UNIVERSIDADE FEDERAL DO MARANHÃO
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA - CCET
ENGENHARIA DA COMPUTAÇÃO

Felipe dos Santos Goiabeira

Mapeamento Declarativo para a Web Semântica: Desenvolvimento e Avaliação do Framework RDFMapper para Integração Objeto-RDF em Python

São Luís - MA
2025

Felipe dos Santos Goiabeira

Mapeamento Declarativo para a Web Semântica: Desenvolvimento e Avaliação do Framework RDFMapper para Integração Objeto-RDF em Python

Trabalho de Conclusão de Curso 2 apresentado ao Curso de Bacharelado em Engenharia da Computação da Universidade Federal do Maranhão como requisito parcial para a obtenção do grau de Bacharel em Engenharia da Computação.

Bacharelado em Engenharia da Computação

Universidade Federal do Maranhão

Orientador: Prof. Dr. Sérgio Souza Costa

São Luís - MA

2025

Ficha gerada por meio do SIGAA/Biblioteca com dados fornecidos pelo(a) autor(a).
Diretoria Integrada de Bibliotecas/UFMA

dos Santos Goiabeira, Felipe.
Mapeamento Declarativo para a Web Semântica: Desenvolvimento e Avaliação do Framework RDFMapper para Integração Objeto-RDF em Python / Felipe dos Santos Goiabeira. - 2025.
86 p.

Orientador(a): Sérgio Sousa Costa.
Monografia (Graduação) - Curso de Engenharia da Computação, Universidade Federal do Maranhão, Universidade Federal do Maranhão, 2025.

1. Web Semântica. 2. Mapeamento Objeto-rdf. 3. Python. 4. Dados Conectados. 5. Rdf. I. Sousa Costa, Sérgio. II. Título.

Figura 1 – Enter Caption

Felipe dos Santos Goiabeira

Mapeamento Declarativo para a Web Semântica: Desenvolvimento e Avaliação do Framework RDFMapper para Integração Objeto-RDF em Python

Trabalho de Conclusão de Curso 2 apresentado ao Curso de Bacharelado em Engenharia da Computação da Universidade Federal do Maranhão como requisito parcial para a obtenção do grau de Bacharel em Engenharia da Computação.

Trabalho de conclusão de curso 2. São Luís - MA, 28 de Julho de 2025:

Prof. Dr. Sérgio Souza Costa
Orientador
Universidade Federal do Maranhão

São Luís - MA
2025

"Estou em encantador estado de confusão."

(Ada Lovelace)

Resumo

A Web Semântica e os Dados Conectados oferecem um paradigma poderoso para a representação e integração de conhecimento, utilizando o Resource Description Framework (RDF) como modelo de dados fundamental. Contudo, existe uma lacuna significativa entre o modelo de grafos do RDF e o paradigma de orientação a objetos, predominante no desenvolvimento de software. Essa divergência impõe uma elevada carga cognitiva aos desenvolvedores, que são forçados a manipular triplas RDF manualmente, dificultando a produtividade e a adoção dessas tecnologias. Para endereçar essa lacuna, este trabalho apresenta o desenvolvimento e a avaliação do `RDFMapper`, uma biblioteca Python que implementa um Mapeamento Objeto-RDF. Inspirado em frameworks ORM, o `RDFMapper` utiliza técnicas de metaprogramação, como decoradores, para permitir o mapeamento declarativo de classes Python para conceitos RDF. A solução inclui uma API de consulta dinâmica (RDFRepository) que abstrai a complexidade do SPARQL e integra um mecanismo de validação de dados via SHACL. A eficácia do `RDFMapper` foi validada por meio de estudos de caso com dados abertos e por uma análise de desempenho comparativa com a biblioteca `RDFLib`. Os resultados demonstram que, embora a abstração introduza um custo de processamento, a biblioteca reduz drasticamente a complexidade do código, melhora a produtividade e apresenta maior eficiência no uso de memória durante a serialização em massa. A aplicação prática em dados reais evidenciou a capacidade da ferramenta em simplificar tanto a integração quanto a análise de dados semânticos. Conclui-se que o `RDFMapper` constitui uma contribuição relevante ao ecossistema Python, oferecendo uma ponte eficaz entre a orientação a objetos e a Web Semântica, e diminuindo a barreira para o desenvolvimento de aplicações baseadas em Dados Conectados.

Palavras-chave: Web Semântica, Mapeamento Objeto-RDF, Python, Dados Conectados, RDF, SPARQL, SHACL.

Lista de ilustrações

Lista de Códigos

Lista de abreviaturas e siglas

ANP	Agência Nacional do Petróleo, Gás Natural e Biocombustíveis
AI	Artificial Intelligence (Inteligência Artificial)
ANP	Agência Nacional do Petróleo, Gás Natural e Biocombustíveis
API	Application Programming Interface (Interface de Programação de Aplicações)
AWS	Amazon Web Services
BBC	British Broadcasting Corporation
CCSO	Computer Science Ontology (Ontologia da Ciência da Computação)
CRUD	Create, Read, Update, and Delete (Criar, Ler, Atualizar e Excluir)
CSV	Comma-Separated Values (Valores Separados por Vírgula)
DC	Dublin Core (Metadados Dublin Core)
DDL	Data Definition Language (Linguagem de Definição de Dados)
DDD	Domain-Driven Design (Design Orientado a Domínio)
DL	Description Logics (Lógicas Descritivas)
ETL	Extract, Transform, Load (Extração, Transformação e Carga)
FOAF	Friend of a Friend (Amigo de um Amigo)
GO	Gene Ontology
HTTP	Hypertext Transfer Protocol (Protocolo de Transferência de Hipertexto)
JPA	Java Persistence API (API de Persistência do Java)
JPQL	Java Persistence Query Language (Linguagem de Consulta de Persistência do Java)
JSON-LD	JavaScript Object Notation for Linked Data
LINQ	Language Integrated Query (Linguagem de Consulta Integrada)
LOD	Linked Open Data (Dados Abertos Conectados)

ML	Machine Learning (Aprendizado de Máquina)
ORM	Object-Relational Mapping (Mapeamento Objeto-Relacional)
OWL	Web Ontology Language (Linguagem de Ontologias para a Web)
RDF	Resource Description Framework
RDFS	RDF Schema (Esquema RDF)
SHACL	Shapes Constraint Language (Linguagem de Restrição de Formas)
SKOS	Simple Knowledge Organization System (Sistema de Organização de Conhecimento Simples)
SNOMED CT	Systematized Nomenclature of Medicine — Clinical Terms
SPARQL	SPARQL Protocol and RDF Query Language
SQL	Structured Query Language (Linguagem de Consulta Estrurada)
TCC	Trabalho de Conclusão de Curso
UFMA	Universidade Federal do Maranhão
URI	Uniform Resource Identifier (Identificador Uniforme de Recursos)
W3C	World Wide Web Consortium
WDQS	Wikidata Query Service (Serviço de Consulta do Wikidata)
XML	eXtensible Markup Language (Linguagem de Marcação Extensível)
XSD	XML Schema Definition (Definição de Esquema XML)

Sumário

1 Introdução

O crescimento exponencial da Web nas últimas décadas consolidou-a como o principal repositório de informações da sociedade contemporânea. Apesar disso, uma parcela significativa dos dados disponíveis encontra-se encapsulada em formatos não estruturados ou distribuída em silos de difícil integração, dificultando o aproveitamento pleno dessas informações por aplicações automatizadas. A proposta dos Dados Conectados (*Linked Data*), idealizada por Tim Berners-Lee, busca transformar esse cenário, promovendo práticas e padrões que viabilizam a publicação de dados de forma estruturada, interligada e acessível para máquinas (BERNERS-LEE, 2006).

Nesse contexto, a pilha tecnológica da Web Semântica, composta por padrões como RDF, SPARQL, RDFS e OWL, constitui um arcabouço robusto para a representação, integração e consulta de conhecimento em ambientes distribuídos. O modelo de grafos RDF, em especial, oferece flexibilidade e expressividade superiores para descrever entidades, relações e domínios complexos, tornando-se fundamental em iniciativas de dados abertos, interoperabilidade e inteligência artificial. Contudo, observa-se na prática um desafio recorrente: a maioria dos sistemas de software modernos é construída sobre o paradigma de orientação a objetos, enquanto as tecnologias da Web Semântica operam sobre grafos e triplas, originando um descompasso paradigmático que dificulta a adoção dessas ferramentas (HITZLER; KRÖTZSCH; RUDOLPH, 2021).

Desenvolvedores acostumados a frameworks e linguagens orientados a objetos, como Python, frequentemente enfrentam dificuldades ao manipular diretamente triplas RDF com bibliotecas como `RDFLib`. Tal abordagem exige um detalhamento excessivo, gerando código repetitivo, suscetível a erros e pouco alinhado à lógica de domínio da aplicação. Essa problemática guarda semelhança com a dificuldade histórica de integração entre objetos e bancos relacionais, solucionada pelo advento dos frameworks de Mapeamento Objeto-Relacional (ORM), como JPA, Hibernate e SQLAlchemy (FOWLER, 2002). No caso do universo semântico, ainda há carência de soluções maduras para esse mapeamento, o que motiva pesquisas e experimentações nesse campo.

Diante desse cenário, este trabalho propõe o desenvolvimento e a avaliação do **RDFMapper**, uma biblioteca Python concebida para atuar como uma camada de Mapeamento Objeto-RDF, capaz de aproximar o paradigma orientado a objetos das tecnologias da Web Semântica e, assim, facilitar o desenvolvimento de aplicações que consumam, manipulem e publiquem dados RDF de forma eficiente e declarativa. O objetivo geral deste estudo é viabilizar, por meio do RDFMapper, a modelagem e manipulação de dados semânticos utilizando classes Python, simplificando o acesso e a integração com

grafos RDF sem exigir do desenvolvedor conhecimentos avançados de SPARQL ou da estrutura interna das triplas.

Para cumprir tal objetivo, foram definidos objetivos específicos que norteiam a condução da pesquisa: projetar e implementar um sistema de mapeamento declarativo, baseado em decoradores, capaz de associar classes, atributos e propriedades RDF; incorporar mecanismos para o tratamento de relacionamentos complexos e prevenção de circularidade em grafos; criar uma API de consulta dinâmica, inspirada nos principais frameworks ORM, que abstraia a complexidade do SPARQL; validar a biblioteca por meio de estudos de caso utilizando dados reais e, por fim, realizar uma análise experimental comparativa para avaliar desempenho, produtividade e uso de recursos frente à abordagem tradicional baseada em `RDFLib` puro.

A relevância deste trabalho reside no potencial de promover maior aderência das práticas e padrões da Web Semântica no desenvolvimento de sistemas orientados a objetos, democratizando o acesso às tecnologias de dados conectados. Ao propor e avaliar o `RDFMapper`, espera-se contribuir para o avanço do estado da arte na integração entre objetos e grafos, oferecendo aos desenvolvedores uma solução robusta, expressiva e alinhada às tendências de software moderno.

Para tanto, este documento está estruturado da seguinte forma: a seção seguinte apresenta o referencial teórico sobre Web Semântica, dados conectados e mapeamento objeto-grafo; na sequência, detalha-se a metodologia de desenvolvimento do `RDFMapper`; posteriormente, são apresentados os resultados experimentais, análises e discussões; por fim, são apresentadas as conclusões e sugestões para trabalhos futuros.

2 Fundamentação Teórica

2.1 Dados Conectados

2.1.1 Definição

O termo *Dados Conectados* (do inglês, *Linked Data*) refere-se a um conjunto de boas práticas para a publicação e interligação de dados na Web, com o uso de tecnologias padronizadas para representação de dados estruturados. O objetivo principal é facilitar o consumo das informações por seres humanos e permitir o processamento automatizado por máquinas.

Apesar da facilidade atual em publicar dados na Web, eles geralmente são disponibilizados em formatos heterogêneos, como *PDF*, *JPEG* e *PNG*, que são compreensíveis para humanos, mas ineficientes para processamento automatizado. Em muitos casos, apenas especialistas conseguem interpretar o conteúdo desses dados.

Um mecanismo automatizado capaz de buscar, estruturar e interligar informações provenientes de diversas fontes, de forma legível por máquina e pronta para consumo imediato, permite que desenvolvedores acessem dados de múltiplas origens em tempo real, combinando-os de maneira automatizada e eficiente. Essa abordagem é viabilizada pelo uso dos Dados Conectados (BANDEIRA et al., 2015).

A iniciativa de Dados Conectados foi proposta por Tim Berners-Lee, em 2006, por meio da subseção “Linked Data” no documento “Design Issues”, como parte do movimento da Web Semântica. Desde então, diversas organizações adotaram o modelo: (i) o Google passou a utilizar JSON-LD no Gmail; (ii) a IBM anunciou o suporte a Dados Conectados em seu banco de dados DB2; (iii) o Facebook integrou a Graph API com RDF; (iv) a BBC gerou páginas com base em Dados Conectados; e (v) o governo do Reino Unido passou a disponibilizar dados em RDF.

A adoção de Dados Conectados permite que qualquer pessoa publique dados em um formato acessível tanto por humanos quanto por máquinas, promovendo maior interoperabilidade e fomentando aplicações inovadoras em áreas como mobilidade urbana, serviços públicos e comércio eletrônico. A varejista Best Buy, por exemplo, registrou um aumento de 15% a 30% nos cliques originados no Google após adotar RDF como formato de serialização (ISOTANI; BITTENCOURT, 2015).

2.1.2 Representação

A Web foi inicialmente concebida para o consumo humano, utilizando uma infraestrutura simples e padronizada. Contudo, aplicações enfrentam dificuldades no processamento das informações devido à ausência de mecanismos para agregação semântica.

Segundo (ISOTANI; BITTENCOURT, 2015), uma das soluções é o uso de metadados que descrevam os dados publicados na Web, permitindo localizar, estruturar e processar essas informações. Para garantir a interoperabilidade entre sistemas, esses metadados devem obedecer a padrões sintáticos e semânticos, além de utilizar vocabulários padronizados.

2.1.3 Princípios dos Dados Conectados

A construção da Web de Dados Conectados (*Linked Data*) fundamenta-se em um conjunto de boas práticas propostas por Tim Berners-Lee. Para compreender esses princípios, é necessário, primeiramente, esclarecer o conceito de **URI** (*Uniform Resource Identifier*).

Uma **URI** é um identificador globalmente único utilizado para nomear recursos de forma padronizada e interoperável. Pode ser entendida como um endereço universal que representa, de maneira inequívoca, qualquer entidade na Web — como uma pessoa, documento, conceito ou objeto físico (BERNERS-LEE; FIELDING; MASINTER, 2005). As URIs são empregadas tanto para identificação quanto para localização de recursos, principalmente quando acessíveis por meio do protocolo HTTP.

Com base nesse conceito, Berners-Lee definiu quatro princípios fundamentais para a publicação e interligação de dados na Web (BERNERS-LEE, 2006):

1. **Usar URIs como identificadores únicos para coisas**: cada entidade ou conceito deve ser representado por uma URI distinta, garantindo sua identificação inequívoca.
2. **Tornar essas URIs acessíveis por HTTP**: ao acessar uma URI, deve-se obter informações úteis sobre o recurso por meio de representações padronizadas.
3. **Fornecer representações úteis dessas URIs em RDF**: os dados retornados devem seguir o modelo RDF, favorecendo a semântica e a interoperabilidade entre sistemas.
4. **Incluir links para outras URIs, permitindo a descoberta de novos dados**: a interconexão entre recursos promove a navegabilidade e a expansão do conhecimento de forma distribuída.

Esses princípios têm como objetivo transformar a Web em uma grande rede de dados interligados, promovendo o compartilhamento, a reusabilidade e a integração de informações entre diferentes domínios.

2.1.4 Ontologias e Vocabulários

O conceito de ontologia tem origem na filosofia, com registros desde o século XIX por Rudolf Gockel. No contexto computacional, especialmente na Web Semântica, uma ontologia descreve formalmente os tipos de entidades em um domínio e como elas se relacionam (BREITMAN; CASANOVA; TRUSZKOWSKI, 2010).

De acordo com o W3C, uma ontologia deve conter:

- Classes (ou "coisas") nos domínios de interesse;
- Relacionamentos entre essas coisas;
- Propriedades (ou atributos) associados às entidades.

Além das ontologias, o uso de vocabulários padronizados é essencial para garantir a compreensão uniforme dos dados. Vocabulários definem termos comuns para descrever os dados, promovendo interoperabilidade. Exemplos incluem:

- **FOAF** (Friend of a Friend): descreve pessoas e suas relações;
- **DC** (Dublin Core): descreve recursos digitais;
- **SKOS** (Simple Knowledge Organization System): utilizado em sistemas de organização do conhecimento;
- **OWL** (Web Ontology Language): permite a criação de ontologias complexas.

Segundo (SEGUNDO, 2015), os vocabulários RDF, FOAF, RDFS, DC e OWL são os mais usados em datasets públicos. Em 2014, por exemplo, o RDF estava presente em 98,22% dos conjuntos de dados. O FOAF foi adotado por 69,1% dos datasets, enquanto em 2011 esse número era de apenas 27,46%. A mesma tendência foi observada para o Dublin Core (DC), cujo uso subiu de 31,19% em 2011 para 56,01% em 2014.

2.2 Padrões da W3C para Dados Conectados

A construção da Web Semântica e dos Dados Conectados se baseia em tecnologias padronizadas pelo W3C. Os principais padrões são:

- **RDF (Resource Description Framework)**: estrutura os dados como triplas no formato sujeito-predicado-objeto, permitindo a modelagem de relações entre recursos.
- **RDFS (RDF Schema)** e **OWL (Web Ontology Language)**: são utilizados para definir vocabulários, possibilitando inferência lógica e estruturação semântica dos dados.
- **SPARQL**: linguagem de consulta para grafos RDF, permite a extração e manipulação de dados estruturados.

Essas tecnologias formam a base técnica sobre a qual são construídos os datasets conectados, garantindo interoperabilidade entre diferentes fontes de dados na Web (W3C RDF Working Group, 2014).

2.2.1 O Ecossistema RDF: Modelo de Dados, Armazenamento e Consulta

O *Resource Description Framework* (RDF) é um modelo de dados padronizado pelo W3C, projetado para representar informações estruturadas e interconectadas na Web (W3C RDF Working Group, 2014). A estrutura fundamental do RDF é a **tripla**, uma asserção no formato `sujeito-predicado-objeto` que descreve uma propriedade de um recurso.

- O **sujeito** identifica o recurso que está sendo descrito.
- O **predicado** define a propriedade ou o tipo de relação.
- O **objeto** é o valor dessa propriedade, que pode ser um literal (texto, número, data) ou outro recurso.

Por exemplo, a tripla a seguir afirma que o recurso `ex:Joao` possui o nome "João da Silva":

Código 1 Exemplo de uma tripla RDF declarando um nome

```
1 ex:Joao foaf:name "João da Silva" .
```

Para garantir unicidade e interoperabilidade global, os sujeitos e predicados são geralmente identificados por URIs. Opcionalmente, podem ser usados **nós em branco** (*blank nodes*) para representar recursos que existem mas não necessitam de um identificador global.

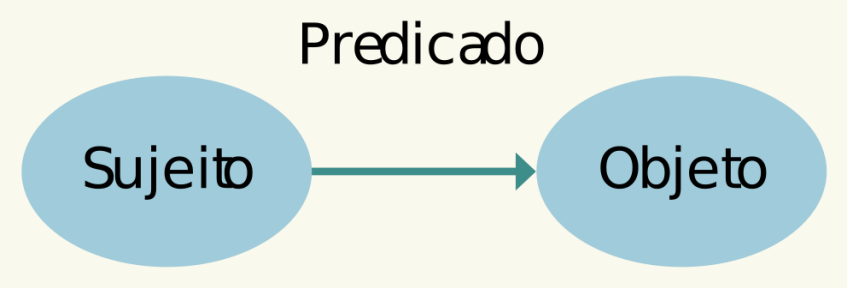


Figura 2 – Estrutura de uma tripla RDF. Fonte: (ISOTANI; BITTENCOURT, 2015)

Individualmente, cada tripla é uma pequena afirmação. No entanto, o verdadeiro poder do RDF emerge quando múltiplas triplas são interconectadas, formando um **grafo direcionado e rotulado**. Nesta estrutura, recursos (sujeitos e objetos) são os nós, e as propriedades (predicados) são as arestas que os conectam. Essa natureza de grafo permite a fácil integração de diferentes conjuntos de dados: para unir dois grafos RDF, basta unir seus conjuntos de triplas, formando um grafo de conhecimento maior e mais rico, que é a ideia central por trás dos Dados Conectados (*Linked Data*) (HITZLER; KRÖTZSCH; RUDOLPH, 2021).

