UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

JOSÉ VITOR DA CUNHA HISSE ALVES CABRAL

Driver para Visualização de Dados RDF via SPARQL no Metabase

RIO DE JANEIRO
2025

JOSÉ VITOR DA CUNHA HISSE ALVES CABRAL

Driver para Visualização de Dados RDF via SPARQL no Metabase

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Orientadora: Profa. Giseli Rabello Lopes
Coorientador: João Pedro Valladão Pinheiro

RIO DE JANEIRO
2025

CIP - Catalogação na Publicação

C117d

Cabral, José Vitor da Cunha Hisse Alves
Driver para Visualização de Dados RDF via SPARQL no Metabase / José Vitor da Cunha Hisse Alves Cabral. -- Rio de Janeiro, 2025.
116 f.

Orientadora: Giseli Rabello Lopes.
Coorientador: João Pedro Valladão Pinheiro.
Trabalho de conclusão de curso (graduação) - Universidade Federal do Rio de Janeiro, Instituto de Computação, Bacharel em Ciência da Computação, 2025.

1. Web Semântica. 2. Dados Conectados. 3. Business Intelligence. 4. SPARQL. 5. RDF. I. Lopes, Giseli Rabello, orient. II. Pinheiro, João Pedro Valladão, coorient. III. Título.

Elaborado pelo Sistema de Geração Automática da UFRJ com os dados fornecidos pelo(a) autor(a), sob a responsabilidade de Miguel Romeu Amorim Neto - CRB-7/6283.

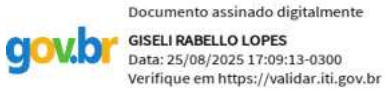
JOSÉ VITOR DA CUNHA HISSE ALVES CABRAL

Driver para Visualização de Dados RDF via SPARQL no Metabase

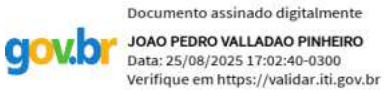
Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 22 de agosto de 2025

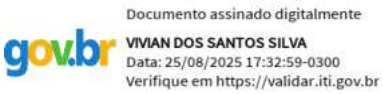
BANCA EXAMINADORA:



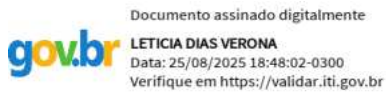
Giseli Rabello Lopes, D.Sc. (UFRJ)



João Pedro Valladão Pinheiro, M.Sc.
(PUC-Rio)



Vivian dos Santos Silva, Ph.D. (UFRJ)



Letícia Dias Verona, M.Sc. (UFRJ)

AGRADECIMENTOS

Em primeiro lugar, agradeço aos meus pais, Maristela e Gefferson, pelo apoio e pelo incentivo que sempre me motivaram a continuar estudando e a me dedicar aos meus objetivos.

Agradeço à minha esposa, Ana Clara, pelo apoio e pela compreensão ao longo destes anos.

Agradeço à professora Giseli pela orientação, paciência e dedicação durante todo o processo de elaboração deste trabalho.

Agradeço à professora Maria Luiza Machado pela orientação acadêmica ao longo da graduação e por sempre me tranquilizar nos momentos desafiadores.

Agradeço ao João Pedro pela amizade e pela coorientação neste trabalho.

Agradeço às professoras Vivian e Letícia por aceitarem fazer parte da banca de avaliação deste trabalho.

Por fim, agradeço a todos os professores, servidores e funcionários da Universidade Federal do Rio de Janeiro, que, apesar das mais variadas dificuldades e desafios, conseguem cuidar de nossa universidade e manter o ensino público de qualidade.

