlalchemy.org/ Acesso em: 27 nov. 2024.

FOWLER, M. **Patterns of Enterprise Application Architecture**. 2002. Disponível em:https://dl.ebooksworld.ir/motoman/Patterns%20of%20Enterprise%20Application%20Architecture.pdf. Acesso em 26/01/2025.

GEEKSFORGEEKS. ***Django ORM vs SQLAlchemy***. Disponível em: https://www.geeksforgeeks.org/django-orm-vs-sqlalchemy/. Acesso em: 30/01/2025.

PIMENTA, M. S. **Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application desig**. 2017. Disponível em: https://www.sciencedirect.com/science/article/abs/pii/S0950584916301859. Acesso em: 10/05/2025.

APPJOT. ***Scaling open source DBs for performance***. 2023. Disponível em: https://appjot.com/2023/11/12/scaling-open-source-dbs-for-performance. Acesso em 27 maio 2025.

GETACHEW, Samuel. **Differences Between Django ORM and SQLAlchemy: A Deep Dive. Medium**, 2023. Disponível em: https://medium.com/django-unleashed/differences-between-django-orm-and-sqlalchemy-a-deep-dive-abc123. Acesso em: 29 maio 2025.