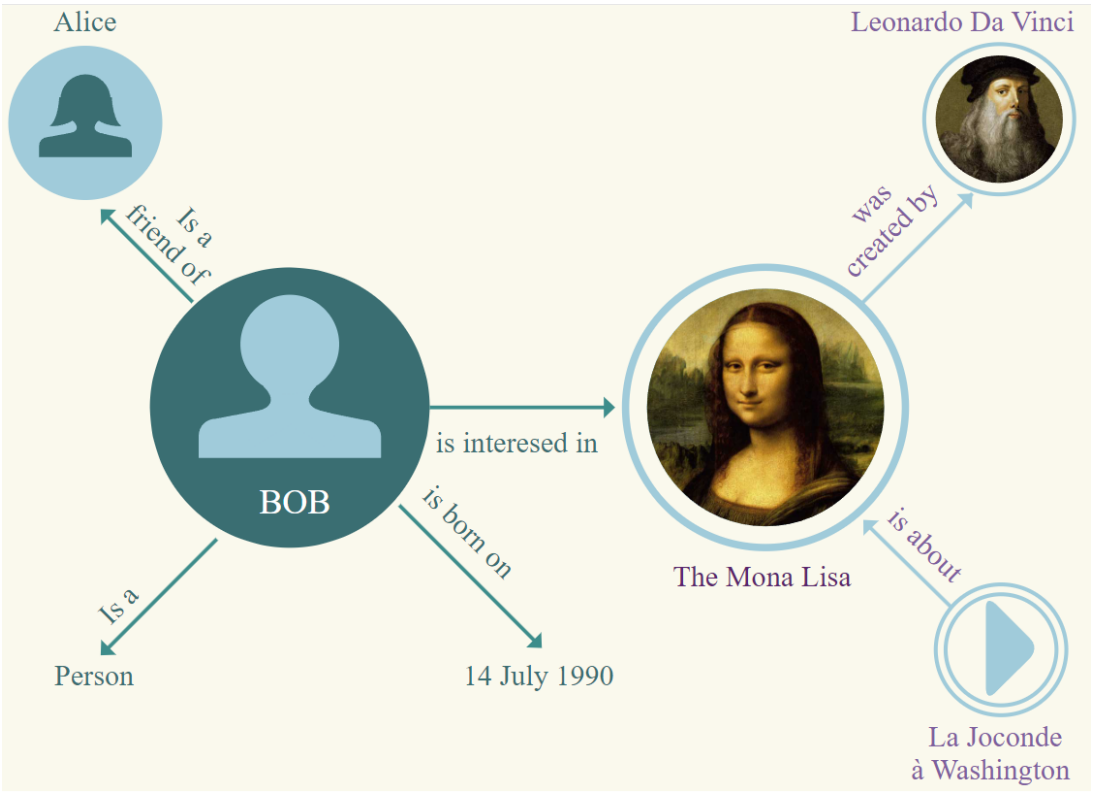


Figura 3 – Triplas interconectadas formando um grafo. Fonte: (ISOTANI; BITTENCOURT, 2015)

Diferentemente do modelo relacional, que organiza dados em tabelas com esquemas rígidos, o modelo de grafo do RDF é inerentemente flexível e ideal para dados semi-estruturados ou com alta conectividade. Essa flexibilidade é aprimorada por vocabulários e

ontologias, como RDFS (*RDF Schema*) e OWL (*Web Ontology Language*), que permitem definir esquemas (classes, hierarquias, restrições) e habilitam a **inferência semântica** — a capacidade de derivar novas triplas a partir das já existentes (ANTONIOU; HARMELEN, 2011).

Para persistir, gerenciar e consultar esses grafos de maneira eficiente, utilizam-se bancos de dados especializados chamados ***triplestores***. Trata-se de sistemas orientados a grafos, projetados para otimizar a indexação e a recuperação de triplas. Exemplos populares incluem **Virtuoso**, **Blazegraph**, **GraphDB** e **Fuseki** (WYLOT et al., 2018). A linguagem padrão para consultar dados em *triplestores* é a **SPARQL** (*SPARQL Protocol and RDF Query Language*), que permite a extração de informações por meio da correspondência de padrões de triplas no grafo.

2.2.2 RDFS – RDF Schema

O *RDF Schema* (RDFS) é uma extensão do RDF que permite descrever vocabulários por meio da definição de classes (`rdfs:Class`) e propriedades (`rdf:Property`), assim como suas relações hierárquicas por meio dos termos `rdfs:subClassOf`, `rdfs:domain` e `rdfs:range`. Isso permite estruturar semanticamente os dados RDF e possibilita inferências lógicas simples (BRICKLEY; GUHA, 2014).

Por exemplo, é possível declarar que a propriedade `foaf:mbox` tem como domínio a classe `foaf:Person`, o que implica que qualquer recurso que possua essa propriedade pode ser considerado uma instância da referida classe.

2.2.3 OWL – Web Ontology Language

Se o RDFS pode ser visto como um dicionário, responsável por definições e relações simples entre termos, a **OWL** funciona como um código de leis estruturado, capaz de expressar restrições, regras e propriedades complexas para modelagem de conhecimento. Ela vai além das definições, estabelecendo regras complexas e restrições lógicas sobre como os conceitos podem se relacionar. Por exemplo, com OWL é possível afirmar que "uma pessoa deve ter exatamente dois pais biológicos" ou que "um país não pode ser parte de si mesmo". É esse nível de expressividade formal que permite a um computador raciocinar sobre os dados, verificar a consistência e inferir novos conhecimentos de forma autônoma, algo que o RDFS não consegue fazer sozinho.

Formalmente, a linguagem OWL é uma evolução do RDFS, desenvolvida pelo W3C para representar ontologias ricas por meio de uma base lógica rigorosa, conhecida como **Lógicas Descritivas (Description Logics – DL)**. Enquanto o RDFS permite uma estruturação básica, o OWL possibilita uma modelagem muito mais precisa, adequada para domínios onde a clareza semântica e a verificação de consistência são cruciais, como

em aplicações biomédicas, de engenharia ou jurídicas (GROUP, 2012).

Com OWL, é possível definir uma vasta gama de axiomas (regras e fatos) sobre classes e propriedades. Alguns exemplos das capacidades expressivas do OWL incluem:

- **Restrição de cardinalidade**: Permite especificar exatamente quantas vezes uma propriedade pode ser usada (ex: "uma pessoa tem exatamente 2 braços"). Utiliza termos como `owl:cardinality`, `owl:minCardinality` e `owl:maxCardinality`.
- **Classes equivalentes e disjuntas**: É possível declarar que duas classes diferentes têm exatamente os mesmos membros (`owl:equivalentClass`) ou que não podem ter nenhum membro em comum (`owl:disjointWith`, ex: "um ser humano não pode ser simultaneamente um "bebê" e um "adulto"").
- **Definição de propriedades complexas**: Pode-se definir o comportamento lógico das relações, como propriedades inversas (`owl:inverseOf`, ex: "se A é "filho de" B, então B é "pai de" A"), transitivas, simétricas e funcionais.
- **Composição de classes**: Permite criar novas classes a partir da combinação de outras, usando operações lógicas como interseção (`owl:intersectionOf`, ex: "definir a classe "Mãe" como a interseção de "Mulher" e "Pessoa que tem filho""), união e complemento.

Esses recursos fazem do OWL uma linguagem fundamental para domínios com regras de negócio rígidas e estrutura semântica bem definida.

Dialetos do OWL:

OWL é dividido em três sublinguagens (ou perfis) com diferentes níveis de expressividade e complexidade computacional:

- **OWL Lite**: Versão mais simples, para casos de uso com hierarquias básicas e restrições simples.
- **OWL DL (Description Logic)**: Oferece máxima expressividade enquanto mantém a garantia de que as consultas e inferências terminarão em um tempo finito (decidibilidade). É o perfil mais utilizado na prática.
- **OWL Full**: Combina totalmente a sintaxe do OWL com a do RDF, permitindo máxima flexibilidade, mas sem as garantias computacionais do OWL DL.

Inferência e Raciocínio:

O principal benefício da modelagem com OWL é a capacidade de realizar inferência automática por meio de softwares especializados chamados ***reasoners*** (raciocinadores),

como *HermiT*, *Pellet* e *Fact++*. Eles conseguem: (i) verificar se uma ontologia é logicamente consistente (não contém contradições); (ii) classificar hierarquias automaticamente (inferindo que 'Poodle' é uma subclasse de 'Mamífero', mesmo que não tenha sido dito explicitamente); e (iii) derivar novos fatos a partir dos axiomas definidos. Por exemplo, a partir das seguintes declarações:

Código 2 Definição de classe equivalente em OWL para inferência

```
ex:Aluno owl:equivalentClass [
  owl:intersectionOf ( ex:Pessoa [
    owl:hasValue ex:statusMatriculado ;
    owl:onProperty ex:situacao ] )
] .
```

Um *reasoner* pode inferir que qualquer indivíduo do tipo `ex:Pessoa` com a propriedade `ex:situacao` igual a `ex:statusMatriculado` é, por definição, um `ex:Aluno`, mesmo que isso não esteja explícito no dado original.

Aplicações Práticas:

OWL é amplamente utilizado na construção de ontologias em diversas áreas, como:

- **Ciências da Vida e Saúde**: Como na ontologia *Gene Ontology (GO)* ou *SNOMED CT*, que estruturam o vasto conhecimento biomédico.
- **Web Semântica e *Linked Data***: Para garantir a interoperabilidade e o entendimento mútuo entre vocabulários distintos.
- **Sistemas Inteligentes**: Permitindo que agentes de software compreendam e raciocinem sobre dados estruturados de forma autônoma.

A combinação de OWL com RDF, RDFS e SPARQL forma a base técnica da Web Semântica, conforme proposta por Tim Berners-Lee.

2.2.4 SPARQL – SPARQL Protocol and RDF Query Language

O *SPARQL Protocol and RDF Query Language* (SPARQL) é a linguagem oficial recomendada pelo W3C para consulta e manipulação de dados RDF. Ela permite a formulação de consultas declarativas sobre grafos RDF, extraindo informações com base em padrões de tripla, e inclui recursos comparáveis às cláusulas de linguagens relacionais, como `SELECT`, `WHERE`, `ORDER BY`, `GROUP BY` e `FILTER` (W3C SPARQL Working Group, 2013).

Um exemplo básico de consulta SPARQL que retorna os nomes de pessoas em um grafo RDF:

Código 3 Consulta SPARQL para selecionar nomes de pessoas

```
SELECT ?nome WHERE {
  ?pessoa a foaf:Person ;
          foaf:name ?nome .
}
```

SPARQL também permite:

- **Consultas federadas**: acessam múltiplos endpoints SPARQL simultaneamente com `SERVICE`.
- **Inferência**: quando combinada com RDFS ou OWL e um reasoner, SPARQL pode acessar fatos derivados logicamente.
- **SPARQL Update**: extensão que permite inserção (`INSERT`), exclusão (`DELETE`) e modificação de triplas RDF.
- **Expressões agregadas**: como `COUNT`, `AVG`, `SUM`, fundamentais para análise de dados.

SPARQL é essencial para sistemas baseados em Linked Data, permitindo consultas sobre bases como DBpedia, Wikidata e BioPortal, bem como em repositórios locais RDF.

2.2.5 SHACL – Shapes Constraint Language

O *Shapes Constraint Language* (SHACL) é uma linguagem padronizada pelo W3C utilizada para validar grafos RDF com base em restrições semânticas previamente definidas (W3C RDF Data Shapes Working Group, 2017). Diferentemente de linguagens orientadas à consulta, como SPARQL, o SHACL é voltado para verificação de integridade de dados. Por meio dele, é possível definir *shapes* (formas) que descrevem como instâncias de dados RDF devem se comportar, estabelecendo restrições como tipo de dado, cardinalidade, valores permitidos, propriedades obrigatórias, padrões regex, entre outras.

Por exemplo, uma *shape* pode declarar que uma instância da classe `foaf:Person` deve possuir obrigatoriamente a propriedade `foaf:name`, que deve ser um literal do tipo `xsd:string`. Caso algum recurso do grafo RDF não atenda a essas restrições, ele será considerado inválido.

O SHACL também permite lógica condicional, herança entre shapes, reutilização modular e a execução de regras personalizadas escritas em SPARQL (*SHACL-SPARQL*), ampliando significativamente seu poder expressivo. Essa linguagem é amplamente utilizada em contextos onde a qualidade, consistência e conformidade dos dados RDF são essenciais, como em aplicações governamentais, bancos de dados científicos e integração de dados corporativos.

Ferramentas como o `pySHACL` possibilitam a aplicação dessas validações diretamente em ambientes Python, promovendo sua integração com pipelines de ingestão e transformação de dados semânticos (SOMMER, 2024).

2.2.6 Principais Bancos de Dados para Dados Conectados e RDF

A persistência de dados no modelo RDF é realizada por sistemas especializados denominados triplestores ou bancos de dados orientados a grafos RDF. Esses sistemas são otimizados para armazenar e consultar grandes volumes de triplas RDF, suportar inferência semântica com base em RDFS/OWL e oferecer compatibilidade com a linguagem SPARQL. Abaixo, são apresentados os principais bancos de dados amplamente adotados:

- **Virtuoso** Um dos triplestores mais populares, desenvolvido pela OpenLink Software. Suporta RDF, SPARQL 1.1, RDFS e OWL, e oferece recursos como versionamento, segurança granular e integração com dados relacionais. É amplamente utilizado em aplicações de Linked Data e pelo projeto DBpedia, é um dos projetos mais importantes do ecossistema de dados conectados, seu objetivo é extrair informações estruturadas a partir da Wikipedia e disponibilizá-las na forma de dados RDF, interligando-as a outros conjuntos de dados na Web Semântica.(ERLING; MIKHAILOV, 2012).

- **Blazegraph** Banco de dados RDF open-source com suporte a SPARQL 1.1, inferência limitada e alta performance. Era utilizado pela Wikimedia Foundation para alimentar o Wikidata Query Service, embora hoje esteja descontinuado (SYSTAP, 2024).

- **GraphDB** Desenvolvido pela Ontotext, oferece suporte completo a RDF, SPARQL e inferência ontológica com OWL 2 RL. É bastante usado em ambientes corporativos e acadêmicos que demandam alta escalabilidade e controle de consistência semântica (Ontotext, 2023).

- **Apache Jena Fuseki** Triplestore desenvolvido pela Apache Foundation como parte do projeto Jena. Fornece um servidor SPARQL completo, com suporte à leitura/escrita de grafos, ontologias e segurança básica. É frequentemente utilizado em ambientes de pesquisa e prototipação (FOUNDATION, 2024a).

- **Stardog** Banco RDF comercial com funcionalidades avançadas como controle de versão de grafos, replicação, integração com ML e consultas híbridas SPARQL/SQL. Suporta raciocínio com OWL 2 DL e SHACL para validação (UNION, 2024).

- **AllegroGraph** Banco RDF de alto desempenho desenvolvido pela Franz Inc., voltado para aplicações corporativas em grafos semânticos. Suporta SPARQL, Prolog, visualização, consulta geoespacial e integração com IA (INC., 2024).

- **Amazon Neptune** Serviço gerenciado de banco de grafos da AWS que suporta RDF com SPARQL e Property Graph com Gremlin. É voltado para aplicações escaláveis na nuvem com requisitos de alta disponibilidade e segurança (SERVICES, 2024).

2.3 Grandes Projetos de Dados Conectados

O avanço da Web Semântica possibilitou o surgimento de diversas iniciativas voltadas à publicação de dados interligados (*Linked Data*) em larga escala. Entre os projetos mais significativos encontram-se o **Wikidata**, mantido pela *Wikimedia Foundation*, e o **DBpedia**, ambos pilares fundamentais na estruturação e disseminação de dados conectados abertos na Web.

2.3.1 Wikidata e o Wikidata Query Service

Wikidata é uma base de conhecimento colaborativa, multilíngue e estruturada que armazena dados utilizados por diversos projetos da Wikimedia, como a Wikipédia, além de servir como um repositório central de dados conectados reutilizáveis na Web (VRANDEčIć; KRöTZSCH, 2014).

Lançado em 2012, o Wikidata foi desenvolvido com o objetivo de fornecer uma fonte comum de dados para projetos Wikimedia, reduzindo a redundância entre edições de diferentes idiomas e promovendo maior consistência e atualidade das informações. Seu modelo de dados é baseado em triplas RDF, possibilitando fácil integração com outras bases e tecnologias semânticas.

O ***Wikidata Query Service*** (WDQS) é uma interface de consulta SPARQL pública que permite explorar e recuperar informações complexas diretamente do grafo RDF do Wikidata. Este serviço é alimentado continuamente por atualizações do próprio Wikidata e utiliza o triplestore **Blazegraph** como backend, garantindo escalabilidade e performance (FOUNDATION, 2024b).

2.3.2 DBpedia

O **DBpedia** é um projeto de extração de conhecimento estruturado a partir da Wikipédia. Por meio de técnicas de processamento de linguagem natural, o DBpedia converte infocaixas e outros elementos da enciclopédia em triplas RDF, publicando-as como Linked Data (AUER et al., 2007).

Lançado em 2007, o DBpedia é amplamente utilizado como fonte de referência na Web Semântica e atua como um hub central na LOD (*Linked Open Data Cloud*). Ele fornece identificadores URI para milhões de entidades (como pessoas, locais, organizações,

obras etc.) e estabelece vínculos com diversas outras fontes, como GeoNames, MusicBrainz, Freebase, entre outras.

Além disso, o DBpedia oferece uma interface SPARQL pública e dados disponíveis para download em diversos formatos RDF, promovendo a reutilização e integração dos dados em aplicações acadêmicas, comerciais e governamentais.

2.3.3 GeoNames

GeoNames é um banco de dados geográfico abrangente que fornece nomes, coordenadas, elevação e outros atributos para lugares em todo o mundo. Lançado em 2005, o GeoNames integra dados de várias fontes e os disponibiliza como Linked Data, utilizando URIs persistentes para cada local. Ele atua como um ***hub* central para dados geográficos** na Linked Open Data Cloud (LOD Cloud), sendo fundamental para vincular informações geográficas em diversos outros conjuntos de dados e aplicações (GEONAMES, 2025).

2.3.4 MusicBrainz

MusicBrainz é uma enciclopédia de música de código aberto que coleta metadados musicais detalhados sobre artistas, lançamentos, gravações e obras. Fundado em 2000, seu objetivo é criar um **identificador universal para a música**. Os dados do MusicBrainz são publicados como **Linked Data** com URIs para entidades musicais, permitindo a interoperabilidade e o enriquecimento de aplicações musicais. É uma fonte primária para o gerenciamento de bibliotecas musicais, serviços de streaming e análise de tendências (MUSICBRAINZ, 2025).

2.3.5 Linked Open Data Cloud (LOD Cloud)

A **Linked Open Data Cloud (LOD Cloud)** não é um projeto singular, mas uma **visualização do vasto ecossistema de conjuntos de dados abertos publicados como Linked Data** e interconectados na Web Semântica. Ela ilustra a densidade de links entre diferentes conjuntos de dados, com projetos como DBpedia, Wikidata, GeoNames e MusicBrainz atuando como nós centrais. A LOD Cloud demonstra a promessa da Web de Dados, onde informações de domínios variados podem ser descobertas e integradas semanticamente, evidenciando a importância da interoperabilidade e padronização na publicação de dados (LOD. . . , 2025).

2.3.6 BBC Linked Data

A **BBC (British Broadcasting Corporation)** foi uma das pioneiras na aplicação de Dados Conectados em larga escala no contexto de mídia. A iniciativa visava organizar

e publicar metadados sobre seus programas, notícias, artistas e locais, utilizando padrões da Web Semântica. Ao disponibilizar seus dados como Linked Data, a BBC **melhorou a descoberta de conteúdo** e a navegação entre diferentes mídias, além de otimizar a reutilização interna de informações. Este projeto serve como um modelo para grandes organizações de mídia que buscam gerenciar e distribuir vastos acervos de conteúdo de forma eficiente e interconectada (DAVIES; BIRON, 2012).

2.3.7 Importância desses projetos

Tanto o Wikidata quanto o DBpedia, juntamente com GeoNames, MusicBrainz, a LOD Cloud e a iniciativa da BBC, representam exemplos concretos do potencial da Web Semântica para organização e disseminação de conhecimento em escala global. Eles demonstram como a aplicação de padrões como RDF, SPARQL e OWL, aliados a comunidades ativas, podem tornar os dados mais acessíveis, interligados e úteis para humanos e máquinas.

Estes projetos também impulsionam pesquisas em áreas como Inteligência Artificial, Web Semântica, linguística computacional, recuperação de informação e aprendizado de máquina, servindo como fontes ricas de dados abertos e estruturados.

2.4 Motivação: A Lacuna entre o Modelo de Grafos e a Orientação a Objetos

A pilha de tecnologias da Web Semântica, com o RDF como modelo de dados fundamental, oferece uma capacidade sem precedentes para a representação de conhecimento e integração de dados heterogêneos em escala Web (HITZLER; KRÖTZSCH; RUDOLPH, 2021). A flexibilidade do modelo de grafos e a expressividade de linguagens de consulta como SPARQL permitem a criação de bases de conhecimento ricas e interconectadas, superando muitas das limitações dos silos de dados tradicionais.