RESUMO

O crescimento exponencial dos dados conectados, que passaram de 95 para 1350 bases RDF (*Resource Description Framework*) entre 2009 e 2024, ocorreu em paralelo à democratização das ferramentas de BI (*Business Intelligence*). Contudo, a falta de conectores nativos nessas plataformas ainda mantém esses ecossistemas isolados. O trabalho desenvolve um *driver* SPARQL para o Metabase que conecta diretamente a *endpoints* SPARQL via HTTP (*Hypertext Transfer Protocol*) e HTTPS (*Hypertext Transfer Protocol Secure*), executa consultas SPARQL dos tipos SELECT e ASK e expõe o grafo RDF no modelo tabular da ferramenta. O trabalho estabelece 12 critérios de aceitação e implementa a estratégia de tabela de propriedades particionada por classes para representar classes como tabelas e propriedades como colunas. O *driver* descobre metadados (classes e predicados mais frequentes), converte tipos XSD (*XML Schema Definition*) básicos para tipos nativos do Metabase e permite a parametrização com sintaxe de duplas chaves. A avaliação abrange oito *endpoints* SPARQL públicos, incluindo Wikidata, DBpedia e UniProt, com conectividade por HTTPS, incluindo cenários com certificado TLS (*Transport Layer Security*) inválido. Os testes confirmam a execução correta de SELECT e ASK e a criação de 8 visualizações (tabela, barras horizontais, barras verticais, rosca, linha, bolhas, número único e mapa) a partir de resultados tabulares. O projeto disponibiliza o *driver* como artefato JAR (*Java Archive*) e imagem Docker e utiliza fluxo de integração contínua com verificações estáticas, testes unitários e empacotamento automatizado. Os resultados indicam viabilidade técnica e utilidade prática para explorar grafos RDF no Metabase, reduzindo as barreiras entre SPARQL e ferramentas de BI. As principais limitações concentram-se no suporte apenas aos tipos XSD básicos, na ausência de suporte a *endpoints* com autenticação e na exibição limitada às classes mais frequentes no *query builder*; trabalhos futuros podem ampliar a cobertura de tipos, adicionar autenticação e evoluir a conversão da consulta visual para SPARQL.

Palavras-chave: Web Semântica; Dados Conectados; Business Intelligence; SPARQL; RDF.

ABSTRACT

The exponential growth of Linked Data, RDF (Resource Description Framework) datasets expanding from 95 to 1,350 between 2009 and 2024, has unfolded alongside the democratization of Business Intelligence (BI) tools. Yet the lack of native connectors in these platforms still keeps these two ecosystems isolated. This work presents a SPARQL driver for Metabase that connects directly to SPARQL endpoints over HTTP (Hypertext Transfer Protocol) and HTTPS (Hypertext Transfer Protocol Secure), retrieves information by executing SELECT and ASK queries, and exposes RDF graphs in a tabular model. This work defines 12 acceptance criteria and adopts a Property Class Table strategy for visualization in Metabase, representing classes as tables and properties as columns. The driver discovers metadata (frequency analysis of classes and predicates), converts basic XSD (XML Schema Definition) datatypes to native Metabase types, and supports parameterization using a double-brace syntax. The evaluation covers eight public SPARQL endpoints, including Wikidata, DBpedia, and UniProt, with HTTPS connectivity, scenarios involving invalid TLS (Transport Layer Security) certificates. Tests confirm correct execution of SELECT and ASK queries and the creation of eight visualizations (table, horizontal bars, vertical bars, pie, line, bubbles, single value, and map) from tabular results. The project delivers the driver as a JAR (Java Archive) artifact and a Docker image, and uses a continuous integration pipeline with static checks, unit tests, and automated packaging. The results indicate technical feasibility and practical utility for exploring RDF graphs in Metabase, reducing the barriers between SPARQL and BI tools. The main limitations are support restricted to basic XSD types, lack of support for authenticated endpoints, and exposure limited to the most frequent classes in the query builder; future work can broaden datatype coverage, add authentication, and advance the conversion from visual queries to SPARQL.

Keywords: Semantic Web; Linked Data; Business Intelligence; SPARQL; RDF.