Contudo, enquanto o modelo de grafos do RDF é otimizado para a máquina e para a representação de dados, ele apresenta um desalinhamento paradigmático com a maioria das linguagens de programação modernas, que são predominantemente orientadas a objetos (um paradigma de programação que organiza o software em "objetos" que representam entidades do mundo real, como uma "Pessoa" ou um "Produto", cada um com suas próprias características e comportamentos). Desenvolvedores de software estão habituados a modelar o domínio de suas aplicações em termos de classes, objetos, atributos e relacionamentos, uma abstração que se provou eficaz para gerenciar a complexidade de sistemas de software (FOWLER, 2002).

Na prática, o desenvolvimento de aplicações sobre grafos RDF utilizando bibliotecas

de baixo nível, como a "RDFLib" em Python ou a "Apache Jena" em Java, exige que o programador opere diretamente no nível dos dados. Isso envolve a construção explícita de triplas (sujeito-predicado-objeto), o gerenciamento manual de URIs e *namespaces*, e a escrita de código repetitivo (*boilerplate*) para realizar operações básicas de criação, leitura, atualização e exclusão (CRUD). Esse processo, além de trabalhoso, é suscetível a erros e afasta o desenvolvedor da lógica de negócio da aplicação (HEATH; BIZER, 2011).

Essa lacuna entre o modelo de dados e o modelo de domínio da aplicação não é um problema novo. No mundo dos bancos de dados relacionais, ele foi solucionado com sucesso pela introdução de frameworks de **Mapeamento Objeto-Relacional (ORM)**, como SQLAlchemy e JPA/Hibernate. Essas ferramentas automatizam a tradução entre o modelo de objetos da aplicação e o modelo relacional do banco de dados, permitindo que os desenvolvedores manipulem objetos de forma intuitiva, enquanto o ORM gerencia a complexidade das consultas SQL e da persistência de dados em segundo plano (BAUER; KING, 2015).

De forma análoga, surge a necessidade de uma camada de abstração para o mundo RDF — um **Mapeamento Objeto-RDF** — que ofereça os mesmos benefícios de produtividade, legibilidade e manutenibilidade. A adoção de tal ferramenta permitiria que os desenvolvedores definissem seus modelos de domínio como classes Python (ou de outra linguagem) e manipulassem instâncias dessas classes, deixando a cargo do mapeador a tarefa de traduzir essas operações para a manipulação de triplas em um grafo RDF e a execução de consultas SPARQL. Isso não apenas acelera o desenvolvimento, mas também torna as tecnologias da Web Semântica mais acessíveis a uma comunidade mais ampla de programadores que não são especialistas em RDF (BISCHOF et al., 2011).

2.5 Mapeamento de Dados e Persistência

2.5.1 O Papel dos ORMs

O *Object-Relational Mapping* (ORM) resolve a lacuna entre o modelo OO e o modelo relacional, traduzindo objetos em linhas de tabelas e vice-versa (FOWLER, 2002; AMBLER, 2003). Entre os principais benefícios da utilização de ORM estão o aumento da produtividade dos desenvolvedores, a redução da repetição de código para acesso a dados, a diminuição do acoplamento entre a lógica de negócio e a persistência, bem como o suporte a técnicas de programação orientada a objetos como herança, polimorfismo e encapsulamento (BERNSTEIN; NEWCOMER, 2008).

Além disso, frameworks ORM geralmente oferecem funcionalidades adicionais, como:

- Geração automática do esquema de banco de dados a partir das classes do sistema;

- Validação de dados com base nas restrições definidas no modelo de objetos;
- Controle de transações e gerenciamento de conexões com o banco de dados;
- Cache de dados para melhorar a performance de acesso;
- Suporte a consultas baseadas em objetos, utilizando linguagens específicas como HQL (Hibernate Query Language) ou LINQ (Language Integrated Query).

A especificação JPA, por exemplo, estabelece convenções e anotações que tornam a persistência de objetos Java transparente, facilitando a definição de entidades, relacionamentos e restrições (Eclipse Foundation, 2020; KEITH; SCHINCARIOL, 2009). Sua abordagem declarativa, via anotações, inspirou diretamente o design da API do RDFMapper, que adota decoradores Python para cumprir objetivo análogo.

2.5.2 O Desafio do Mapeamento Objeto-Grafo

A transposição do paradigma ORM para o universo RDF é complexa devido a diferenças estruturais:

- **Estrutura**: Tabelas relacionais são rígidas (*schema-on-write*), enquanto grafos RDF são flexíveis (*schema-on-read*).
- **Identidade**: No relacional, chaves primárias locais; no RDF, URIs globais.
- **Esquema**: DDL relacional versus ontologias RDFS/OWL.
- **Relacionamentos**: SQL utiliza JOINs; RDF navega via padrões no grafo.

Para ilustrar, considere o seguinte "mini-schema":

Modelo Relacional

```
Tabela Pessoa
-------------
id | nome
1  | João

Tabela Email
------------
id | pessoa_id | email
1  |     1     | joao@email.com
```

Modelo RDF

```
ex:Joao a foaf:Person ;
  foaf:name "João" ;
  foaf:mbox "joao@email.com" .
```

No relacional, relacionamentos são explícitos via chaves estrangeiras e JOINs. No RDF, a relação está diretamente no grafo, e sua consulta/navegação é feita via SPARQL. O

esquema relacional precisa ser alterado para adicionar atributos; no RDF, basta adicionar novas propriedades a qualquer entidade. Essas distinções justificam abordagens próprias para mapeamento objeto-grafo.

SQLAlchemy (Python) e JPA (Java) exemplificam maturidade ORM no mundo relacional. Ambos usam metaprogramação — decoradores, metaclasses — para mapear objetos de forma declarativa e flexível, tornando a API intuitiva para desenvolvedores.

A abordagem declarativa, consolidada pelo JPA e por frameworks como SQLAlchemy, serviu de inspiração para o RDFMapper, que utiliza decoradores Python para mapear classes, propriedades e relacionamentos RDF de forma transparente e extensível.

2.5.3 JPA – Java Persistence API como Exemplo Canônico

Para ilustrar o conceito de uma especificação ORM madura e padronizada, a **Java Persistence API (JPA)** serve como um exemplo canônico do ecossistema Java. Embora este trabalho seja focado em Python, a análise da arquitetura da JPA é fundamental para compreender os princípios de design — como o uso de anotações para mapeamento declarativo e a gestão do ciclo de vida de entidades — que influenciaram o desenvolvimento de ferramentas de persistência em diversas linguagens.

A JPA é uma especificação oficial da plataforma Jakarta EE que define um conjunto de padrões para o mapeamento, persistência e recuperação de dados entre objetos Java e bancos de dados relacionais (Eclipse Foundation, 2020). Seu objetivo principal é abstrair o acesso aos dados por meio de uma camada de persistência orientada a objetos, reduzindo a complexidade do código boilerplate e promovendo uma abordagem mais limpa e manutenível para aplicações empresariais.

O modelo de persistência do JPA permite que classes Java sejam mapeadas para tabelas relacionais por meio de anotações ou arquivos XML. Por exemplo, a anotação `@Entity` indica que uma classe Java é uma entidade persistente, enquanto `@Id` define a chave primária. As relações entre objetos também são representadas com anotações como `@OneToMany`, `@ManyToOne`, `@OneToOne`, entre outras, possibilitando o mapeamento de associações complexas, herança e composição (KEITH; SCHINCARIOL, 2009).

O JPA é uma especificação, e sua implementação depende de provedores como *Hibernate*, *EclipseLink*, *OpenJPA* e *DataNucleus*, que fornecem a lógica de persistência real por trás da interface padronizada. Um dos principais componentes da API é a interface `EntityManager`, responsável por operações como inserção, atualização, remoção e consulta de entidades no banco de dados. O ciclo de vida das entidades é gerenciado pelo contexto de persistência, que controla o estado das instâncias em memória (transientes, gerenciadas, destacadas ou removidas) (BERNSTEIN; NEWCOMER, 2008).

Além disso, o JPA fornece uma linguagem de consulta orientada a objetos chamada

JPQL (*Java Persistence Query Language*), que é semelhante à SQL, porém baseada no modelo de entidades, em vez do modelo relacional. JPQL permite realizar consultas complexas sem expor a lógica da estrutura do banco de dados, favorecendo a portabilidade entre diferentes SGBDs (Sistemas Gerenciadores de Bancos de Dados) (RUBINGER; BURKE, 2010).

2.5.4 O Desafio do Mapeamento Objeto-Grafo: Do Relacional ao Semântico

Embora o sucesso dos ORMs no mundo relacional sirva como inspiração, a transposição direta desse paradigma para o ecossistema da Web Semântica é impraticável devido a diferenças conceituais e estruturais profundas entre os dois modelos de dados (ANGLES; GUTIERREZ, 2008). A tentativa de mapear objetos para grafos RDF revela um conjunto de desafios que exigem uma abordagem especializada, distinta daquela adotada por frameworks como JPA ou SQLAlchemy. As principais divergências são:

- **Estrutura: Tabelas Rígidas vs. Grafos Flexíveis.** O modelo relacional é fundamentado em esquemas rígidos, onde os dados devem se conformar a uma estrutura de tabelas e colunas predefinida (*schema-on-write*). Qualquer alteração na estrutura exige uma migração formal do esquema (usando comandos "ALTER TABLE"). Em contraste, o RDF opera sob um modelo de grafo flexível e semiestruturado (*schema-on-read*), onde novas propriedades (predicados) e relações podem ser adicionadas a um recurso a qualquer momento sem a necessidade de alterar um esquema centralizado. Essa natureza dinâmica, embora poderosa, torna o mapeamento para classes com atributos fixos um desafio significativo (WOOD et al., 2014).

- **Identificadores: Chaves Locais vs. URIs Globais.** Em bancos de dados relacionais, a identidade de uma entidade é garantida por chaves primárias (geralmente inteiros ou UUIDs), cujo escopo é local à tabela ou ao banco de dados. As relações são estabelecidas por chaves estrangeiras que referenciam essas chaves primárias. No RDF, a identidade é conferida por **URIs** (*Uniform Resource Identifiers*), que são identificadores globalmente únicos. Um mapeador objeto-grafo deve, portanto, gerenciar a resolução e a criação de URIs, um conceito fundamentalmente diferente da autoincrementação de chaves em bancos de dados relacionais (HEATH; BIZER, 2011).

- **Esquema: DDL vs. Ontologias.** A semântica e as restrições do modelo relacional são definidas por meio de uma Linguagem de Definição de Dados (DDL), que especifica tipos de dados, restrições de nulidade e integridade referencial. No mundo semântico, o "esquema" é definido por vocabulários e ontologias (usando RDFS e OWL), que descrevem classes, hierarquias, domínios, intervalos e relações lógicas complexas. Uma ferramenta de mapeamento para RDF deve ser capaz não apenas

de mapear atributos, mas também de interpretar e, idealmente, alavancar a riqueza semântica expressa por essas ontologias (HITZLER; KRÖTZSCH; RUDOLPH, 2021).

- **Relações: JOINs vs. Navegação no Grafo.** A recuperação de dados relacionados em SQL é realizada por meio da operação "JOIN", que combina linhas de múltiplas tabelas com base em chaves correspondentes. Em RDF, as relações são inerentes ao modelo de dados; a recuperação de informações relacionadas é feita por meio da navegação de arestas no grafo, utilizando padrões de triplas em consultas SPARQL. Um mapeador objeto-grafo deve abstrair a lógica de construção desses padrões de grafo, que é conceitualmente diferente da construção de "JOINs" em SQL (ANGLES; GUTIERREZ, 2008).

Essas distinções fundamentais justificam a necessidade de frameworks projetados especificamente para o mapeamento objeto-grafo, pois as premissas sobre as quais os ORMs tradicionais foram construídos não se aplicam diretamente ao paradigma da Web Semântica.

2.5.5 Comparação e a Ponte para o Ecossistema Python: SQLAlchemy

Enquanto a JPA representa o padrão no ecossistema Java, o ambiente Python possui sua própria ferramenta proeminente que, embora não seja uma especificação formal, tornou-se o padrão *de facto* para o Mapeamento Objeto-Relacional: o **SQLAlchemy**. A análise comparativa entre eles não apenas destaca diferentes filosofias de design, mas também introduz as técnicas de **metaprogramação** que fazem do SQLAlchemy uma biblioteca tão poderosa e extensível — um tema que será aprofundado na seção 2.6 para fundamentar a arquitetura da solução proposta neste trabalho.

O **SQLAlchemy** é uma biblioteca ORM amplamente adotada no ecossistema Python. Ela oferece duas camadas principais: o *Core*, que permite a construção de consultas SQL de forma programática, e a camada ORM, que realiza o mapeamento objeto-relacional por meio de classes Python decoradas com metadados (GRINBERG, 2018). Diferentemente do JPA, o SQLAlchemy valoriza a flexibilidade e a composição explícita das consultas. Enquanto JPA abstrai muitas decisões por meio de convenções e anotações, o SQLAlchemy oferece um controle mais granular da geração de SQL.

O **Entity Framework** é a principal solução ORM da plataforma .NET. Assim como o JPA, ele utiliza anotações e arquivos de configuração para realizar o mapeamento entre entidades e tabelas, suportando LINQ (Language Integrated Query) para consultas fortemente tipadas dentro da linguagem C# (Microsoft Docs, 2024). Ambos promovem o desenvolvimento orientado a domínio (DDD), mas o Entity Framework possui uma integração mais direta com ferramentas da Microsoft, enquanto o JPA se destaca pela portabilidade entre provedores de persistência.

No contexto de **dados semânticos**, ferramentas como o **RDFAlchemy** estendem o conceito de ORM para grafos RDF. Baseado em RDFLib, o RDFAlchemy permite mapear classes Python para triplas RDF, utilizando URIs como identificadores e relacionamentos. Apesar de seguir a mesma filosofia de abstração dos ORMs tradicionais, o RDFAlchemy apresenta limitações significativas. Seu suporte é restrito a operações básicas conhecidas como *CRUD* (*Create*, *Read*, *Update*, *Delete*), que consistem, respectivamente, na criação, leitura, atualização e remoção de dados. Essas operações representam o conjunto fundamental de interações com bancos de dados, e sua limitação implica em menor expressividade e controle sobre relacionamentos complexos, regras de integridade ou validações avançadas. Além disso, o RDFAlchemy carece de integração com mecanismos de validação como SHACL (*Shapes Constraint Language*) e sofre com baixa manutenção do projeto, comprometendo sua viabilidade para uso em ambientes produtivos mais exigentes (DODDS, 2012).

Essa abordagem difere significativamente das soluções tradicionais como o **Java Persistence API (JPA)**, que operam sobre bancos de dados relacionais estruturados e oferecem suporte robusto a relacionamentos, herança, validações e mecanismos de transações. Enquanto o RDFAlchemy foca na simplicidade e na representação mínima de grafos RDF, o JPA proporciona uma infraestrutura madura e extensível, voltada para sistemas corporativos com demandas elevadas de confiabilidade e escalabilidade.

A principal distinção entre os ORMs convencionais (JPA, SQLAlchemy, Entity Framework) e os orientados a RDF (como RDFAlchemy) está na natureza do modelo de dados. Enquanto o modelo relacional é baseado em tabelas com esquemas fixos e chaves primárias, o modelo RDF é fundamentado em grafos de triplas que representam relacionamentos semânticos entre entidades. Isso implica em desafios adicionais, como a necessidade de mapeamento para URIs, suporte a vocabulários ontológicos e inferência semântica.

Atualmente, há poucos ORMs verdadeiramente voltados à Web Semântica. Iniciativas como o **Empusa** e **Sparql-Generate** focam em transformação e geração de grafos, mas não fornecem uma abstração completa de persistência. Assim, o desenvolvimento de soluções como **RDFMapper** busca preencher essa lacuna ao oferecer um mapeamento declarativo de classes Python para triplas RDF, com suporte a validação, consultas dinâmicas e integração com repositórios semânticos.

Dessa forma, embora JPA e seus equivalentes sejam robustos para ambientes relacionais, seu uso direto em contextos de dados conectados e Web Semântica é limitado, exigindo novas abordagens que combinem os benefícios da orientação a objetos com os princípios da modelagem semântica.

Finalmente, para além do mapeamento de dados, um dos maiores benefícios dos ORMs maduros é a capacidade de se integrarem a mecanismos de validação. No ecossistema

Java, por exemplo, a especificação *Bean Validation* permite que as mesmas anotações que definem o modelo de domínio sejam usadas para validar os dados antes mesmo de uma transação com o banco de dados ser iniciada, garantindo a integridade diretamente na camada da aplicação (BAUER; KING, 2015).

De forma análoga, o ecossistema da Web Semântica possui uma solução padronizada e robusta para a validação de grafos: a linguagem **SHACL (*Shapes Constraint Language*)** (W3C RDF Data Shapes Working Group, 2017). Portanto, uma solução de mapeamento objeto-RDF de vanguarda não deve se limitar a traduzir objetos em triplas; ela deve também incorporar a validação via SHACL como parte do seu ciclo de vida. Isso permitiria que um objeto fosse validado contra uma "forma" (shape) SHACL correspondente antes de ser persistido no grafo. Esta integração não apenas garante a qualidade e a conformidade dos dados gerados com as regras do domínio, mas também eleva a abstração a um novo patamar de robustez, espelhando as melhores práticas já consolidadas no desenvolvimento de software tradicional e oferecendo feedback de validação imediato ao desenvolvedor.

2.6 Metaprogramação em Python

Metaprogramação é uma técnica que permite que programas tenham acesso à sua própria estrutura ou comportamento em tempo de execução, podendo modificá-los dinamicamente. Em outras palavras, trata-se da capacidade de um programa manipular código como dados, criando, alterando ou inspecionando suas próprias classes, funções e atributos durante a execução (ANCONA; LAGORIO; ZUCCA, 2007; FOWLER, 2010).

Em Python, a metaprogramação é habilitada principalmente por três mecanismos poderosos:

- **Reflexão**: permite a inspeção e modificação de objetos e seus atributos durante a execução. Módulos como `inspect`, funções como `getattr()`, `hasattr()` e `setattr()` são exemplos de ferramentas de reflexão em Python.
- **Decoradores**: são funções de ordem superior que permitem modificar o comportamento de funções ou classes sem alterar seu código-fonte original. São amplamente utilizados para logging, validação, controle de acesso, entre outros.
- **Metaclasses**: são classes de classes, ou seja, definem o comportamento da criação de classes em Python. Através de metaclasses, é possível interceptar e personalizar a definição de novas classes, permitindo a injeção de métodos, propriedades ou validações automaticamente.

A metaprogramação em Python tem papel crucial na construção de frameworks e bibliotecas que exigem alto nível de abstração e extensibilidade. Por exemplo, frameworks como `Django` e `SQLAlchemy` fazem uso intensivo de metaclasses e decoradores para mapear objetos Python a estruturas de banco de dados de forma dinâmica e declarativa (LUTZ, 2013). Embora poderosa, a metaprogramação deve ser usada com cautela, pois pode dificultar a legibilidade e depuração do código. Ainda assim, quando bem aplicada, permite a construção de APIs mais expressivas, reutilizáveis e adaptáveis, promovendo maior flexibilidade no desenvolvimento de sistemas complexos.

2.6.1 SQLAlchemy: Mapeamento Objeto-Relacional com Metaclasses

O `SQLAlchemy` é um dos principais frameworks ORM (Object-Relational Mapping) em Python, amplamente utilizado na indústria e na academia. Ele emprega metaclasses e reflexão para construir dinamicamente mapeamentos entre classes Python e tabelas relacionais, possibilitando a criação de modelos de dados ricos e expressivos.

Durante a definição de um modelo com `SQLAlchemy`, a metaclasse `DeclarativeMeta` é utilizada para processar os atributos da classe e gerar metadados internos. Isso permite construir esquemas de banco de dados, consultar registros e realizar operações transacionais com alta flexibilidade, sem abrir mão da segurança do tipo e da legibilidade declarativa.

2.6.2 Django: Componentização e Validação Automática com Decoradores

O framework web `Django` utiliza decoradores e metaclasses extensivamente, tanto no mapeamento de modelos como na construção de rotas, validação de formulários e sistemas de autenticação. Decoradores como `@login_required` e `@permission_required` encapsulam lógica comum de segurança, simplificando a aplicação de políticas de acesso.

Além disso, as classes base de modelos herdam comportamentos de metaclasses como `ModelBase`, que geram os metadados necessários para comunicação com o banco de dados e verificação automática de integridade. O uso combinado de decoradores e metaclasses torna o Django um exemplo robusto de como a metaprogramação pode gerar sistemas altamente reutilizáveis.

2.6.3 Pydantic: Validação de Dados com Metaclasses e Anotações

O `Pydantic` é uma biblioteca moderna que utiliza metaprogramação para validação de dados baseada em tipos. Ela permite que modelos sejam definidos com anotações de tipo padrão de Python (usando o módulo `typing`) e utiliza metaclasses para interpretar e validar automaticamente esses tipos no momento da criação da instância.

Com suporte integrado a `JSON Schema` e integração com frameworks como `FastAPI`, o `Pydantic` demonstra como a metaprogramação pode ser usada para alinhar rigor de tipos com a simplicidade da sintaxe Python, promovendo legibilidade e segurança.

2.7 Trabalhos Relacionados

Nesta seção são apresentados e descritos projetos que compartilham objetivos semelhantes ao do *RDF Mapper*, abrangendo desde mapeamento objeto-tríplice até serialização, consultas SPARQL e validação de grafos RDF. As abordagens variam em complexidade, maturidade, escopo e linguagem de implementação.

- **Surfrdf**: Biblioteca orientada a objetos para Python que mapeia classes OWL em objetos e permite a geração automática de consultas SPARQL. É fortemente integrada com o Virtuoso, um triplestore robusto que oferece suporte a SPARQL e ontologias OWL, mas que exige instalação e configuração próprias. Muito utilizada em ambientes acadêmicos. Seu foco está na aderência a ontologias OWL e suporte à inferência semântica. Contudo, apresenta uma curva de aprendizado elevada, e depende fortemente do triplestore.

- **RDFLib**: Considerada a biblioteca padrão para manipulação de dados RDF em Python, oferece funcionalidades completas para construção e serialização de grafos, suporte a SPARQL, e integração com diversas sintaxes (Turtle, RDF/XML, N-Triples, JSON-LD). Apesar da robustez, exige muito código manual para representar modelos de dados mais complexos, dificultando a produtividade em projetos de médio e grande porte.

- **RDFAlchemy**: Camada ORM baseada em RDFLib que permite o mapeamento de classes Python para triplas RDF. Utiliza uma sintaxe semelhante à do SQLAlchemy, o que facilita a adoção por desenvolvedores familiarizados com ORMs. Entretanto, o projeto sofre com baixa manutenção, possui limitações na modelagem de relacionamentos mais complexos e não suporta validação de dados ou integrações modernas.

- **pySHACL**: Ferramenta para validação de grafos RDF com base na linguagem SHACL. É compatível com o padrão W3C, incluindo suporte a SHACL avançado e SHACL-SPARQL. Embora não ofereça funcionalidades de mapeamento objeto-tríplice ou consultas, é amplamente utilizada para garantir integridade semântica dos dados RDF após o mapeamento ou ingestão.

- **Empusa**: Ferramenta baseada em Java para geração automática de código a partir de ontologias OWL. Seu foco está na geração de APIs e documentos a partir de

ontologias, com suporte a RDF4J e SHACL. Muito usada em ambientes corporativos e científicos, oferece também verificação de coerência semântica, mas requer ontologias completas e bem definidas.

- **Karma**: Plataforma desenvolvida pelo ISI da Universidade do Sul da Califórnia que permite mapear dados relacionais, CSV ou JSON para RDF. Utiliza aprendizado de máquina para sugerir mapeamentos com base em ontologias conhecidas. Seu foco está na transformação e publicação de dados ligados, com suporte a modelagem visual. Contudo, não é orientado a código e depende de interface gráfica.

- **Empire**: Framework ORM em Java que permite o mapeamento de objetos Java para triplas RDF, baseado em anotações JPA. Suporta triplestores como Virtuoso, Sesame (atual RDF4J), entre outros. Seu diferencial está na integração com ferramentas do ecossistema Java EE e no suporte a SPARQL queries diretamente nas entidades. Apesar disso, é uma ferramenta pouco mantida atualmente.

- **Apache Jena**: Biblioteca Java amplamente consolidada para manipulação de RDF. Oferece APIs para construção de grafos, SPARQL, inferência, persistência e integração com ontologias OWL. Embora não seja um ORM, é uma das bibliotecas mais completas e utilizadas no mundo para desenvolvimento semântico em Java.

Tabela Comparativa dos Trabalhos Relacionados

Ferramenta	**Linguagem**	**Mapeamento Objeto**	**SPARQL Integrado**	**Diferenciais / Limitações**
Surfrdf	Python	Sim (OWL)	Sim	Foco acadêmico, uso acoplado ao Virtuoso
RDFLib	Python	Não	Sim	Extensa, porém exige muito código manual
RDFAlchemy	Python	Sim (limitado)	Parcial	Baixa manutenção, pouco flexível
pySHACL	Python	Não	Não	Foco exclusivo em validação com SHACL
Empusa	Java	Geração de código	Não	Geração automática de API a partir de OWL
Karma	Web/Java	Visual (sem código)	Não	Usa machine learning para mapeamento RDF
Empire	Java	Sim (JPA)	Sim	Integração com ecossistema Java EE
Apache Jena	Java	Não (manual)	Sim	Biblioteca completa, com suporte a inferência e OWL

Tabela 1 – Comparação entre ferramentas relacionadas ao mapeamento e manipulação de dados RDF

A Tabela 1 apresenta um panorama das principais ferramentas do ecossistema de dados conectados, comparando-as quanto ao suporte ao mapeamento objeto-grafo, integração com SPARQL e características distintivas.

- **Amplitude versus especialização:** Algumas ferramentas, como **Apache Jena** e **RDFLib**, são amplamente utilizadas devido à sua robustez e flexibilidade. No entanto, por serem generalistas, exigem maior esforço do desenvolvedor na modelagem e integração de funcionalidades avançadas, como validação e mapeamento automático de objetos.
- **Mapeamento objeto-grafo:** Soluções como **Surfrdf**, **RDFAlchemy** e **Empire** avançam no sentido de abstrair o grafo RDF para modelos orientados a objetos, embora enfrentem desafios como dependência de triplestores específicos, baixa manutenção ou limitação de escopo.

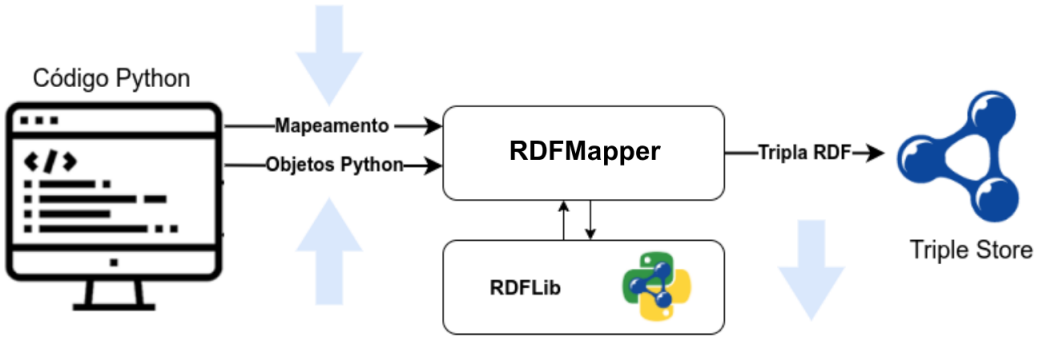
- **Validação e conformidade: pySHACL** destaca-se por sua aderência ao padrão SHACL, garantindo a integridade dos dados, mas não oferece recursos para mapeamento de objetos ou integração de consultas.
- **Abordagem declarativa e integração:** Ferramentas como **Empusa** e **Karma** apostam na geração de código a partir de ontologias ou no mapeamento visual, facilitando a integração inicial, mas podendo limitar a flexibilidade e o controle fino sobre a lógica de negócio.
- **Integração com ecossistemas: Empire** e **Surfrdf** destacam-se por integrar tecnologias estabelecidas como JPA e Virtuoso, respectivamente, embora essa abordagem também gere dependências e complexidade adicional para adoção em novos projetos.

A análise mostra que, embora existam opções para diferentes demandas, nenhuma solução entrega, de forma equilibrada, (i) uma API declarativa, (ii) suporte robusto à validação com SHACL, (iii) integração transparente com grafos RDF, (iv) consulta avançada via SPARQL e (v) flexibilidade para personalização em Python. Esta lacuna motiva o desenvolvimento do **RDFMapper**, cuja proposta é combinar o melhor desses mundos: simplicidade para o desenvolvedor, aderência a padrões da Web Semântica e extensibilidade para aplicações reais.

3 Metodologia

Este trabalho adota uma abordagem aplicada e experimental, cujo objetivo central é o desenvolvimento, implementação e avaliação de uma biblioteca denominada *RDFMapper* [1]. A proposta visa aproximar o paradigma orientado a objetos da modelagem semântica baseada em RDF.

Figura 4 – Arquitetura do RDFMapper.



Fonte: Autoria própria.

A abordagem adotada é declarativa, inspirada em frameworks objeto-relacional (ORM), como Hibernate (Java), Entity Framework (C#) e SQLAlchemy (Python), facilitando o desenvolvimento e a persistência de dados semânticos. A biblioteca está sendo implementada na linguagem Python, e a Figura 4 ilustra o funcionamento do processo de mapeamento proposto.

3.1 Tipo de Pesquisa e Abordagem

O trabalho caracteriza-se como uma pesquisa aplicada, voltada à resolução de um problema prático — a serialização e manipulação eficiente de dados RDF a partir de modelos orientados a objetos. O método científico utilizado é predominantemente experimental, envolvendo a construção, experimentação e comparação de soluções.

3.2 Ferramentas e Tecnologias Utilizadas

O desenvolvimento foi realizado utilizando a linguagem Python (versão 3.10+), em virtude de sua flexibilidade, amplo suporte a bibliotecas de dados semânticos e integração com ferramentas consolidadas. Destacam-se entre os recursos adotados:

[1] O código-fonte do projeto está disponível no GitHub: <https://github.com/felipestgoiabeira/RDFMapper>

- **rdflib**: biblioteca padrão para manipulação e serialização de grafos RDF;
- **pySHACL**: ferramenta para validação de restrições semânticas com SHACL;
- **Pandas** e **Matplotlib**: apoio à análise e visualização de dados experimentais;
- **Datasets públicos reais**, como bases acadêmicas e dados federais de preços de combustíveis, para validação do framework em cenários práticos.

3.3 Etapas do Processo Metodológico

O processo metodológico foi estruturado nas seguintes etapas:

1. **Levantamento de requisitos e análise do estado da arte**: Estudo comparativo de frameworks ORM e bibliotecas RDF para fundamentar as escolhas arquiteturais e identificar limitações das soluções existentes.
2. **Modelagem do framework**: Definição dos componentes centrais, arquitetura de metaprogramação e mecanismos de anotação, priorizando compatibilidade com boas práticas de orientação a objetos e aderência aos padrões da Web Semântica.
3. **Implementação incremental**: Desenvolvimento dos módulos principais do *RDF-Mapper*, com validação contínua por meio de testes unitários e integração.
4. **Validação e experimentação**: Execução de experimentos para avaliação quantitativa (tempo de execução, consumo de memória, linhas de código) e qualitativa (facilidade de uso, curva de aprendizado, percepção de produtividade) em comparação à abordagem tradicional com `rdflib`.

3.4 Critérios de Avaliação e Análise

Os critérios de avaliação empregados contemplam:

- **Produtividade**: Considerando tanto métricas objetivas (linhas de código, tempo de implementação) quanto subjetivas (facilidade de abstração via decoradores, aderência ao paradigma OO e curva de aprendizado).
- **Desempenho**: Avaliação do tempo de serialização, consumo de memória e escalabilidade com diferentes volumes de dados.
- **Aderência semântica**: Capacidade de gerar grafos RDF válidos e de validar restrições SHACL automaticamente.
- **Comparação experimental**: Benchmark com a abordagem de manipulação manual de triplas utilizando `rdflib` puro.

3.5 Justificativas das Escolhas Metodológicas

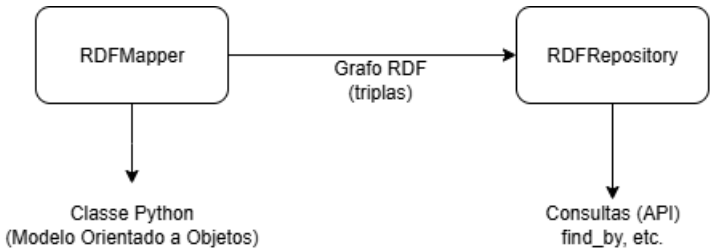
A opção pelo desenvolvimento de um framework orientado a objetos fundamenta-se no potencial de reuso, clareza semântica e facilidade de integração, características já consolidadas em frameworks ORM da indústria. A escolha da linguagem Python decorre de sua expressividade e suporte nativo a bibliotecas para dados conectados. A inspiração em padrões como JPA e SQLAlchemy busca tornar a abordagem familiar ao desenvolvedor, reduzindo a curva de aprendizado e facilitando a abstração das anotações via decoradores. Por fim, a utilização de datasets reais, em especial bases abertas e volumosas, visa comprovar a robustez e aplicabilidade da solução proposta em cenários práticos.

4 Desenvolvimento e Arquitetura do RDF-Mapper

O pacote `rdfmapper` foi concebido para fornecer uma camada de abstração orientada a objetos sobre o modelo RDF, permitindo que desenvolvedores manipulem dados semânticos de forma mais intuitiva e expressiva. A seguir, são descritos os principais componentes e funcionalidades desenvolvidas na biblioteca.

4.1 Estrutura e Componentes do RDFMapper

Figura 5 – Arquitetura Biblioteca RDFMapper



Fonte: Autoria própria.

A Figura 5 apresenta a arquitetura central da biblioteca, destacando seus dois componentes fundamentais: **RDFMapper** e **RDFRepository**. O **RDFMapper** é responsável pelo mapeamento, serialização e desserialização entre classes Python e triplas RDF, abstraindo a complexidade do modelo de grafos e facilitando a integração com aplicações orientadas a objetos. O **RDFRepository**, por sua vez, provê uma camada de acesso baseada em padrões inspirados no JPA, permitindo consultas dinâmicas, filtragem, agregação e paginação diretamente sobre o grafo RDF. Essa divisão modular permite ao desenvolvedor modelar, persistir e consultar dados semânticos de maneira eficiente, alinhando flexibilidade semântica à produtividade do desenvolvimento orientado a objetos.

4.1.1 RDFMapper

É o núcleo do framework, responsável por registrar classes, propriedades e relacionamentos RDF a partir de anotações decorativas nas classes Python. Também implementa os métodos de serialização (`to_rdf`) e desserialização (`from_rdf`), transformando instâncias de objetos Python em grafos RDF e vice-versa.

Durante a serialização, o `RDFMapper` também trata questões como relacionamentos entre objetos e circularidade, utilizando um conjunto interno de objetos já visitados para evitar recursões infinitas em grafos com estruturas cíclicas.

4.1.2 Decoradores de Mapeamento

O framework utiliza decoradores personalizados para declarar metadados semânticos nas classes e atributos Python. Estes decoradores são os seguintes:

- `@rdf_entity(uri)`: define que uma classe corresponde a um tipo RDF específico.
- `@rdf_property(uri, required=False)`: associa um atributo a um predicado RDF, podendo ser marcado como obrigatório.
- `@rdf_one_to_one(uri)`: indica um relacionamento do tipo 1:1 entre duas entidades RDF.
- `@rdf_one_to_many(uri)`: representa relacionamentos do tipo 1:N, armazenando múltiplas instâncias relacionadas.

O uso desses decoradores permite que o desenvolvedor modele sua aplicação de forma declarativa e expressiva, aproximando o paradigma orientado a objetos das estruturas RDF.

4.1.3 RDFRepository

Inspirado nos repositórios encontrados em frameworks como o Spring Data JPA, o `RDFRepository` provê uma API de consulta flexível baseada em convenções de nomenclatura. Entre os métodos disponíveis, destacam-se:

- `find_by_`: busca por propriedades exatas.
- `find_by__`*`like`*: busca parcial por similaridade de string.
- `count_by_`: retorna o número de instâncias com base em filtros.
- Suporte a paginação por meio dos parâmetros `limit` e `offset`.

As consultas são realizadas sobre instâncias armazenadas em um grafo RDF interno, mas o repositório pode ser estendido futuramente para interagir com triplestores como Jena ou Virtuoso.

4.2 Mapeamento de Entidades

4.2.1 Decorador `@rdf_entity`

O decorador `@rdf_entity(uri)` é utilizado para indicar que uma determinada classe Python representa uma **entidade RDF**, ou seja, uma instância de um tipo específico definido em um vocabulário ou ontologia.

Ao aplicar esse decorador, o `RDFMapper` registra a classe como um recurso RDF do tipo especificado no parâmetro `uri`, adicionando automaticamente a tripla de tipo (`rdf:type`) durante o processo de serialização.

Exemplo de uso

Código 4 Uso do decorador @rdf_entity para mapear uma classe

```
@rdf_mapper.rdf_entity(EX.Person)
class Person:
    ...
```

Código 5 Uso do decorador @rdf_entity para mapear uma classe

```
repo.group_by_count(TCC, "curso", order="DESC")
repo.group_by_count(TCC, "modalidade", order="DESC")
repo.group_by_count(TCC, "municipio", order="desc")
repo.group_by_count(TCC, "mes", order="DESC")
```

No exemplo de uso, a classe `Person` será mapeada como uma instância do tipo `ex:Person`. Quando um objeto dessa classe for serializado, será automaticamente adicionada ao grafo a seguinte tripla:

Código 6 Tripla rdf:type gerada pelo decorador @rdf_entity

```
<http://example.org/person/1> rdf:type ex:Person .
```

Esse mecanismo permite alinhar classes Python a conceitos RDF, mantendo a coerência entre o modelo orientado a objetos e a estrutura semântica dos dados.

Além disso, o decorador `@rdf_entity` armazena metadados necessários para o funcionamento da serialização reversa, possibilitando que grafos RDF sejam desserializados corretamente de volta em instâncias Python, desde que o tipo RDF esteja presente no grafo de origem.

4.2.2 Decorador `@rdf_property`

O decorador `@rdf_property(uri, required=False)` é utilizado para associar atributos de uma classe Python a **propriedades RDF** (predicados). Ele estabelece um mapeamento direto entre um atributo interno da classe (por exemplo, `_nome`) e um predicado RDF (como `ex:nome`), permitindo que o valor do atributo seja serializado como uma tripla RDF.

Esse decorador é aplicado sobre o nome do método de acesso (getter), e internamente utiliza o mecanismo de `property` do Python para encapsular os dados, mantendo a compatibilidade com boas práticas de orientação a objetos.

4.2.2.1 Funcionamento

Ao decorar um método com `@rdf_property`, o `RDFMapper`:

Código 7 Uso do decorador @rdf_property com o parâmetro required

```
@rdf_mapper.rdf_entity(EX.Person)
class Person:
    def __init__(self, uri, nome=None):
        self.uri = uri
        self._nome = nome

    @rdf_mapper.rdf_property(EX.nome, required=True)
    def nome(self): pass
```

1. Registra o predicado RDF correspondente.
2. Marca a propriedade como não relacional (primitiva, como `str`, `int`, `bool`, etc.).
3. Permite marcar campos como obrigatórios via `required=True`.
4. Utiliza um atributo de instância prefixado com `_` para armazenar o valor real (ex: `_nome` para `nome`).

No exemplo, esse código produz, na serialização, a seguinte tripla (supondo que o valor seja "João")

Código 8 Tripla de propriedade literal gerada pelo @rdf_property

```
<http://example.org/person/1> ex:nome "João" .
```

4.3 Mapeamento de Relacionamentos

No contexto do RDFMapper, o mapeamento de relacionamentos entre objetos Python que referenciam outras entidades RDF é realizado por meio dos decoradores

`@rdf_one_to_one` e `@rdf_one_to_many`. Esses decoradores permitem representar, de forma declarativa, ligações entre recursos RDF com cardinalidade controlada.

4.3.1 `@rdf_one_to_one`

Esse decorador é utilizado para representar um **relacionamento 1:1**, ou seja, quando um objeto de uma classe contém a referência direta a uma única instância de outra classe.

4.3.1.1 Funcionamento

Ao decorar um método com `@rdf_one_to_one`, o RDFMapper:

- Associa o atributo a um predicado RDF que será utilizado para representar o relacionamento.
- Armazena o tipo de relacionamento como `'one_to_one'`.
- Registra a **classe de destino** por meio do parâmetro `target_class`, normalmente passado como um `lambda`.

Durante a serialização:

- É adicionada uma tripla com o predicado fornecido, tendo como objeto o `URIRef` da entidade relacionada.
- A entidade relacionada também é serializada recursivamente, exceto se já tiver sido visitada (para evitar loops).

Código 9 Mapeando um relacionamento 1:1 com @rdf_one_to_one

```
1 @rdf_mapper.rdf_one_to_one(EX.address, target_class=lambda: Address)
2 def endereco(self): pass
```

O exemplo de uso gera uma tripla como:

Código 10 Tripla de relacionamento gerada pelo @rdf_one_to_one

```
1 <http://example.org/person/1> ex:address <http://example.org/address/10> .
```

E serializa também o recurso `<http://example.org/address/10>` com suas propriedades.