LISTA DE ILUSTRAÇÕES

LISTA DE CÓDIGOS

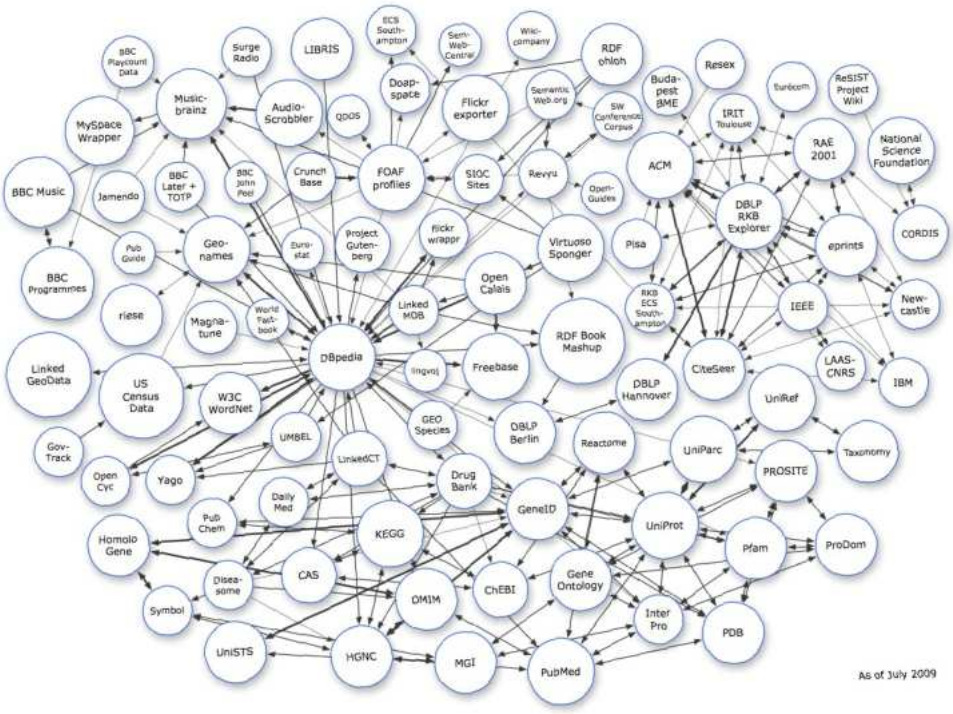
LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Nos últimos quinze anos, o universo de Dados Conectados (*Linked Data*) tem se consolidado como uma rede de informação robusta que amplia a entrega de valor para a sociedade em diversos setores. O diagrama oficial do *Linked Open Data Cloud*[1], site que mapeia bases públicas de dados conectados e as exibe em uma estrutura de nós e conexões, mostra que o número de bases publicadas seguindo o padrão *Resource Description Framework* (RDF) foi de 95, em julho de 2009 (figura 1), para 1350, em dezembro de 2024 (figura 2). Esse aumento foi superior a catorze vezes, o que deixa evidente a adoção crescente dos grafos de conhecimento em diversos contextos, entre eles: governamentais (RAJABI; MIDHA; SOUZA, 2023), científicos (STEAR et al., 2024) e empresariais (WALKER et al., 2022).