4.3.2 `@rdf_one_to_many`

O decorador `@rdf_one_to_many` representa um **relacionamento 1:N**, ou seja, uma entidade relacionada a uma **lista de objetos** de outra classe.

4.3.2.1 Funcionamento

O processo é semelhante ao do `@rdf_one_to_one`, com as seguintes diferenças:

- O atributo associado deve ser uma **lista** (ou outra coleção iterável).
- Cada item da lista é serializado separadamente e também incluído no grafo.
- Para cada item, uma tripla é adicionada com o predicado fornecido.

Código 11 Mapeando um relacionamento 1:N com @rdf_one_to_many

```
1 @rdf_mapper.rdf_one_to_many(EX.phone, target_class=lambda: Phone)
2 def telefones(self): pass
```

O exemplo de uso gera múltiplas triplas como:

Código 12 Múltiplas triplas geradas pelo relacionamento 1:N

```
1 <http://example.org/person/1> ex:phone <http://example.org/phone/1> .
2 <http://example.org/person/1> ex:phone <http://example.org/phone/2> .
```

E inclui a serialização completa de

`<http://example.org/phone/1>` e `<http://example.org/phone/2>`.

4.3.2.2 Detalhes Técnicos da Implementação

Ambos os decoradores utilizam o mecanismo de `property` do Python para encapsular o acesso, da seguinte forma:

- O nome do método decorado é usado como nome do atributo lógico (ex: `telefone`), mas os dados são armazenados internamente com um prefixo (`_telefone`).
- Os metadados do relacionamento (tipo, predicado, classe de destino) são armazenados como atributos da função getter (`_is_relationship`, `_relationship_type`, `_target_class`).
- Durante a serialização (`to_rdf`), esses metadados são lidos dinamicamente via introspecção.

- A serialização recursiva trata automaticamente relacionamentos aninhados e circularidade.

A abordagem proposta permite uma modelagem expressiva de dados interligados, tornando mais natural a representação de relações complexas no universo semântico. Além disso, facilita a manutenção da estrutura dos grafos RDF, pois as entidades Python servem como base para a definição e atualização dos dados, promovendo maior clareza e reutilização do código. Outro ponto relevante é a possibilidade de extensão futura, já que o modelo adotado pode incorporar, de forma incremental, a validação de restrições de cardinalidade definidas em OWL, ampliando ainda mais o rigor semântico e a aderência aos padrões da Web Semântica.

4.4 Consultas Dinâmicas

Um dos diferenciais do RDFMapper em relação a outras bibliotecas RDF em Python é a capacidade de executar **consultas dinâmicas inspiradas no padrão JPA**, como `find_by_nome`, `find_by_nome_like`, `count_by_nome`, entre outras. Essas consultas funcionam sobre grafos RDF carregados em memória (via `rdflib.Graph`), utilizando a linguagem SPARQL de forma transparente para o usuário.

4.4.1 Estratégia `find_by_*`

A estratégia de resolução dinâmica de métodos como `find_by_nome` é implementada na classe `RDFRepository`, utilizando o método especial `__getattr__`. Esse método intercepta chamadas a funções não declaradas explicitamente e analisa seu nome para gerar dinamicamente uma consulta SPARQL.

Código 13 Exemplo de chamada de método de consulta dinâmica

```
repo.find_by_nome("João")
```

O exemplo `repo.find_by_nome("João")` gera uma consulta SPARQL equivalente a:

Código 14 Consulta SPARQL gerada para o método find_by_nome

```
SELECT ?s WHERE {
  ?s a ex:Person ;
     ex:nome "João" .
}
```

4.4.2 Filtros parciais (`like`)

Consultas com sufixo `_like` permitem realizar **buscas por similaridade**, utilizando o operador `FILTER CONTAINS(...)` do SPARQL.

Código 15 Exemplo de consulta com filtro de similaridade (like)

```
repo.find_by_nome_like("Jo")
```

O exemplo `find_by_nome_like("Jo")` gera uma consulta SPARQL equivalente a:

Código 16 Consulta SPARQL com FILTER CONTAINS para o método find_by_nome_like

```
SELECT ?s WHERE {
  ?s a ex:Person ;
     ex:nome ?val .
  FILTER CONTAINS(LCASE(STR(?val)), "jo")
}
```

Essa funcionalidade amplia o poder de busca textual, tornando o framework mais expressivo e útil em aplicações reais.

4.4.3 Filtros compostos

É possível utilizar múltiplos campos na mesma chamada, como

Código 17 Exemplo de consulta com filtro composto (AND)

```
repo.find_by_nome_and_endereco("João", "Rua A")
```

Código 18 Consulta SPARQL com múltiplas cláusulas WHERE

```
SELECT ?s WHERE {
  ?s a ex:Person ;
     ex:nome "João" ;
     ex:endereco "Rua A" .
}
```

A biblioteca converte o nome do método e os argumentos em **cláusulas AND**, baseando-se na ordem dos parâmetros fornecidos.

4.4.4 Paginação e Contagem

Funcionalidades como **paginação** e **contagem de resultados** são essenciais para garantir desempenho e escalabilidade. O `RDFMapper` implementa essas funcionalidades de forma integrada à sua API de consulta dinâmica. Os métodos de consulta aceitam os argumentos opcionais `limit` e `offset`, que são traduzidos diretamente para as cláusulas `LIMIT` e `OFFSET` do SPARQL, como demonstrado nos exemplos a seguir.

Código 19 Exemplo de consulta com paginação (limit e offset)

```
repo.find_by_nome("João", limit=10, offset=20)
```

Gera a seguinte consulta SPARQL:

Código 20 Consulta SPARQL gerada com as cláusulas LIMIT e OFFSET

```
SELECT ?s WHERE {
  ?s a ex:Person ;
     ex:nome "João" .
}
LIMIT 10 OFFSET 20
```

Além da paginação, o framework oferece suporte à contagem de entidades que satisfazem determinados critérios, por meio de métodos dinâmicos com o prefixo `count_by_`, que geram consultas SPARQL com a função de agregação `COUNT`.

4.5 Serialização e Prevenção de Circularidade

A serialização de estruturas de dados orientadas a objetos para RDF exige atenção especial quando os objetos possuem **referências circulares**. O `RDFMapper` resolve esse problema implementando um mecanismo interno de **rastreio de objetos visitados** durante o processo de serialização (`to_rdf`) e um cache de objetos carregados durante a desserialização (`from_rdf`).

4.5.1 Estratégia de Prevenção

Durante a serialização, um conjunto de URIs de objetos já processados é mantido. Antes de serializar uma nova entidade, o framework verifica se sua URI já está neste conjunto, evitando reprocessá-la e, consequentemente, impedindo loops infinitos.

Código 21 Criação de uma referência circular entre objetos

```
person = Person(uri="ex:person1", ...)
address = Address(uri="ex:address1", ...)
person.address = address
address.resident = person  # circularidade
```

Com o controle de objetos visitados, o ciclo é detectado e o grafo é gerado de forma correta e sem redundâncias.

Código 22 Grafo RDF resultante com referência circular preservada

```
ex:person1 rdf:type ex:Person ;
           ex:address ex:address1 .

ex:address1 rdf:type ex:Address ;
            ex:resident ex:person1 .
```

4.5.2 Desserialização com Suporte a Circularidade

De forma análoga, na desserialização, a biblioteca mantém um cache de objetos já instanciados. Ao encontrar uma URI que já está no cache, o framework reutiliza a instância existente em vez de criar uma nova, preservando a identidade e a estrutura de referência circular do grafo original.

Código 23 Grafo de entrada para desserialização com referência circular

```
ex:person1 ex:address ex:address1 .
ex:address1 ex:resident ex:person1 .
```

Essa abordagem garante consistência em estruturas complexas, comuns em ontologias OWL com relacionamentos bidirecionais.

4.6 Validação de Dados com SHACL na `RDFMapper`

Para garantir a integridade dos grafos e a conformidade com o modelo de domínio, a biblioteca `RDFMapper` implementa um suporte integrado à validação utilizando a linguagem SHACL (*Shapes Constraint Language*). Essa funcionalidade permite que as restrições estruturais definidas no modelo de classes Python sejam usadas para validar as instâncias RDF de forma automatizada.

4.6.1 Geração Automática de Shapes

O núcleo da funcionalidade reside no método `to_shacl(classe)`, que inspeciona os decoradores de uma classe mapeada e gera um grafo de "shapes" (formas) SHACL correspondente. Parâmetros nos decoradores, como `minCount=1` ou anotações de tipo ("idade: int"), são traduzidos diretamente em restrições SHACL.

Por exemplo, considere a seguinte definição de classe, onde o nome é obrigatório:

Código 24 Classe Person com restrições de cardinalidade via decorador

```python
@rdf_mapper.rdf_entity(EX.Person)
class Person:
    def __init__(self, uri, nome, idade: int, email):
        self.uri = uri
        self._nome = Literal(nome)
        self._idade = Literal(idade)
        self._email = Literal(email)

    @rdf_mapper.rdf_property(FOAF.name, minCount=1)
    def nome(self): pass

    @rdf_mapper.rdf_property(FOAF.age)
    def idade(self): pass

    @rdf_mapper.rdf_property(FOAF.mbox)
    def email(self): pass
```

Ao executar `rdf_mapper.to_shacl(Person)`, a biblioteca gera o grafo de formas SHACL apresentado no Código 25. Note como as restrições da classe Python são traduzidas para a sintaxe SHACL: o decorador `@rdf_property(FOAF.name, minCount=1)` resulta na propriedade `sh:minCount 1` para o caminho ("sh:path") `foaf:name`. Da mesma forma, a anotação de tipo `idade: int` é usada para inferir o tipo de dado ("sh:datatype") como `xsd:integer`.

Código 25 Grafo de formas SHACL gerado automaticamente a partir da classe Person

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<http://example.org/shape/PersonShape> a sh:NodeShape ;
    sh:property [ sh:datatype xsd:string ;
            sh:maxCount 1 ;
            sh:minCount 0 ;
            sh:path foaf:mbox ],
        [ sh:datatype xsd:integer ;
            sh:maxCount 1 ;
            sh:minCount 0 ;
            sh:path foaf:age ],
        [ sh:datatype xsd:string ;
            sh:maxCount 1 ;
            sh:minCount 1 ;
            sh:path foaf:name ] ;
    sh:targetClass <http://example.org/Person> .
```

4.6.2 Processo de Validação e Análise do Relatório

O método `validate` utiliza o grafo de shapes (fornecido externamente ou gerado automaticamente) para validar um grafo de dados por meio da biblioteca `pySHACL`. Se um objeto é criado de forma inconsistente com as regras, a validação falha.

Código 26 Executando a validação SHACL sobre um objeto inválido

```python
# Instância inválida, pois 'nome' é obrigatório (minCount=1)
person = Person(
    uri="https://example.org/persons/1",
    nome=None,          # Violação da regra minCount=1
    idade=25,
    email="joao@example.com"
)

person_graph = rdf_mapper.to_rdf(person)
conforms, report_graph, report_text = rdf_mapper.validate(
    person_graph, entity_class=Person
)
```

Quando uma violação ocorre, o método retorna "conforms=False" e um relatório detalhado. O Código 27 mostra um exemplo desse relatório.

Código 27 Exemplo de relatório de validação para um grafo inconsistente

```
Constraint Violation in DatatypeConstraintComponent
↪  (http://www.w3.org/ns/shacl#DatatypeConstraintComponent):
    Severity: sh:Violation
    Source Shape: [ sh:datatype xsd:string ; sh:maxCount Literal("1"...) ; sh:minCount
    ↪  Literal("1"...) ; sh:path foaf:name ]
    Focus Node: <https://example.org/persons/1>
    Value Node: Literal("None")
    Result Path: foaf:name
    Message: Value is not Literal with datatype xsd:string
```

A análise detalhada do relatório fornece insights precisos sobre o erro:

Focus Node Aponta para a URI do recurso no grafo de dados que falhou na validação ("<https://example.org/persons/1>").

Result Path Mostra a propriedade específica que causou o erro ("foaf:name").

Source Shape Descreve a regra exata que foi violada (neste caso, a propriedade deveria ter "minCount" de 1 e um "datatype‘ de ‘xsd:string").

Message Apresenta uma mensagem legível explicando o problema ("*Value is not Literal with datatype xsd:string*", pois o valor "None" foi convertido para um literal sem tipo).

Essa integração entre a modelagem em Python e a validação semântica com SHACL confere grande robustez ao processo de geração de dados, garantindo que o grafo RDF produzido esteja em conformidade com as regras de negócio definidas na própria aplicação.

4.7 Guia de Utilização Prática do RDFMapper

Para demonstrar o fluxo de trabalho e a simplicidade de uso do `RDFMapper`, esta seção apresenta um guia prático passo a passo. O objetivo é levar um desenvolvedor desde a modelagem de classes em Python até a execução de consultas sobre os dados RDF gerados, ilustrando as principais funcionalidades da biblioteca.

4.7.1 Passo 1: Configuração Inicial

O primeiro passo consiste em importar os componentes necessários e instanciar o `RDFMapper`, que atuará como o núcleo para o registro das classes e para as operações de mapeamento.

Código 28 Configuração inicial do ambiente e do RDFMapper

```
from rdflib import Namespace, Literal
from rdf_mapper import RDFMapper

# Definição de namespaces para os vocabulários
EX = Namespace("http://example.org/")
FOAF = Namespace("http://xmlns.com/foaf/0.1/")

# Instanciação do mapeador principal
rdf_mapper = RDFMapper()
```

4.7.2 Passo 2: Modelagem das Classes com Decoradores

A etapa de modelagem é onde a semântica dos dados é definida. Utilizando os decoradores do `RDFMapper`, associamos classes e atributos Python a conceitos de ontologias RDF. No exemplo a seguir, definimos as classes “Endereco” e “Pessoa”, incluindo um atributo literal (“nome”) e um relacionamento (“endereco”).

Código 29 Modelagem de classes com decoradores de mapeamento

```
@rdf_mapper.rdf_entity(EX.Endereco)
class Endereco:
    def __init__(self, uri, logradouro):
        self.uri = uri
        self._logradouro = Literal(logradouro)

    @rdf_mapper.rdf_property(EX.logradouro)
    def logradouro(self): pass

@rdf_mapper.rdf_entity(EX.Pessoa)
class Pessoa:
    def __init__(self, uri, nome, endereco):
        self.uri = uri
        self._nome = Literal(nome)
        self._endereco = endereco

    @rdf_mapper.rdf_property(FOAF.name)
    def nome(self): pass

    @rdf_mapper.rdf_one_to_one(EX.moradia, target_class=lambda: Endereco)
    def endereco(self): pass

```

4.7.3 Passo 3: Criando Instâncias e Serializando para RDF

Com as classes modeladas, podemos criar instâncias como faríamos com qualquer objeto Python. Em seguida, o método `to_rdf()` é utilizado para converter o objeto (e seus objetos relacionados) em um grafo `rdflib.Graph`.

Código 30 Instanciação de objetos e serialização para um grafo RDF

```
# Criação das instâncias
end = Endereco(EX["endereco/1"], "Rua Central, 123")
pes = Pessoa(EX["pessoa/1"], "Ana Maria", end)

# Serialização do objeto 'pes' para um grafo RDF
grafo_rdf = rdf_mapper.to_rdf(pes)

# Opcional: imprimir o grafo no formato Turtle
print(grafo_rdf.serialize(format="turtle"))
```

A execução do código acima produzirá o seguinte resultado em formato Turtle, demonstrando que o mapeamento foi realizado com sucesso:

Código 31 Resultado da serialização em formato Turtle

```
@prefix ex: <http://example.org/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

ex:endereco/1 a ex:Endereco ;
    ex:logradouro "Rua Central, 123" .

ex:pessoa/1 a ex:Pessoa ;
    foaf:name "Ana Maria" ;
    ex:moradia ex:endereco/1 .
```

4.7.4 Passo 4: Validando os Dados com SHACL

Para garantir a integridade dos dados gerados, podemos utilizar o método `validate()`. O `RDFMapper` pode gerar um "shape" SHACL automaticamente a partir da classe e validar o grafo RDF contra ele. No exemplo abaixo, a classe "Pessoa" é modificada para exigir a propriedade "nome" ("minCount=1").

Código 32 Executando a validação SHACL sobre um objeto com dados faltantes

```
# Supondo que em @rdf_property(FOAF.name) foi adicionado minCount=1

# Instância inválida (nome está faltando)
pessoa_invalida = Pessoa(EX["pessoa/2"], nome=None, endereco=end)
grafo_invalido = rdf_mapper.to_rdf(pessoa_invalida)

# Executando a validação
conforme, _, relatorio = rdf_mapper.validate(grafo_invalido, entity_class=Pessoa)

print(f"Grafo conforme as regras: {conforme}")
if not conforme:
    print(relatorio)
```

Este processo retorna um booleano indicando a conformidade e um relatório textual detalhando quaisquer violações encontradas, o que é fundamental para garantir a qualidade dos dados.

4.7.5 Passo 5: Consultando os Dados com RDFRepository

Uma vez que os dados estão em formato de grafo, o `RDFRepository` permite recuperá-los de forma intuitiva. Primeiro, o repositório é instanciado, associado a um grafo e a uma classe de entidade. Em seguida, os métodos dinâmicos podem ser chamados.

Código 33 Utilização do RDFRepository para consultas dinâmicas

```python
from rdf_mapper import RDFRepository

# Supondo que 'grafo_rdf' contém os dados de várias pessoas
repositorio_pessoas = RDFRepository(rdf_mapper, grafo_rdf, Pessoa)

# Exemplo de consulta: encontrar pessoas com o nome "Ana Maria"
resultados = repositorio_pessoas.find_by_nome(nome="Ana Maria")

for pessoa in resultados:
    print(f"Encontrado: {pessoa.uri}, Nome: {pessoa.nome}")

# Exemplo de consulta agregada
contagem_por_curso = repo_tcc.group_by_count(TCC, "curso")
print(contagem_por_curso)
```

Este guia demonstra um fluxo de trabalho completo e coeso, evidenciando como o `RDFMapper` abstrai a complexidade das tecnologias da Web Semântica e oferece uma interface de alta produtividade para o desenvolvedor Python.

5 Distribuição e Instalação

O pacote `RDFMapper` é distribuído de forma aberta, estando disponível publicamente no repositório GitHub[1]. A publicação do código-fonte em uma plataforma aberta facilita o acesso, a colaboração e o reuso por parte da comunidade acadêmica e de desenvolvedores interessados em aplicações de Dados Conectados na linguagem Python.

5.1 Instruções de Instalação

A instalação do `RDFMapper` pode ser realizada diretamente a partir do repositório GitHub, utilizando o *pip*, o gerenciador de pacotes padrão do Python. Recomenda-se a utilização de um ambiente virtual (*virtualenv* ou *venv*) para evitar conflitos de dependências.

1. Certifique-se de ter o `git` e o `pip` instalados em sua máquina.
2. (Opcional) Crie e ative um ambiente virtual:

```
python -m venv venv
source venv/bin/activate  # Linux/Mac
venv\Scripts\activate     # Windows
```

3. Instale diretamente a partir do GitHub:

```
pip install git+https://github.com/felipestgoiabeira/RDFMapper.git
```

4. Após a instalação, a biblioteca estará disponível para importação nos scripts Python:

```
from rdf_mapper import RDFMapper
```

Caso deseje contribuir com o desenvolvimento ou acessar os exemplos completos, basta realizar o *clone* do repositório:

Código 34 Comando para clonar o repositório do projeto

```
git clone https://github.com/felipestgoiabeira/RDFMapper.git
```

Após o clone, recomenda-se a instalação das dependências de desenvolvimento, caso vá executar os testes ou trabalhar na evolução do projeto:

[1] <https://github.com/felipestgoiabeira/RDFMapper>

Código 35 Instalação das dependências de desenvolvimento

```
cd RDFMapper
pip install -r requirements.txt
```

5.2 Próximos Passos

Após a instalação bem-sucedida, o usuário está pronto para começar a utilizar o `RDFMapper`. Recomenda-se seguir o tutorial apresentado na Seção 4.7 para um passo a passo detalhado sobre como modelar, serializar, validar e consultar dados. Adicionalmente, o repositório clonado contém uma pasta de exemplos com scripts completos e funcionais que demonstram as diversas funcionalidades da biblioteca em cenários práticos.

5.3 Como Contribuir

O desenvolvimento do `RDFMapper` é um processo contínuo e contribuições da comunidade são muito bem-vindas. Desenvolvedores interessados em reportar problemas (*bugs*), sugerir novas funcionalidades ou submeter melhorias no código podem fazê-lo através do sistema de *Issues* (problemas) e *Pull Requests* (solicitações de integração) do repositório oficial no GitHub. Toda colaboração que vise aprimorar a ferramenta e expandir suas capacidades será valorizada.

6 Resultados e Validação

Este capítulo apresenta os resultados obtidos com a aplicação do framework `RDFMapper`, validando sua eficácia por meio de uma comparação de produtividade, estudos de caso com dados do mundo real e uma análise de desempenho quantitativa.

6.1 Comparação de Produtividade: RDFMapper vs. RDFLib Puro

Para demonstrar os ganhos em produtividade e simplicidade proporcionados pelo RDFMapper, foi criado um cenário simples com duas classes: `Pessoa` e `Endereco`. A tarefa consiste em instanciar uma pessoa com seu respectivo endereço e serializar o grafo RDF no formato `Turtle`. As implementações a seguir ilustram a diferença de complexidade.

6.1.1 Implementação com RDFLib Puro

Código 36 Código com RDFLib puro

```
from rdflib import Graph, Namespace, URIRef, Literal, RDF

EX = Namespace("http://example.org/")
FOAF = Namespace("http://xmlns.com/foaf/0.1/")
g = Graph()
g.bind("ex", EX)
g.bind("foaf", FOAF)

pessoa_uri = URIRef(EX["pessoa/1"])
endereco_uri = URIRef(EX["endereco/1"])

g.add((pessoa_uri, RDF.type, EX.Pessoa))
g.add((endereco_uri, RDF.type, EX.Endereco))
g.add((pessoa_uri, FOAF.name, Literal("Ana Maria")))
g.add((endereco_uri, EX.logradouro, Literal("Rua Central, 123")))
g.add((pessoa_uri, EX.moradia, endereco_uri))

g.serialize(destination="baseline.ttl", format="turtle")
```

6.1.2 Implementação com RDFMapper

Código 37 Código com RDFMapper

```
from rdflib import Namespace, Literal
from src.rdf_mapper.rdf_mapper import RDFMapper

EX = Namespace("http://example.org/")
FOAF = Namespace("http://xmlns.com/foaf/0.1/")
rdf_mapper = RDFMapper()

@rdf_mapper.rdf_entity(EX.Endereco)
class Endereco:
    def __init__(self, uri, logradouro):
        self.uri = uri
        self._logradouro = Literal(logradouro)

    @rdf_mapper.rdf_property(EX.logradouro)
    def logradouro(self): pass

@rdf_mapper.rdf_entity(EX.Pessoa)
class Pessoa:
    def __init__(self, uri, nome, endereco):
        self.uri = uri
        self._nome = Literal(nome)
        self._endereco = endereco

    @rdf_mapper.rdf_property(FOAF.name)
    def nome(self): pass

    @rdf_mapper.rdf_one_to_one(EX.moradia, target_class=lambda: Endereco)
    def endereco(self): pass

end = Endereco(EX["endereco/1"], "Rua Central, 123")
pes = Pessoa(EX["pessoa/1"], "Ana Maria", end)

g = rdf_mapper.to_rdf(pes)
g.serialize(destination="mapper.ttl", format="turtle")
```

6.1.3 Análise Quantitativa e Justificativa

Tabela 2 – Comparação de linhas de código por complexidade

Métrica	**RDFLib puro**	**RDFMapper**
Código de Configuração/Modelo (reutilizável)	6 linhas	~21 linhas
Código por Instância (criação e adição)	7 linhas	2 linhas
Custo para 100 objetos (após setup)	**700 linhas**	**200 linhas**

Como a Tabela 2 demonstra, a abordagem com `RDFLib` puro exige um baixo custo inicial de configuração (apenas 6 linhas para inicializar o grafo e os namespaces). Contudo, seu custo de manutenção é linear e alto: para cada nova entidade `Pessoa` com seu `Endereco`, são necessárias aproximadamente 7 novas linhas de código (2 para URIs e 5 para as chamadas `g.add()`). Em um cenário com centenas ou milhares de registros, essa abordagem se torna verbosa, repetitiva e altamente suscetível a erros.

Por outro lado, o desenvolvimento com o `RDFMapper` requer um investimento inicial maior na modelagem das classes e na definição dos decoradores, o que pode resultar em um número maior de linhas de código na configuração do modelo de domínio. No entanto, este custo inicial é rapidamente amortizado à medida que o sistema cresce, já que a estrutura criada é robusta e altamente reutilizável. Após a definição das classes, a criação de novas instâncias demanda significativamente menos código, proporcionando maior produtividade no desenvolvimento em larga escala.

Outro aspecto relevante diz respeito à facilidade de abstração e à curva de aprendizado para o desenvolvedor. O uso de decoradores para mapeamento é uma abordagem amplamente difundida em *frameworks* modernos, como a especificação JPA, tornando o paradigma do `RDFMapper` mais natural e intuitivo para desenvolvedores já familiarizados com padrões orientados a objetos e anotações declarativas. Por consequência, a curva de aprendizado é consideravelmente menor em comparação com a manipulação manual de triplas em `RDFLib`, que exige conhecimento aprofundado dos conceitos de RDF, bem como atenção a detalhes sintáticos e semânticos na construção de cada tripla.

Além disso, do ponto de vista do desenvolvedor, o `RDFMapper` permite um alinhamento mais próximo entre o modelo de domínio da aplicação e a estrutura dos dados RDF, promovendo maior clareza, facilidade de manutenção e expressividade no código. O caráter orientado a objetos do `RDFMapper` favorece o raciocínio, a reutilização de componentes e a evolução incremental do sistema, enquanto a abordagem com `RDFLib` puro se mostra menos adequada para aplicações que demandam modelagem de domínio sofisticada.

Portanto, embora o critério de produtividade não deva se restringir ao número de linhas de código, a abordagem do `RDFMapper` oferece vantagens qualitativas adicionais. Entre elas, destacam-se a menor curva de aprendizado, maior familiaridade por parte dos desenvolvedores devido à inspiração em *frameworks* consolidados, e uma experiência de desenvolvimento mais alinhada com o paradigma orientado a objetos, aspectos não contemplados por métricas puramente quantitativas. Essas características tornam o `RDFMapper` especialmente adequado para equipes de desenvolvimento que valorizam clareza, facilidade de manutenção e evolução contínua dos sistemas.

6.2 Estudos de Caso: Aplicação com Dados Reais

6.2.1 Resultados com Dados Reais de Trabalhos de Conclusão de Curso

Para avaliar a aplicabilidade do *RDFMapper* em um cenário acadêmico real, foi utilizada a base de dados aberta “Trabalhos de Conclusão de Curso Defendidos” da Universidade Federal do Maranhão (UFMA), disponível publicamente em seu portal de

dados[1]. O conjunto de dados, em formato CSV, contém informações sobre as defesas de dissertações e teses da instituição.

O primeiro passo consistiu em modelar os dados por meio de uma classe Python. A classe `TCC`, mostrada no Código 38, foi criada para representar cada registro do dataset. Nela, cada coluna de interesse, como "municipio", "curso", "titulo" e "orientador", foi mapeada para uma propriedade RDF sob um namespace local ("EX"), utilizando os decoradores `@rdf_entity` e `@rdf_property`.

Código 38 Classe `TCC` para mapeamento dos dados de defesas

```
@rdf_mapper.rdf_entity(EX.TCC)
class TCC:
    def __init__(
        self, uri,
        municipio=None, curso=None, turno=None, modalidade=None,
        nivel=None, mes=None, ano=None,
        titulo=None, autor=None, orientador=None,
        tipo=None, data_defesa=None
    ):
        self.uri = uri
        self._municipio = municipio
        self._curso = curso
        self._turno = turno
        self._modalidade = modalidade
        self._nivel = nivel
        self._mes = mes
        self._ano = ano
        self._titulo = titulo
        self._autor = autor
        self._orientador = orientador
        self._tipo = tipo
        self._data_defesa = data_defesa

    @rdf_mapper.rdf_property(EX.municipio)
    def municipio(self): pass

    @rdf_mapper.rdf_property(EX.curso)
    def curso(self): pass

    @rdf_mapper.rdf_property(EX.turno)
    def turno(self): pass

    @rdf_mapper.rdf_property(EX.modalidade)
    def modalidade(self): pass

      # ... (demais propriedades omitidas para brevidade) ...

    @rdf_mapper.rdf_property(EX.data_defesa)
    def data_defesa(self): pass

```

Com a classe de mapeamento definida, foi implementado um script para a transformação dos dados. Para a ingestão e manipulação do arquivo CSV, foi utilizada a biblioteca **Pandas**, a principal ferramenta do ecossistema Python para análise de dados. O Pandas

[1] Os dados foram extraídos de: <https://dadosabertos.ufma.br/dataset/trabalhos-de-conclusao-de-curso-defendidos>

permitiu carregar o dataset de forma eficiente em uma estrutura de DataFrame, que facilita a iteração sobre cada linha, o tratamento de dados ausentes e a conversão de tipos.

A partir disso, o script iterou sobre o DataFrame, instanciando um objeto `TCC` para cada registro. A coleção de objetos resultante foi então serializada em um único e coeso grafo RDF com o método `to_rdf_many` do RDFMapper. Subsequentemente, a API de consultas do `RDFRepository` foi empregada para extrair as métricas agregadas. Para a etapa de visualização, os dados extraídos foram utilizados para gerar os gráficos com a biblioteca **Matplotlib**, uma das mais consolidadas e flexíveis para a criação de gráficos estáticos e interativos em Python. Essa combinação de ferramentas demonstra um pipeline completo: ingestão e limpeza com Pandas, mapeamento semântico com RDFMapper, análise com RDFRepository e, por fim, a visualização dos insights com Matplotlib.

As principais análises e agrupamentos foram realizados por meio de chamadas como:

- `repo.group_by_count(TCC, "curso", order="DESC")`: Para identificar os cursos com maior número de defesas de TCCs (Figura 6).
- `repo.group_by_count(TCC, "modalidade")`: Para examinar a distribuição entre modalidades (Figura 7).
- `repo.group_by_count(TCC, "municipio")`: Para analisar a distribuição geográfica das defesas (Figura 8).
- `repo.group_by_count(TCC, "mes")`: Para avaliar a tendência temporal das defesas (Figura 9).
- `repo.find_by_titulo_like("EDUCAÇÃO")`: Para buscas temáticas.

A seguir, estão algumas das visualizações obtidas:

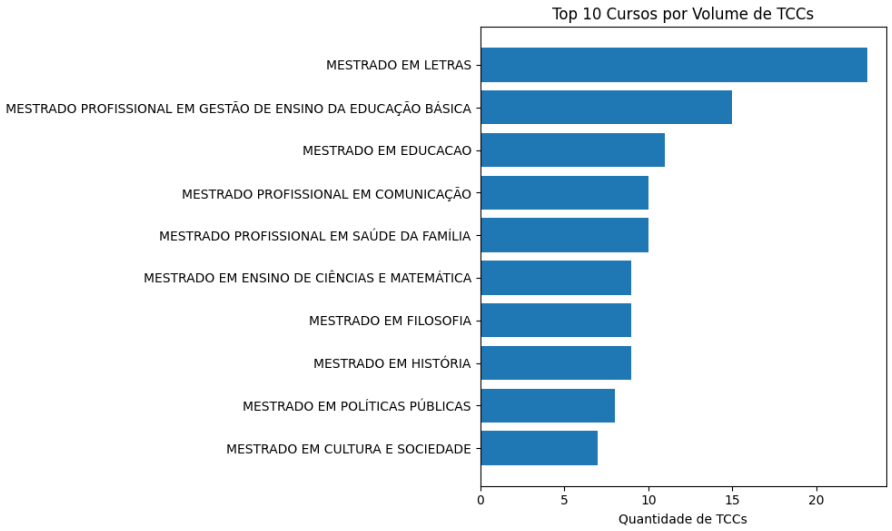


Figura 6 – Top 10 cursos por volume de TCCs.

A análise do volume de trabalhos por curso, ilustrada na Figura 6, revela a predominância de programas de pós-graduação entre os mais produtivos. Destaca-se o Mestrado em Letras, que lidera a lista com um volume superior a 20 trabalhos defendidos no período analisado.

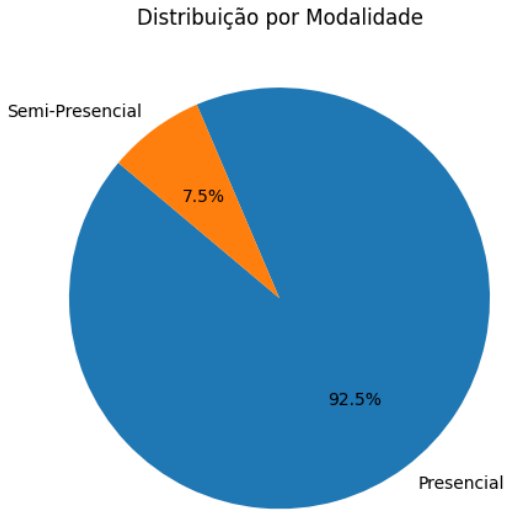


Figura 7 – Distribuição percentual por modalidade de defesa.

A Figura 7 exibe a distribuição das defesas por modalidade, onde se observa uma predominância expressiva do formato presencial. Este resultado sugere uma forte retomada dos formatos acadêmicos tradicionais. Contudo, a existência de defesas em modalidades semi-presenciais ou a distância, ainda que minoritária, indica a incorporação de práticas flexibilizadas que podem ter sido aceleradas em anos recentes, consolidando um modelo híbrido na instituição.

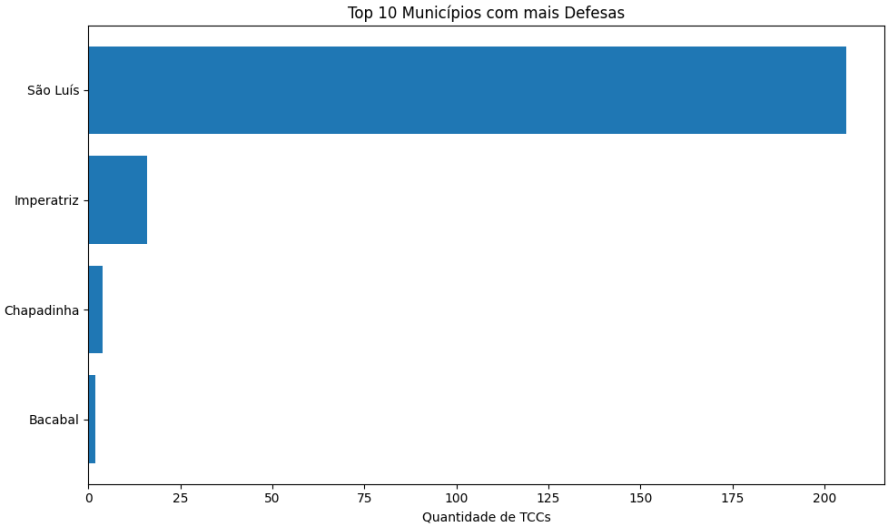


Figura 8 – Top 10 municípios por número de defesas.

A distribuição geográfica das defesas, apresentada na Figura 8, mostra uma concentração massiva na capital, São Luís, com mais de 200 trabalhos registrados. Este número é esperado, dado que o principal campus da UFMA, com a maior oferta de cursos e programas de pós-graduação, está localizado na cidade.

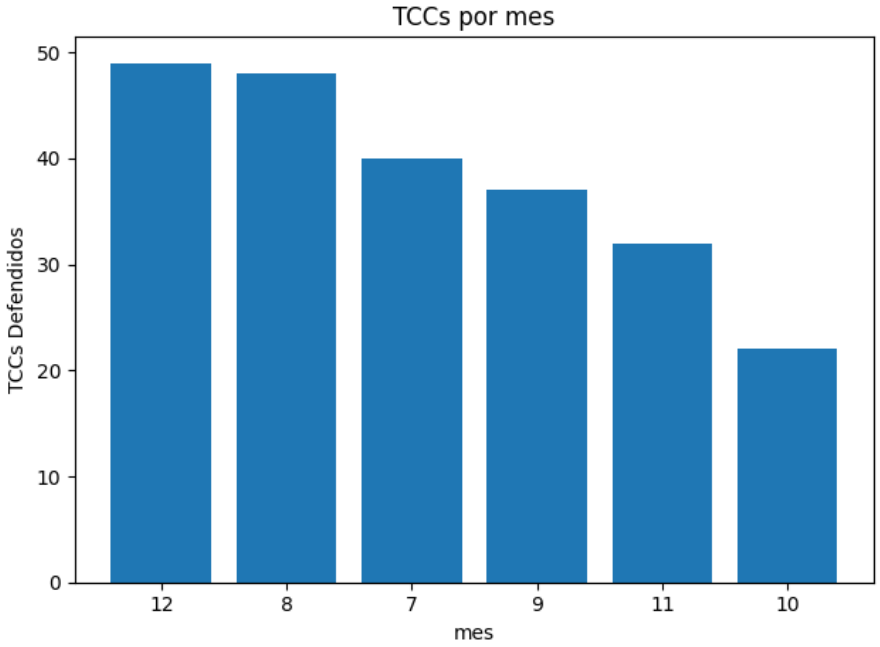


Figura 9 – Distribuição mensal de defesas de TCCs em 2024.

Ao analisar a distribuição temporal das defesas ao longo do ano de 2024 (Figura 9), identifica-se um pico de apresentações no mês de dezembro. Este padrão é característico do calendário acadêmico, onde os meses de final de semestre concentram os prazos para a conclusão dos cursos. O grande volume em dezembro reflete o esforço de alunos e

orientadores para cumprir as exigências de integralização curricular antes do encerramento do ano letivo.

Essas consultas evidenciam a facilidade de obtenção de estatísticas agregadas a partir de dados reais, sem a necessidade de escrita manual de consultas SPARQL. O uso do `RDFRepository` proporciona uma interface intuitiva e expressiva para análise exploratória e visualização dos dados, viabilizando estudos de tendências, distribuição por curso, modalidade, localidade e análise temática. O modelo objeto-RDF simplifica tanto a integração de dados quanto a extração de insights analíticos, confirmando o potencial do framework para aplicações reais em contexto acadêmico.

6.2.2 Análise dos Dados de Preço de Combustíveis

Para o segundo estudo de caso, o *RDFMapper* foi aplicado a um conjunto de dados de maior volume e complexidade: a "Série Histórica de Preços de Combustíveis e de GLP", um vasto conjunto de dados abertos disponibilizado pela Agência Nacional do Petróleo, Gás Natural e Biocombustíveis (ANP) no portal de dados do governo federal[2]. O objetivo foi validar a capacidade da ferramenta de lidar com milhões de registros e facilitar a análise exploratória sobre eles.

De forma análoga ao estudo anterior, o processo iniciou-se com a modelagem dos dados. A classe `Combustiveis`, apresentada no Código 39, foi criada para representar cada linha do dataset original. Cada coluna do arquivo CSV, como "Municipio", "Produto" e "Bandeira", foi mapeada para uma propriedade RDF correspondente por meio dos decoradores do framework.

[2] Os dados foram extraídos de: <https://dados.gov.br/dados/conjuntos-dados/serie-historica-de-precos-de-combustiveis-e-de-glp>

Código 39 Classe `Combustiveis` para mapeamento da série histórica de preços

```
@rdf_mapper.rdf_entity(EX.Combustiveis)
class Combustiveis:
    def __init__(self,  uri, Regiao___Sigla, Estado___Sigla, Municipio, Revenda,
    ↪  CNPJ_da_Revenda, Nome_da_Rua, Numero_Rua, Complemento, Bairro, Cep, Produto,
    ↪  Data_da_Coleta, Valor_de_Venda, Valor_de_Compra, Unidade_de_Medida, Bandeira):
        self.uri = uri
        self._Regiao___Sigla: str = Regiao___Sigla
        self._Estado___Sigla: str = Estado___Sigla
        self._Municipio: str = Municipio
        self._Revenda: str = Revenda
        self._CNPJ_da_Revenda: str = CNPJ_da_Revenda
        self._Nome_da_Rua: str = Nome_da_Rua
        self._Numero_Rua: str = Numero_Rua
        self._Complemento: str = Complemento
        self._Bairro: str = Bairro
        self._Cep: str = Cep
        self._Produto: str = Produto
        self._Data_da_Coleta: str = Data_da_Coleta
        self._Valor_de_Venda: str = Valor_de_Venda
        self._Valor_de_Compra: str = Valor_de_Compra
        self._Unidade_de_Medida: str = Unidade_de_Medida
        self._Bandeira: str = Bandeira

    @rdf_mapper.rdf_property(EX.Regiao___Sigla)
    def Regiao___Sigla(self):
        return self._Regiao___Sigla

    @rdf_mapper.rdf_property(EX.Estado___Sigla)
    def Estado___Sigla(self):
        return self._Estado___Sigla

    @rdf_mapper.rdf_property(EX.Municipio)
    def Municipio(self):
        return self._Municipio