Figura 1 – LOD Cloud em 14 de julho de 2009

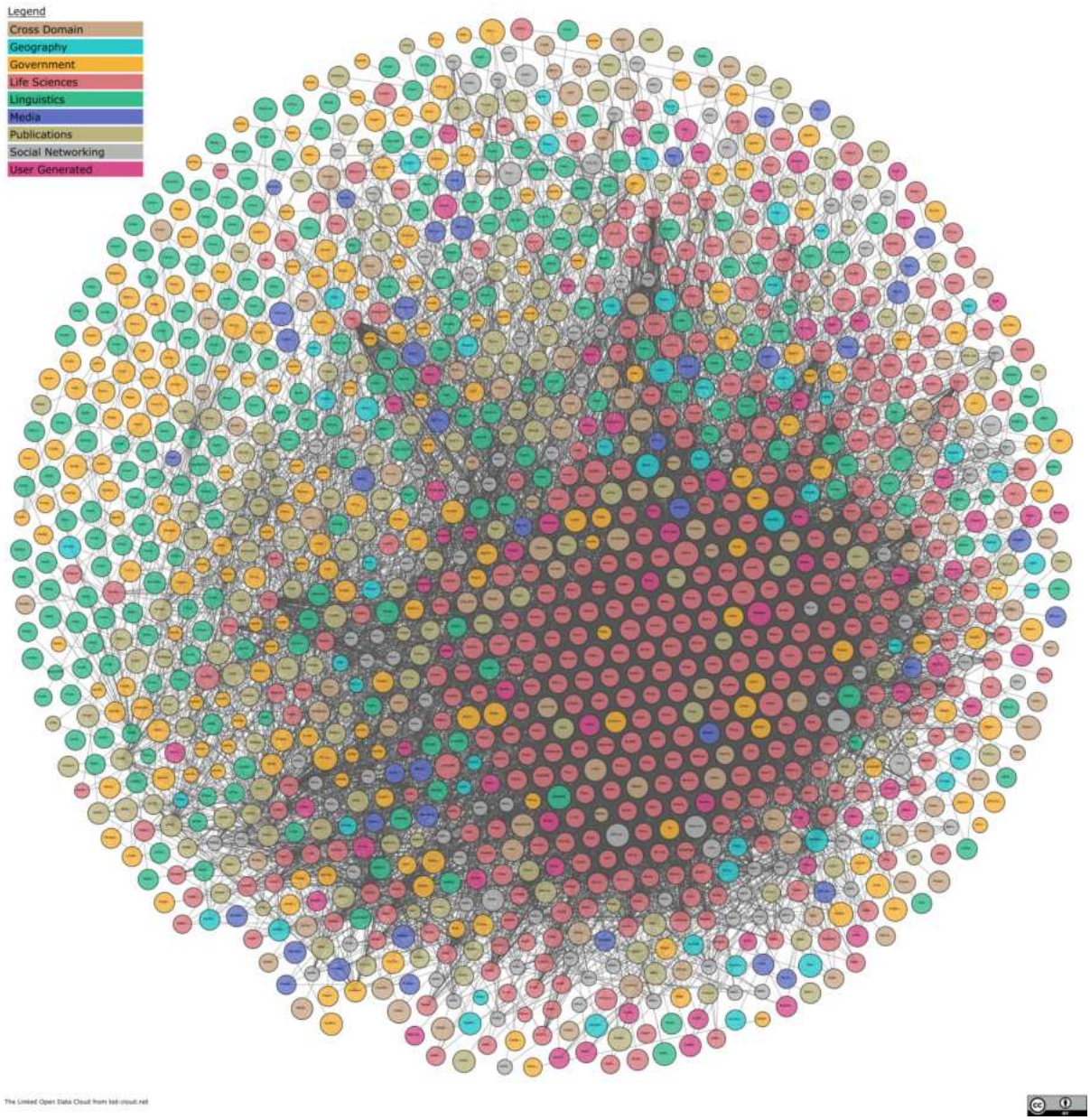


Fonte: https://lod-cloud.net/versions/2009-07-14/lod-cloud.png

No mesmo período, o mercado de *Business Intelligence* (BI) passou por uma forte democratização. Uma das formas de medir essa crescente adoção de ferramentas de BI é por meio da popularidade das principais plataformas *open-source* de visualização de dados, medida pelo número de estrelas no GitHub ao longo dos anos, como mostrado na figura 3. Vale ressaltar que estudos apontam que três em cada quatro desenvolvedores