    # ... (demais propriedades omitidas para brevidade) ...

    @rdf_mapper.rdf_property(EX.Bandeira)
    def Bandeira(self):
        return self._Bandeira
```

Após a definição do modelo, o conjunto de dados foi processado, convertendo cada linha do CSV em uma instância da classe `Combustiveis` e, em seguida, serializando a coleção de objetos para um grafo RDF. O tratamento dos dados seguiu a mesma abordagem integrada anteriormente, potencializando a análise exploratória ao aliar ferramentas de processamento tabular e visualização gráfica à consulta semântica. A exploração e análise do dataset foram realizadas a partir das operações nativas do `RDFRepository`, permitindo agrupar, filtrar e consultar informações relevantes de maneira eficiente. Os principais resultados extraídos incluem:

- **Distribuição das bandeiras de postos:** Utilizando o método `group_by_count(Combustiveis, "Bandeira")`, , foi possível identificar as distribuidoras líderes de mercado (Figura 10).
- **Estados e municípios mais monitorados:** As funções `group_by_count(Combustiveis,`

`"Estado__Sigla")` e `group_by_count(Combustiveis, "Municipio")` permitiram identificar as regiões com maior quantidade de registros (Figuras 11 e 12).

- **Perfil dos produtos comercializados:** O agrupamento por `Produto` evidenciou a predominância de gasolina, etanol e diesel entre os registros coletados (Figura 13).

Esses resultados foram obtidos a partir de simples consultas de agregação e filtragem, exemplificadas a seguir:

Código 40 Consultas agregadas com RDFRepository para o dataset de combustíveis

```
# Top 10 estados
estados = repo.group_by_count(Combustiveis, "Estado___Sigla", order="DESC")[:10]
# Top 10 municípios
municipios = repo.group_by_count(Combustiveis, "Municipio", order="DESC")[:10]
# Principais bandeiras (após aglutinação das menores)
bandeiras = repo.group_by_count(Combustiveis, "Bandeira", order="DESC")
# Tipos de produto comercializados
produtos = repo.group_by_count(Combustiveis, "Produto", order="DESC")
```

A seguir, apresentamos as visualizações derivadas dessas consultas:

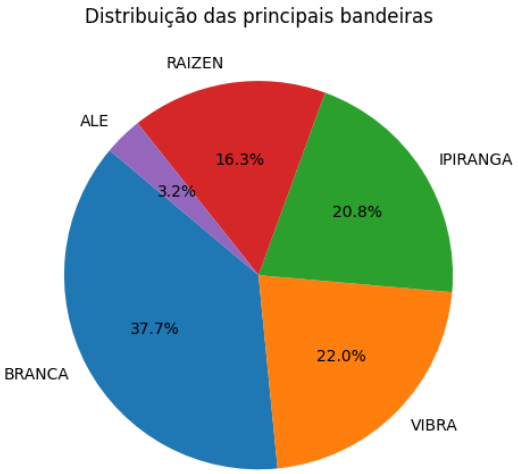


Figura 10 – Distribuição percentual das principais bandeiras de postos.

A Figura 10 ilustra a distribuição de mercado entre as principais bandeiras de postos de combustíveis. A análise evidencia um mercado concentrado em grandes distribuidoras como Vibra (antiga BR Distribuidora), Ipiranga e Raízen (Shell), que detêm uma parcela significativa das vendas. Contudo, um dos destaques mais relevantes é a forte presença da "bandeira branca", que representa os postos independentes. A expressiva fatia de mercado desses postos sugere um ambiente competitivo e pulverizado, onde players menores coexistem com as grandes redes.

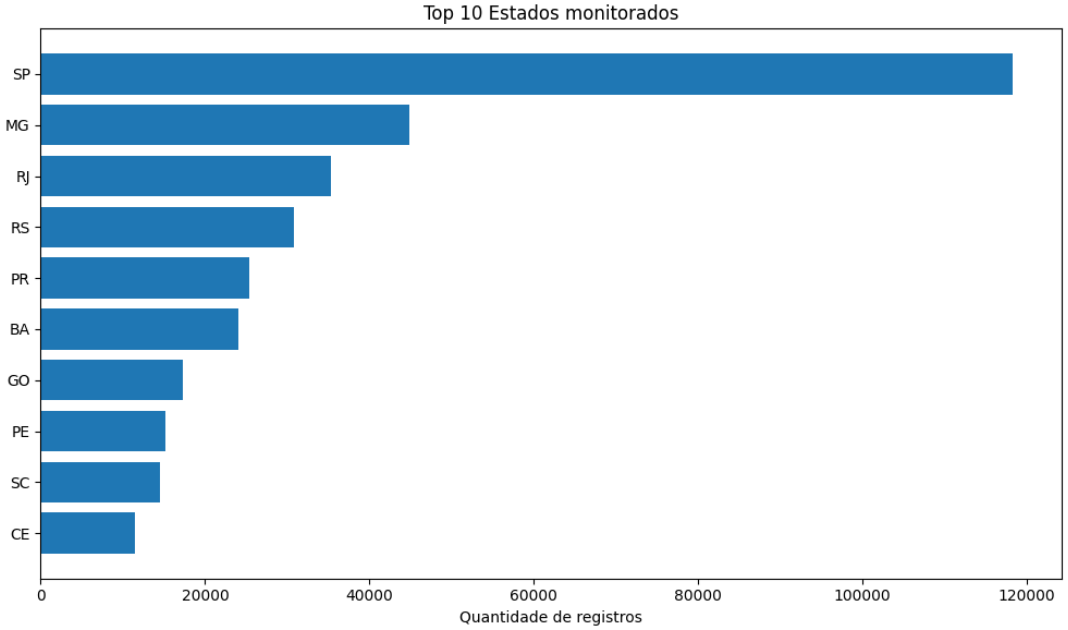


Figura 11 – Top 10 estados brasileiros com maior volume de registros.

A análise do volume de registros por estado, conforme a Figura 11, evidencia que São Paulo lidera com ampla margem, seguido por Minas Gerais e Rio de Janeiro. Este resultado é coerente com a realidade demográfica e econômica do Brasil, uma vez que estes são os três estados mais populosos e com as maiores frotas de veículos do país. O volume de registros de preços coletados, portanto, serve como um reflexo direto da atividade econômica e da demanda por combustíveis nessas regiões.

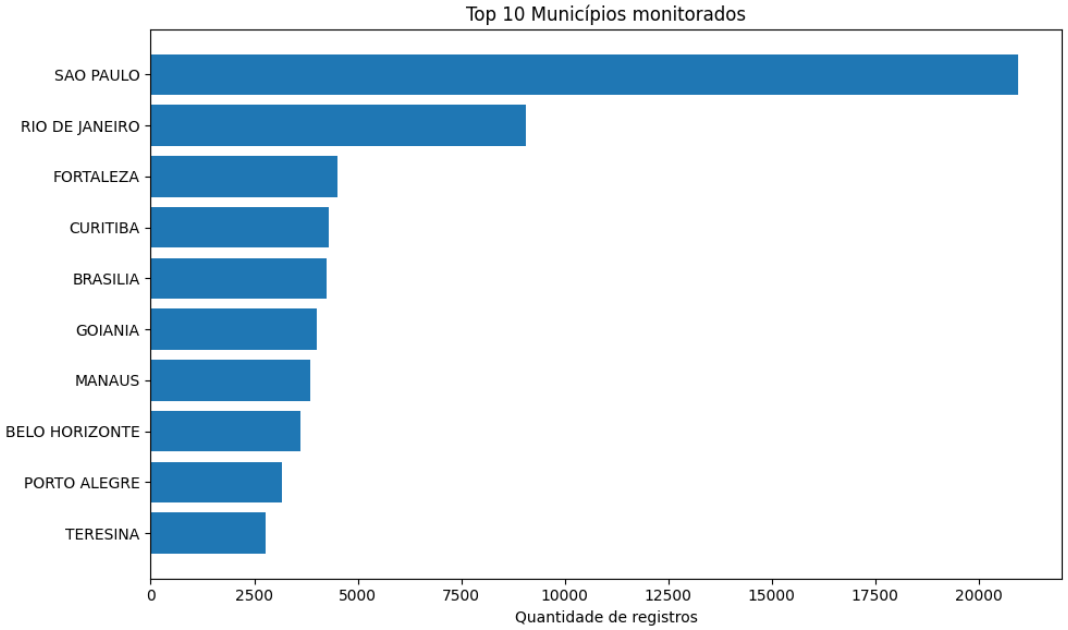


Figura 12 – Top 10 municípios monitorados no dataset de combustíveis.

De forma análoga à análise estadual, a Figura 12 demonstra que a capital São Paulo

apresenta o maior volume de registros de vendas, seguida por outras grandes metrópoles como Rio de Janeiro e Fortaleza. A predominância de capitais e grandes centros urbanos no ranking é esperada, refletindo a maior concentração populacional, de veículos e de postos de combustíveis nessas áreas. A análise em nível municipal permite identificar os principais mercados consumidores do país.

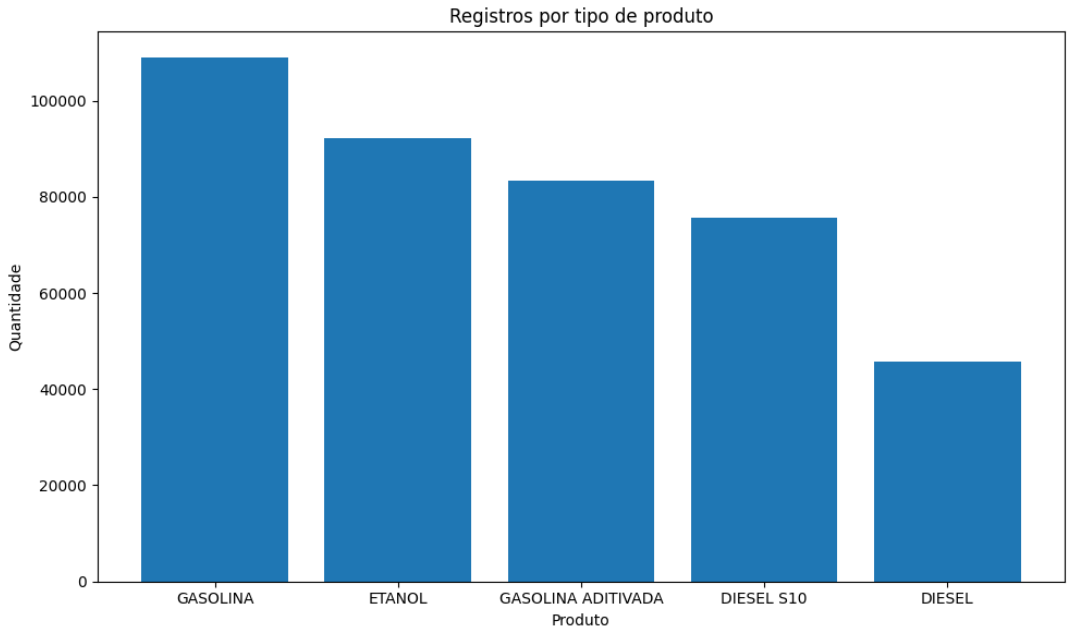


Figura 13 – Registros por tipo de produto comercializado (ex: gasolina, etanol, diesel).

A distribuição por tipo de produto, visível na Figura 13, destaca a gasolina como o combustível com maior número de registros de venda, seguida de perto pelo etanol. Este padrão espelha a composição da frota nacional de veículos leves, majoritariamente composta por automóveis flex-fuel, que consomem tanto gasolina quanto etanol. A forte presença de ambos no mercado evidencia essa característica estrutural do setor automotivo brasileiro.

Dessa forma, o uso conjunto do *RDFMapper* e do *RDFRepository* permitiu não só a serialização semântica de grandes volumes de dados, mas também uma análise exploratória eficiente, sem exigir conhecimento prévio em SPARQL ou manipulação manual de triplas. As visualizações derivadas das operações agregadas evidenciam o potencial da abordagem para integração, análise e interoperabilidade de dados abertos.

6.3 Análise de Desempenho Experimental

Para quantificar as diferenças entre as abordagens, foram conduzidos experimentos de desempenho medindo o tempo de execução e o pico de uso de memória para operações

de serialização em massa e de consulta. Os testes foram realizados com volumes de dados crescentes, de 1.000 a 100.000 instâncias de entidades.

Os resultados apresentados nas Tabelas 3 e 4, bem como nos Gráficos 14 e 15, comparam o desempenho do processo de serialização em massa utilizando o RDFMapper (`to_rdf`) e a abordagem tradicional com o `rdflib` puro (chamadas diretas a `g.add()`).

Tabela 3 – Tempo de serialização (em segundos) por volume de entidades.

Volume	**Tempo (`rdf_mapper`) [s]**	**Tempo (`rdflib`) [s]**
1.000	0,44	0,11
10.000	4,44	1,20
50.000	22,79	6,58

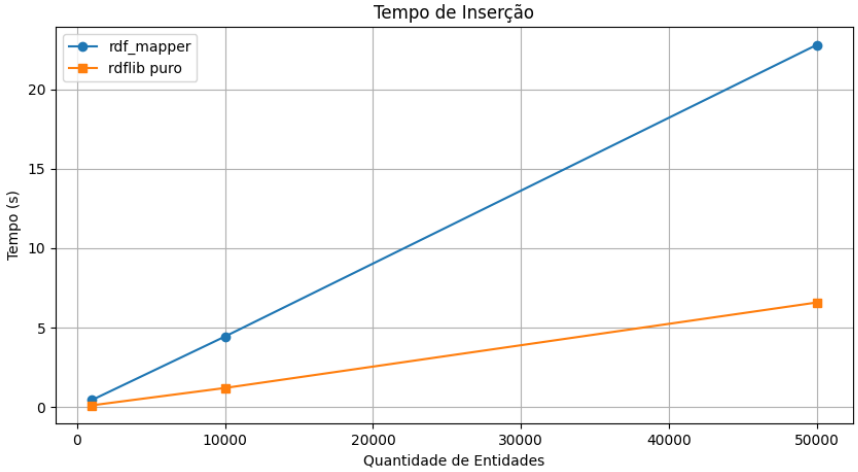


Figura 14 – Gráfico do tempo de execução para serialização em massa.

O tempo necessário para converter um grande volume de objetos Python em um grafo RDF cresce de forma aproximadamente linear em ambas as abordagens, à medida que o número de entidades aumenta. No entanto, observa-se que o `rdflib` puro é mais rápido em todos os cenários avaliados, exigindo menos da metade do tempo do `RDFMapper` ao processar 50.000 entidades (Tabela 3). Este resultado se deve ao fato do `rdflib` operar diretamente sobre estruturas otimizadas de triplas, sem o custo adicional da introspecção de metadados e da conversão orientada a objetos presente no `RDFMapper`.

Tabela 4 – Consumo de memória de pico (em MB) durante a serialização.

Volume	**Memória (`rdf_mapper`) [MB]**	**Memória (`rdflib`) [MB]**
1.000	1,88	4,10
10.000	13,33	41,85
50.000	62,74	206,12

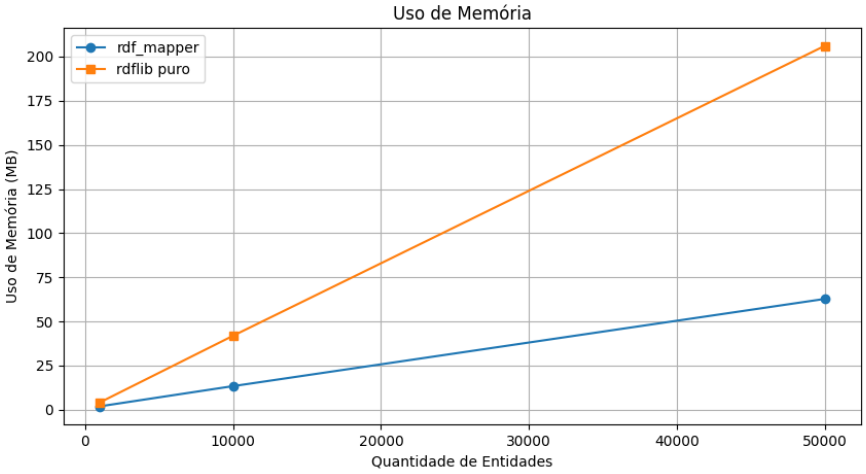


Figura 15 – Gráfico do consumo de memória durante a serialização em massa.

Os dados de consumo de memória indicam que o `RDFMapper` é mais eficiente neste quesito. Mesmo manipulando grandes volumes de entidades, seu uso de memória permanece substancialmente menor do que o `rdflib` puro (Tabela 4). Para 50.000 registros, o `rdflib` consome mais de 200 MB, enquanto o `RDFMapper` permanece abaixo de 65 MB. Isso sugere que a representação inicial dos dados como objetos Python, antes da serialização final, é mais compacta do que as estruturas intermediárias internas utilizadas pelo `rdflib`.

Para cenários onde o gargalo principal é o tempo de serialização, o `rdflib` puro apresenta vantagens claras, sendo recomendado para pipelines ETL, ingestão em larga escala ou exportações massivas de dados RDF. Já o `RDFMapper` demonstra melhor uso de memória, sendo vantajoso em ambientes restritos ou aplicações embarcadas, além de agregar valor significativo na manutenção do código, reutilização de modelos e integração OO.

Os resultados mostram que o `rdflib` puro é a escolha ideal quando o principal requisito é a velocidade na serialização de grandes volumes de dados, ao passo que o `RDFMapper` se destaca pelo baixo consumo de memória e maior organização do código. Assim, em ambientes onde a limitação de recursos é fator crítico ou a modelagem orientada a objetos traz ganhos em produtividade e manutenção, o uso do `RDFMapper` pode ser mais vantajoso, mesmo com um leve aumento no tempo de execução.

6.3.1 Desempenho da Serialização em Massa

Os testes de serialização mediram o desempenho ao converter um grande volume de objetos Python em um grafo RDF.

6.3.2 Análise dos Resultados – RDFRepository vs. `rdflib`

Os resultados apresentados nas Tabelas 5 e 6 e nos Gráficos 16 e 17 comparam o desempenho da API de consultas dinâmica do RDFRepository (implementado sobre o `rdf_mapper`) com o uso de consultas diretas via `rdflib` (SPARQL puro) sobre grandes volumes de entidades RDF.

Tabela 5 – Tempo de consulta (em segundos) por volume de entidades.

Volume	**Tempo (`rdf_mapper`) [s]**	**Tempo (`rdflib`) [s]**
1.000	0,046	0,055
10.000	0,382	0,003
50.000	1,924	0,003
100.000	3,840	0,003

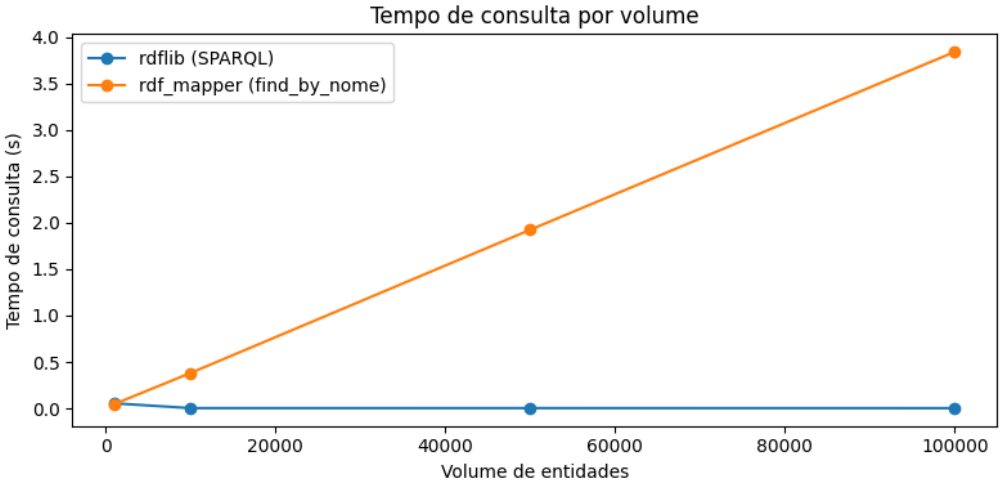


Figura 16 – Gráfico do tempo de consulta por volume de entidades.

O RDFRepository utiliza uma abordagem dinâmica, inspirada em ORMs como JPA, permitindo chamadas como `find_by_nome()` ou `count_by_nome()`. Embora traga grande produtividade e flexibilidade para o desenvolvedor, seu mecanismo depende de iteração sobre objetos Python desserializados a partir do grafo RDF. Como consequência, o tempo de consulta no RDFRepository cresce linearmente com o volume de dados, atingindo quase 4 segundos para 100.000 entidades. Por outro lado, as consultas SPARQL puras com o `rdflib` mantêm performance praticamente constante, graças à engine de busca otimizada para grafos RDF.

Tabela 6 – Uso de memória (em MB) durante a consulta.

Volume	**Memória (`rdf_mapper`) [MB]**	**Memória (`rdflib`) [MB]**
1.000	6,88	4,25
10.000	56,27	41,45
50.000	272,23	198,16
100.000	416,68	329,23

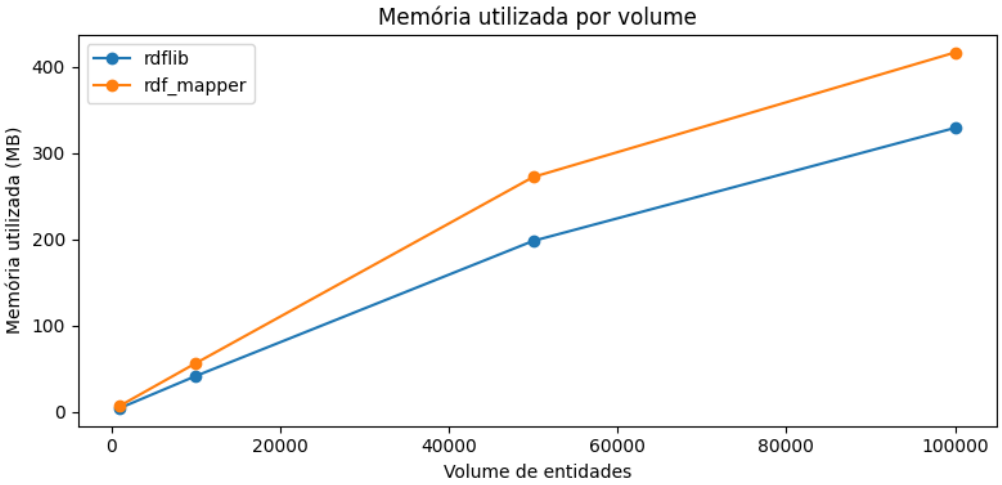


Figura 17 – Gráfico da memória utilizada durante a consulta.

O consumo de memória do RDFRepository também cresce mais rapidamente, devido ao carregamento dos dados em estruturas de objetos Python. O `rdflib` se mostra mais eficiente, conseguindo escalar melhor para grandes volumes de dados, ainda que também apresente crescimento gradual de memória.

O RDFRepository é altamente indicado para cenários em que a produtividade, legibilidade do código e integração com o paradigma orientado a objetos são mais importantes do que o desempenho bruto. Para cenários de grande escala, que exigem consultas rápidas sobre milhões de registros, o uso direto do `rdflib` e de SPARQL puro é mais apropriado.

A comparação evidencia que o `rdflib` puro é significativamente superior em desempenho de consultas, graças à sua engine SPARQL otimizada. O RDFMapper, por sua vez, oferece flexibilidade e facilidade de uso para cenários orientados a objetos, mas apresenta tempo de resposta maior e maior uso de memória à medida que o volume de entidades cresce. Portanto, para aplicações com alto volume de consultas e requisitos de latência, o `rdflib` tradicional permanece como referência, enquanto o RDFMapper é mais indicado para prototipagem rápida e integração com sistemas Python orientados a objetos.

7 Discussão

A análise dos resultados apresentados no capítulo anterior permite uma discussão aprofundada sobre as vantagens, os compromissos e as implicações da abordagem proposta pelo `RDFMapper` em comparação com a manipulação direta de RDF.

7.1 Análise Qualitativa da Produtividade

É importante destacar que, na abordagem com `rdf_mapper`, a definição das classes e seus respectivos mapeamentos RDF são feitas uma única vez e podem ser reutilizadas. Esse fator é especialmente relevante em cenários reais com grande volume de dados, como no caso de dados abertos de combustíveis, onde é comum lidar com milhões de registros. A reutilização da estrutura proporciona não apenas uma expressiva economia de código, mas também maior organização e clareza.

Por outro lado, ao utilizar apenas a biblioteca `rdflib`, não existe uma camada de abstração orientada a objetos. Assim, cada instância de objeto precisa ser manualmente serializada com múltiplas chamadas explícitas a `g.add(...)`. Isso implica em maior verbosidade, duplicação de lógica e maior propensão a erros, além de dificultar a manutenção e escalabilidade do código.

Os benefícios qualitativos podem ser resumidos nos seguintes pontos:

- **Reusabilidade**: A definição das classes pode ser reaproveitada em diferentes contextos, o que não é viável no código com *rdflib* puro.
- **Semântica mais clara**: A semântica dos dados RDF é incorporada diretamente no modelo de domínio por meio de decoradores, facilitando a manutenção e a leitura do código.
- **Ausência de manipulação manual de triplas**: O *RDFMapper* abstrai completamente a criação de triplas RDF, o que elimina a necessidade de chamadas manuais à função `add()` e reduz significativamente a chance de erros.
- **Verbosidade reduzida**: A abordagem com *RDFMapper* exige menos linhas de código e apresenta menor repetição de estruturas, tornando o desenvolvimento mais ágil e limpo.
- **Escalabilidade**: Em cenários de grande volume de dados, a adoção do *RDFMapper* torna viável a manipulação de milhões de entidades com regras semânticas consistentes, tarefa que se torna impraticável com a manipulação manual.

7.2 Análise dos Resultados de Desempenho

A análise conjunta dos resultados quantitativos revela um claro **trade-off entre produtividade do desenvolvedor e desempenho computacional bruto**.

7.2.1 Análise do Custo de Abstração

Na **serialização**, o `rdflib` puro é consistentemente mais rápido (até 3,5x). Isso ocorre porque ele executa operações de baixo nível, adicionando triplas diretamente a uma estrutura de dados otimizada. Em contrapartida, o `rdf_mapper` incorre em um *overhead* inevitável devido à **introspecção** — o processo de analisar dinamicamente os decoradores e a estrutura das classes Python para traduzi-los em triplas RDF. Este é o "custo" da abstração.

Notavelmente, o `rdf_mapper` demonstrou ser muito mais **eficiente em memória** (usando até 3,3x menos memória). Uma possível justificativa é que a representação dos dados como objetos Python nativos, antes da serialização final, é mais compacta do que as estruturas internas que o `rdflib` utiliza para armazenar o grafo em memória durante sua construção.

7.2.2 Impacto na Performance de Consulta

A diferença de desempenho é ainda mais acentuada nas **consultas**. O método do `rdf_mapper` ('find_by_') exibe um crescimento de tempo linear (complexidade O(n)), pois sua estratégia consiste em uma **iteração em memória** sobre a coleção de objetos Python. Em contraste, a consulta SPARQL via `rdflib` puro tem um desempenho quase constante (O(1) ou O(log n) para este caso), pois utiliza um **motor de consulta otimizado com estruturas de dados indexadas**, que localiza as triplas de forma muito mais eficiente.

7.2.3 Síntese e Recomendações

A escolha entre as duas abordagens é guiada pelos requisitos do projeto:

- **Escolha `RDFMapper` quando:** O foco é a **produtividade**, **clareza do código** e **manutenibilidade**. É ideal para aplicações web, APIs e sistemas orientados a objetos onde a lógica de negócio é complexa e se beneficia da expressividade das classes. A economia de memória na serialização também o torna atraente para ambientes com recursos limitados.
- **Escolha `RDFLib` puro quando:** O requisito primário é o **máximo desempenho** em processamento de dados. É a escolha superior para pipelines de ETL (Extração,

Transformação e Carga), tarefas de análise de dados em larga escala e cenários que dependem de consultas SPARQL complexas e otimizadas.

Em suma, o `rdf_mapper` paga um custo de desempenho computacional para oferecer um ganho significativo em desempenho humano e qualidade de software, uma troca vantajosa em muitos cenários de desenvolvimento de software modernos.

8 Conclusão

Este trabalho partiu da identificação de uma lacuna fundamental entre o poder do modelo de dados RDF, pilar da Web Semântica, e a praticidade do paradigma de orientação a objetos, dominante no desenvolvimento de software. A manipulação de grafos RDF por meio de bibliotecas de baixo nível exige um esforço considerável e conhecimento especializado, dificultando a adoção de tecnologias semânticas por uma comunidade mais ampla de desenvolvedores. Para endereçar este desafio, propôs-se, desenvolveu-se e avaliou-se o **RDFMapper**, uma biblioteca de Mapeamento Objeto-RDF para Python.

Os resultados obtidos confirmam que o RDFMapper atinge seu objetivo central. A análise qualitativa demonstrou uma redução drástica na verbosidade e na complexidade para realizar operações de CRUD. A aplicação da biblioteca em conjuntos de dados reais e complexos — dados de TCCs da UFMA e de preços de combustíveis da ANP — comprovou sua viabilidade prática, com a API do `RDFRepository` facilitando a extração de *insights* de forma ágil. Além disso, a integração de um mecanismo de validação com SHACL, capaz de gerar "shapes"a partir dos modelos de classe, representa um avanço significativo na garantia da integridade e conformidade dos dados gerados.

A análise de desempenho experimental revelou o esperado *trade-off* entre abstração e performance. Enquanto a manipulação direta com `RDFLib` se mostrou mais rápida, o RDFMapper apresentou uma notável vantagem de eficiência no consumo de memória durante a serialização. Essa troca de desempenho computacional por um ganho expressivo em produtividade, legibilidade e robustez dos dados é a principal proposta de valor da ferramenta.

Contribuições

As principais contribuições deste trabalho são:

- **O desenvolvimento do framework RDFMapper**, uma nova ferramenta de código aberto para o ecossistema Python que oferece uma camada de abstração de alto nível para manipulação de dados RDF.
- **A validação de uma abordagem inspirada em ORM para dados semânticos**, demonstrando que os princípios de mapeamento declarativo e repositórios dinâmicos são eficazes para o modelo de grafos.
- **A integração de um mecanismo de validação baseado em SHACL**, que permite a geração automática de restrições a partir do modelo de classes, fortalecendo a integridade dos dados de forma declarativa.

- **A redução da barreira de entrada para a Web Semântica**, fornecendo a desenvolvedores Python uma ferramenta que lhes permite aproveitar o poder dos Dados Conectados de forma mais produtiva e segura.

Do ponto de vista do desenvolvimento de software, destaca-se que o projeto do *RDFMapper* foi fortemente inspirado em frameworks consolidados, como o Spring Data JPA. Contudo, optou-se deliberadamente por uma implementação mais enxuta e focada, priorizando a simplicidade de uso e a curva de aprendizado reduzida. Em vez de abarcar a ampla gama de funcionalidades presentes em soluções robustas do ecossistema Java, buscou-se desenvolver um conjunto de recursos essenciais capazes de atender às demandas mais comuns do mapeamento Objeto-RDF em Python.

Essa abordagem deliberadamente minimalista visa tornar a biblioteca mais acessível e menos propensa a complexidades desnecessárias, facilitando o processo de adoção, aprendizado e manutenção por parte dos desenvolvedores. O resultado é uma solução que, embora ofereça um leque de recursos mais restrito do que grandes frameworks do mercado, proporciona ganhos expressivos em produtividade, clareza de código e aderência ao paradigma orientado a objetos. Dessa forma, o *RDFMapper* contribui não apenas para a redução do esforço de desenvolvimento, mas também para a promoção de boas práticas de modelagem e integração de dados semânticos em aplicações Python.

Trabalhos Futuros

Apesar dos resultados positivos, este trabalho abre diversas avenidas para futuras evoluções da ferramenta. As seguintes direções são propostas:

- **Otimização de Desempenho:** Investigar técnicas de otimização para a camada de consulta, como a geração de consultas SPARQL mais complexas de forma automática ou a implementação de mecanismos de cache.
- **Integração com Triplestores Externos:** Estender o `RDFRepository` para se comunicar diretamente com *endpoints* SPARQL de triplestores como Virtuoso ou GraphDB, permitindo operar sobre grafos de grande escala.
- **Aprimoramento do Suporte a SHACL:** Expandir a geração automática de "shapes"para abranger restrições mais complexas, especialmente aquelas envolvendo relacionamentos entre entidades, e não apenas atributos literais.
- **Expansão do Suporte a Ontologias:** Aprimorar o mapeamento para suportar construções mais avançadas de RDFS e OWL, como inferência de subclasses e propriedades transitivas.

Em suma, conclui-se que o RDFMapper representa um passo significativo para tornar as tecnologias da Web Semântica mais pragmáticas e acessíveis. Ao abstrair complexidades e promover uma interface de desenvolvimento familiar e robusta, ferramentas como esta são essenciais para catalisar a criação de uma nova geração de aplicações inteligentes e verdadeiramente conectadas.

Referências

AMBLER, S. W. Mapping objects to relational databases. *Journal of Object Technology*, v. 2, n. 5, p. 53–62, 2003. Disponível em: <http://www.jot.fm/issues/issue_2003_07/article2.pdf>. Citado na página 16.

ANCONA, D.; LAGORIO, G.; ZUCCA, E. Towards a formal semantics for python metaclasses. In: ACM. *Proceedings of the 2007 ACM symposium on Applied computing*. [S.l.], 2007. p. 1239–1243. Citado na página 22.

ANGLES, R.; GUTIERREZ, C. Survey of graph database models. *ACM Computing Surveys (CSUR)*, ACM, v. 40, n. 1, p. 1–39, 2008. Citado 2 vezes nas páginas 19 e 20.

ANTONIOU, G.; HARMELEN, F. van. *A Semantic Web Primer*. 2. ed. [S.l.]: MIT Press, 2011. ISBN 9780262018289. Citado na página 8.

AUER, S.; BIZER, C.; KOBILAROV, G.; LEHMANN, J.; CYGANIAK, R.; IVES, Z. Dbpedia: A nucleus for a web of open data. Springer, p. 722–735, 2007. Citado na página 13.

BANDEIRA, J. M.; ALCANTARA, W.; SOBRINHO, A. B.; ÁVILA, T. J. T.; BITTENCOURT, I.; ISOTANI, S. Dados abertos conectados. *III Simpósio Brasileiro de Tecnologia da Informação*, 2015. Citado na página 3.

BAUER, C.; KING, G. *Java Persistence with Hibernate*. [S.l.]: Manning Publications, 2015. Citado 2 vezes nas páginas 16 e 22.

BERNERS-LEE, T. *Linked data*. 2006. Disponível em: <https://www.w3.org/DesignIssues/LinkedData.html>. Citado 2 vezes nas páginas 1 e 4.

BERNERS-LEE, T.; FIELDING, R.; MASINTER, L. *Uniform Resource Identifier (URI): Generic Syntax*. 2005. <https://www.rfc-editor.org/rfc/rfc3986>. RFC 3986. Citado na página 4.

BERNSTEIN, P. A.; NEWCOMER, E. *Principles of Transaction Processing*. 2. ed. [S.l.]: Morgan Kaufmann, 2008. ISBN 978-1558606234. Citado 2 vezes nas páginas 16 e 18.

BISCHOF, S.; DECKER, B.; KURZ, T.; SCHANDL, B. Empire: A jpa-based rdf object-mapper. In: *The Semantic Web: Research and Applications*. [S.l.]: Springer Berlin Heidelberg, 2011. (Lecture Notes in Computer Science, v. 6643), p. 336–350. Citado na página 16.

BREITMAN, K. K.; CASANOVA, M. A.; TRUSZKOWSKI, W. *Semantic Web: Concepts, Technologies and Applications*. [S.l.]: Springer, 2010. Citado na página 5.

BRICKLEY, D.; GUHA, R. V. *RDF Schema 1.1*. 2014. <https://www.w3.org/TR/rdf-schema/>. World Wide Web Consortium (W3C) Recommendation. Citado na página 8.

DAVIES, J.; BIRON, P. The bbc and linked data. *Semantic Web Journal*, IOS Press, v. 3, n. 1, p. 31–36, 2012. Citado na página 15.

DODDS, L. *RDFAlchemy: An ORM for RDF in Python*. 2012. Acessado em julho de 2025. Disponível em: <https://github.com/rdfalchemy/rdfalchemy>. Citado na página 21.

Eclipse Foundation. *Jakarta Persistence 3.0*. [S.l.], 2020. Disponível em: <https://jakarta.ee/specifications/persistence/3.0/>. Citado 2 vezes nas páginas 17 e 18.

ERLING, O.; MIKHAILOV, I. Virtuoso, a hybrid rdbms/graph column store. v. 35, n. 1, p. 3–8, 2012. Citado na página 12.

FOUNDATION, A. S. *Apache Jena Fuseki*. 2024. Disponível em: <https://jena.apache.org/documentation/fuseki2/>. Citado na página 12.

FOUNDATION, W. *Wikidata Query Service*. 2024. Online; acesso em julho de 2025. <https://query.wikidata.org/>. Citado na página 13.

FOWLER, M. *Patterns of Enterprise Application Architecture*. Boston: Addison-Wesley, 2002. ISBN 978-0321127426. Citado 3 vezes nas páginas 1, 15 e 16.

FOWLER, P. P. *Metaprogramming Ruby: Program Like the Ruby Pros*. [S.l.]: Pragmatic Bookshelf, 2010. Citado na página 22.

GEONAMES. 2025. <https://www.geonames.org/>. Accessed: 2025-07-19. Citado na página 14.

GRINBERG, M. *Flask Web Development: Developing Web Applications with Python*. [S.l.]: O'Reilly Media, 2018. Citado na página 20.

GROUP, W. O. W. *OWL 2 Web Ontology Language Document Overview*. 2012. <https://www.w3.org/TR/owl2-overview/>. Citado na página 9.

HEATH, T.; BIZER, C. *Linked Data: Evolving the Web into a Global Data Space*. [S.l.]: Morgan & Claypool Publishers, 2011. (Synthesis Lectures on the Semantic Web: Theory and Technology). Citado 2 vezes nas páginas 16 e 19.

HITZLER, P.; KRÖTZSCH, M.; RUDOLPH, S. *Foundations of Semantic Web Technologies*. [S.l.]: CRC Press, 2021. Citado 4 vezes nas páginas 1, 7, 15 e 20.

INC., F. *AllegroGraph Semantic Graph Platform*. 2024. Disponível em: <https://allegrograph.com/>. Citado na página 12.

ISOTANI, S.; BITTENCOURT, I. I. *Dados abertos conectados: em busca da web do conhecimento*. [S.l.]: Novatec Editora, 2015. Citado 4 vezes nas páginas 6, 3, 4 e 7.

KEITH, M.; SCHINCARIOL, M. *Pro JPA 2: Mastering the Java Persistence API*. Berkeley, CA, USA: Apress, 2009. Citado 2 vezes nas páginas 17 e 18.

LOD Cloud Diagram. 2025. <https://lod-cloud.net/>. Accessed: 2025-07-19. Citado na página 14.

LUTZ, M. *Learning Python*. 5. ed. [S.l.]: O'Reilly Media, Inc., 2013. Citado na página 23.

Microsoft Docs. *Entity Framework Documentation*. 2024. Acessado em julho de 2025. Disponível em: <https://learn.microsoft.com/en-us/ef/>. Citado na página 20.

MUSICBRAINZ. 2025. <https://musicbrainz.org/>. Accessed: 2025-07-19. Citado na página 14.

Ontotext. *Ontotext GraphDB Documentation*. 2023. <https://graphdb.ontotext.com/documentation/10.4/>. Último acesso: 05/12/2023. Citado na página 12.

RUBINGER, A. L.; BURKE, B. *Enterprise JavaBeans 3.1*. 6th. ed. [S.l.]: O'Reilly Media, 2010. Citado na página 19.

SEGUNDO, J. E. S. Web semântica, dados ligados e dados abertos: uma visão dos desafios do brasil frente às iniciativas internacionais. *Tendências da Pesquisa Brasileira em Ciência da Informação*, v. 8, n. 2, 2015. Citado na página 5.

SERVICES, A. W. *Amazon Neptune Documentation*. 2024. Disponível em: <https://docs.aws.amazon.com/neptune/>. Citado na página 13.

SOMMER, A. *pySHACL: SHACL validator for RDF data*. 2024. <https://github.com/RDFLib/pySHACL>. Acesso em: jul. 2025. Citado na página 12.

SYSTAP. *Blazegraph*. 2024. Disponível em: <https://github.com/blazegraph/database>. Citado na página 12.

UNION, S. *Stardog Knowledge Graph Platform*. 2024. Disponível em: <https://www.stardog.com/>. Citado na página 12.

VRANDEčIć, D.; KRöTZSCH, M. Wikidata: A free collaborative knowledgebase. *Communications of the ACM*, ACM, v. 57, n. 10, p. 78–85, 2014. Citado na página 13.

W3C RDF Data Shapes Working Group. *Shapes Constraint Language (SHACL)*. 2017. <https://www.w3.org/TR/shacl/>. Acesso em: jul. 2025. Citado 2 vezes nas páginas 11 e 22.

W3C RDF Working Group. *RDF 1.1 Concepts and Abstract Syntax*. 2014. <https://www.w3.org/TR/rdf11-concepts/>. Acesso em: jul. 2025. Citado na página 6.

W3C SPARQL Working Group. *SPARQL 1.1 Query Language*. 2013. <https://www.w3.org/TR/sparql11-query/>. W3C Recommendation. Citado na página 10.

WOOD, D.; LANTHALER, M.; GÜTL, C.; PFUNDNER, J.; GESER, G. Linked data: A survey of research techniques. *Journal of Web Semantics*, Elsevier, v. 27, p. 14–34, 2014. Citado na página 19.

WYLOT, M.; HAUSWIRTH, M.; CUDRÉ-MAUROUX, P.; SAKR, S. Rdf data storage and query processing schemes: A survey. *ACM Computing Surveys (CSUR)*, ACM New York, NY, USA, v. 51, n. 4, p. 1–36, 2018. Citado na página 8.