[1] https://lod-cloud.net

Figura 2 – LOD Cloud em 31 de dezembro de 2024

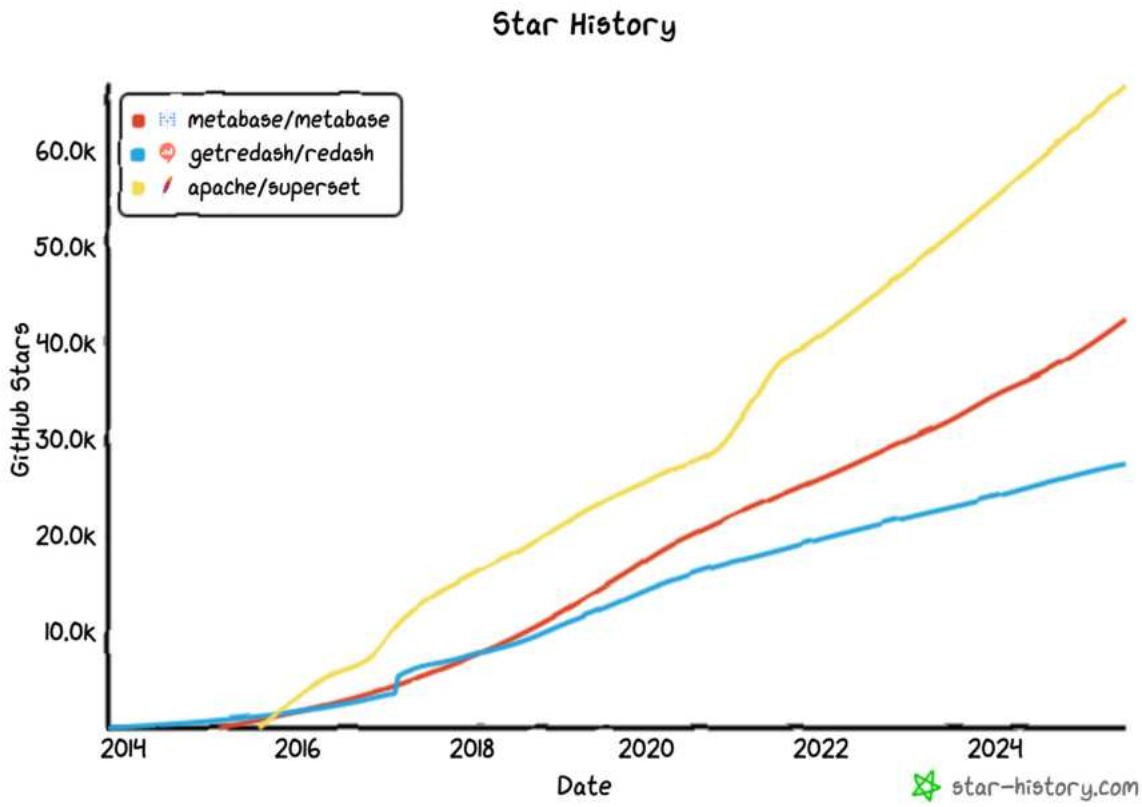


Fonte: https://lod-cloud.net/versions/2024-12-31/lod-cloud.png

utilizam essa métrica para avaliar se utilizariam determinado *software* e que a análise da evolução dessa métrica deve excluir possíveis distorções provenientes de ações temporais de *marketing* (BORGES; VALENTE, 2018). Por exemplo, uma análise mais detalhada da figura 3 mostra que o Apache Superset, o Metabase e o Redash, criados entre 2014 e 2016, evoluíram de menos de 5 mil estrelas em 2017 para, respectivamente, cerca de 65 mil, 40 mil e 25 mil até os primeiros meses de 2025, apontando o interesse da comunidade de dados ao longo de quase 10 anos.

Mesmo com a popularidade das plataformas de BI, ainda há diferenças significativas entre os protocolos de comunicação utilizados pelos bancos de dados relacionais e o da

Figura 3 – Histórico de estrelas das principais ferramentas *open-source* de BI



Fonte: https://www.star-history.com, adicionando os repositórios apache/superset, getredash/redash e metabase/metabase

Web Semântica. De um lado, temos protocolos tradicionais como JDBC (*Java Database Connectivity*) e o esquema rígido do BI; de outro, temos SPARQL sobre o protocolo de rede *Hypertext Transfer Protocol* (HTTP) e o modelo em grafo RDF. Tais diferenças mantêm os dois ecossistemas ainda isolados, o que dificulta a análise dos dados semânticos em ferramentas de BI, fazendo com que analistas e desenvolvedores recorram a *scripts* manuais ou soluções provisórias que visam transformar os dados no modelo tabular esperado por essas ferramentas. Ou seja, o avanço paralelo em ambos os temas não tem sido suficiente para diminuir a lacuna existente nesses dois assuntos (LABORIE et al., 2015).

1.1 OBJETIVOS

Para preencher essa lacuna, este trabalho desenvolve um *driver* para o Metabase, ferramenta de BI *open-source*. O objetivo é permitir a conexão direta da plataforma a *endpoints* SPARQL via HTTP, oferecendo suporte a consultas dos tipos `SELECT` e `ASK`. Além disso, os resultados dessas consultas são integrados à interface visual da ferramenta, permitindo que profissionais de BI explorem dados RDF por meio de gráficos e *dashboards*

interativos.

Para se adequar à representação tabular adotada pelo Metabase, que exibe em sua interface as tabelas de uma fonte de dados e suas respectivas colunas, é feito o mapeamento de classes e propriedades RDF. As classes passam a representar "tabelas" e as propriedades, "colunas", possibilitando que a ferramenta interprete corretamente os metadados e os apresente ao usuário, com o objetivo de facilitar o entendimento do grafo a ser consultado.

A compatibilidade do *driver* é validada em oito *endpoints* SPARQL amplamente utilizados pela comunidade, entre eles, se destacam DBpedia[2] e Wikidata[3], duas grandes fontes públicas de dados conectados. Além disso, foram registradas as limitações identificadas durante os testes e as soluções de contorno aplicadas.

Para demonstrar a utilidade prática, é construído um conjunto de visualizações abrangendo diferentes domínios de dados previamente selecionados. Essas visualizações evidenciam a possibilidade de criar diversos tipos de gráficos suportados pelo Metabase a partir de dados obtidos via consultas SPARQL, sem recorrer a transformações intermediárias.

Toda a solução é acompanhada de documentação para reprodução, incluindo guias de compilação, instalação, configuração e contribuição. Assim, pesquisadores e profissionais podem replicar e estender o produto resultante. O código-fonte está disponível em repositório público no GitHub, sob a licença Affero General Public License[4], respeitando a licença original do Metabase.

Também faz parte do escopo deste trabalho a implantação de um fluxo de integração contínua (CI). Esse fluxo é capaz de: (i) executar análises estáticas de código para garantir padrões de estilo e detecção precoce de erros, (ii) rodar testes unitários, (iii) empacotar o artefato resultante em um arquivo JAR versionado e (iv) publicar automaticamente esse JAR no repositório git do projeto. O objetivo do CI é garantir a qualidade do código, a rastreabilidade de compilações e a facilidade de distribuição para uso pela comunidade, eliminando etapas manuais e encurtando o tempo entre contribuições de código e distribuição do *driver* compilado.

As medições que consideram fatores externos, como latência de rede, são excluídas do escopo, uma vez que *endpoints* públicos podem sofrer variações imprevisíveis de carga causadas por uso da comunidade. Da mesma forma, não são implementadas camadas de autenticação para acesso a *endpoints* SPARQL. No entanto, ambos os aspectos permanecem como possibilidades para trabalhos futuros, caso sejam identificados como relevantes.

1.2 JUSTIFICATIVA

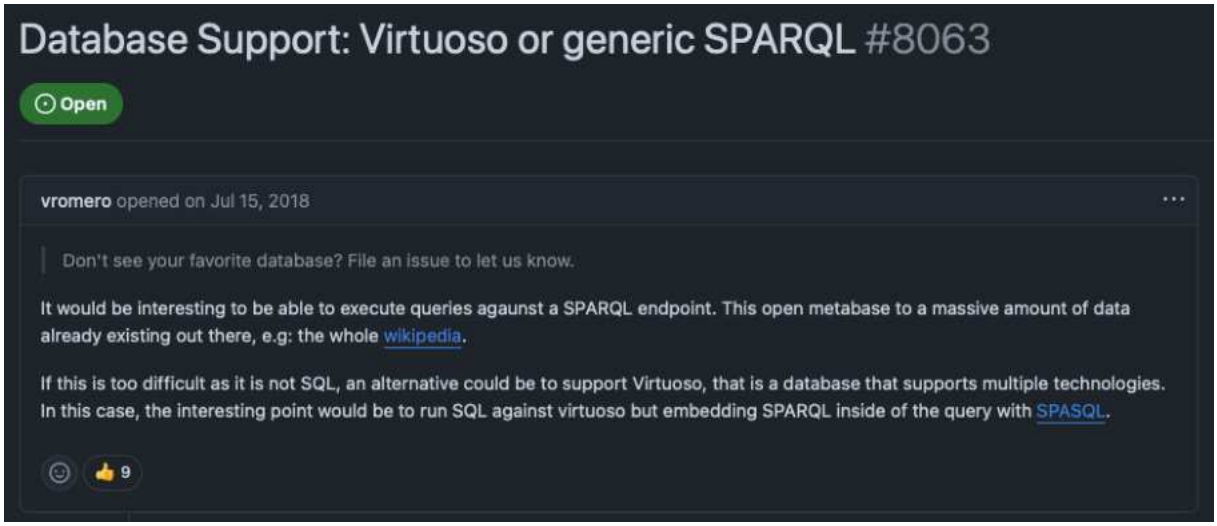
O uso de SPARQL por meio de ferramentas de BI é apontado como desafio desde o início da década de 2010. Segundo Leida et al. (2011), analistas precisavam utilizar *scripts*

[2]https://www.dbpedia.org
[3]https://www.wikidata.org
[4]https://www.gnu.org/licenses/agpl-3.0.html

manuais para converter resultados de consultas em grafos RDF em formatos tabulares, identificando a falta de conectores nativos como obstáculo significativo à adoção de Dados Conectados e da Web Semântica.

A falta de suporte nativo também pode ser verificada no repositório da própria comunidade do Metabase: a discussão de número 8063 (figura 4), cujo título é *"Database Support: Virtuoso or generic SPARQL"* e que foi aberta em 2018 e ainda está sem resolução, possui reações positivas e sugestões de contribuições, demonstrando uma demanda ainda não atendida. Complementando a necessidade de uma ponte entre esses dois universos, a discussão de número 4510, "Use superset with SPARQL endpoint", no repositório do Apache Superset foi aberta também em 2018 e fechada em 2019 sem solução (figura 5), porém, nessa mesma discussão, há comentários recentes de 2023 indicando ainda a necessidade de uma solução para este problema.

Figura 4 – Issue #8063 no repositório do Metabase no GitHub



Fonte: https://github.com/metabase/metabase/issues/8063

A relevância acadêmica desta implementação contribui para o estudo de interoperabilidade e visualização de dados semânticos por meio de gráficos, além de possibilitar a difusão desses temas e contribuir para trabalhos futuros. O desenvolvimento de um *driver open-source* poderá servir como base de estudo para mapeamento visual de dados em formato de grafo para o formato tabular, experiência do usuário com interface amigável para geração de visualizações e novas abordagens de integração de dados semânticos.

Na esfera prática, o resultado deste trabalho diminui uma barreira técnica que impede organizações e pesquisadores de explorar o ecossistema de dados abertos em RDF, como os da DBpedia e Wikidata, por meio de uma plataforma de BI de autosserviço. Instituições de ensino e pesquisa, órgãos governamentais e empresas que já dispõem de equipes de BI poderão utilizar essas fontes externas para obter informações ricas, sem a necessidade de transformações manuais.

Figura 5 – Issue #4510 no repositório do Apache Superset no GitHub

Fonte: https://github.com/apache/superset/issues/4510

Do lado social, facilitar o acesso a dados da Web Semântica pode acelerar iniciativas de transparência governamental, ciência aberta e jornalismo de dados. Ao tornar grafos RDF acessíveis em ferramentas de BI amplamente adotadas, o projeto reduz, embora não elimine, barreiras que limitam o uso de dados abertos.

Para o autor, o desenvolvimento deste trabalho foi uma oportunidade de aprofundar conhecimentos em dados conectados, RDF e Web Semântica. Também permitiu o aprendizado de novas competências, como o uso da linguagem Clojure e de conceitos de programação funcional. Além disso, implementar um fluxo de integração contínua ampliou a experiência técnica do autor no que diz respeito à automação de testes, empacotamento e distribuição de *software*. Por fim, as atividades de planejamento, implementação e validação do *driver* contribuíram para consolidar conceitos de engenharia e arquitetura de *software*, fortalecendo tanto a formação profissional quanto a acadêmica.

1.3 METODOLOGIA

Foi adotada uma abordagem de engenharia de *software* baseada em entregas incrementais, implementando pequenas funcionalidades por vez e sem utilizar ciclos com janelas de tempo fixas, como ocorre no método Scrum. Para a organização e o acompanhamento do desenvolvimento, optou-se pelo método Kanban, que se mostrou mais adequado a um contexto de tarefas contínuas e iterações com durações variadas. Diferentemente do Scrum, que exige *sprints* de duração fixa e cerimônias estruturadas, o Kanban oferece maior flexibilidade para um projeto individual conduzido em paralelo à escrita acadêmica, permitindo ajustes dinâmicos de prioridade conforme o avanço da pesquisa.

A primeira fase se concentrou na análise da arquitetura da plataforma de BI para compreender sua estrutura de *drivers* e como ela se comunica com as diferentes fontes de dados suportadas. Esta etapa envolveu o estudo de documentação técnica, código,

exemplos de *drivers* existentes e qualquer outro material útil para a compreensão das camadas de abstração e interfaces necessárias para a implementação.

A segunda etapa consistiu no estudo do protocolo SPARQL, do formato *JavaScript Object Notation* (JSON) retornado pelo servidor e no aprofundamento dos conceitos da Web Semântica, essenciais para que a solução pudesse converter e interpretar de forma correta os dados recebidos. Foram analisados os diferentes tipos de consultas SPARQL e suas particularidades. Além disso, foi realizado o estudo do armazenamento de dados conectados em bancos de dados relacionais, o que serviu de suporte para a representação visual das classes e propriedades do grafo RDF como "tabelas" e "colunas".

A terceira fase correspondeu ao desenvolvimento da solução. Nesta etapa, foram criados os componentes necessários para estabelecer conexões com *endpoints* SPARQL, realizar o mapeamento entre modelos de dados, executar consultas e processar resultados. Três desafios centrais nessa implementação foram a transformação do modelo RDF para o modelo tabular da plataforma de BI, preservando a semântica original dos dados, a conversão de tipos de dados entre os diferentes formatos e a tradução entre diferentes linguagens de consulta.

A quarta fase constituiu a validação e avaliação da solução, utilizando *endpoints* SPARQL públicos como DBpedia e Wikidata para verificar a eficácia da abordagem proposta. Foi verificada a capacidade de conexão, a execução de consultas e a visualização de resultados. Os testes incluíram diferentes tipos de consultas SPARQL e cenários de uso, permitindo identificar limitações e oportunidades de melhoria. A avaliação foi realizada por meio de critérios de aceitação previamente definidos, comprovando o cumprimento dos objetivos estabelecidos. Também foram discutidas e documentadas as limitações identificadas e possíveis soluções de contorno, contribuindo para o conhecimento sobre integração entre ferramentas de BI e dados RDF.

Por fim, a documentação e disponibilização constituíram a etapa final, garantindo que o trabalho possa ser reproduzido e ampliado, facilitando trabalhos futuros. Esta fase também compreendeu a elaboração de documentação para o uso da solução e a replicação dos resultados, e a disponibilização do artefato gerado por este trabalho de maneira pública e irrestrita, garantindo que o projeto possa ser utilizado por outros pesquisadores e profissionais de BI.

1.4 ESTRUTURA DO TRABALHO

Este trabalho é organizado em sete capítulos, estruturados de forma a guiar o leitor desde a contextualização do problema até a apresentação das conclusões. A forma como os capítulos foram organizados busca manter uma progressão lógica, permitindo que cada seção sirva de base para a compreensão da próxima.

O capítulo 1 apresenta o contexto no qual a pesquisa se desenvolve, descrevendo o

cenário atual, as motivações e os problemas que justificam sua realização. Também aborda os objetivos que orientam o desenvolvimento do estudo, assim como detalha a metodologia utilizada e apresenta a estrutura geral do documento, permitindo que se conheçam os conteúdos que serão abordados nos capítulos seguintes.

O capítulo 2 apresenta a fundamentação teórica necessária para a compreensão do trabalho. São abordados os conceitos de Linked Data e RDF, o protocolo SPARQL, bem como os fundamentos de BI. Essa base permite compreender os desafios técnicos da integração entre os dois temas.

O capítulo 3 expõe os trabalhos relacionados. É realizada uma revisão das iniciativas existentes para integração entre SPARQL e ferramentas de visualização de dados em formato gráfico a partir de consultas em grafos RDF. Também são apresentadas diferentes abordagens e suas limitações.

O capítulo 4 justifica a escolha da plataforma de BI utilizada como base para a implementação. É apresentado o processo de seleção das ferramentas de BI *open-source* disponíveis, incluindo a análise comparativa de critérios como suporte a linguagens de consulta, variedade de visualizações e arquitetura modular. O capítulo detalha as características do Metabase, sua estrutura de *drivers* e a linguagem *Metabase Query Language* (MBQL), fundamentando a decisão técnica de utilizá-la como plataforma base para o desenvolvimento do *driver* SPARQL.

O capítulo 5 aborda tanto o planejamento quanto a implementação do *driver*. Inicialmente, apresenta a arquitetura do *driver* SPARQL para Metabase, seus requisitos, critérios de aceitação e as estratégias de mapeamento visual entre o modelo de dados em grafo e o modelo tabular. Em seguida, detalha a implementação, incluindo as tecnologias utilizadas, os componentes criados e os desafios enfrentados durante o desenvolvimento. O processo de integração contínua também é descrito neste capítulo.

O capítulo 6 discorre sobre a avaliação e a discussão dos resultados. Primeiramente, descreve os testes realizados com base nos critérios de aceitação definidos no capítulo 5. Na sequência, executa consultas a *endpoints* SPARQL públicos contemplando consultas dos tipos `SELECT` e `ASK`. Em seguida, avalia as conversões de tipos do SPARQL para os tipos nativos do Metabase e apresenta os gráficos construídos no ambiente de BI. Por fim, oferece uma análise crítica e abrangente do *driver*, destacando compatibilidades, limitações e oportunidades de evolução.

Por fim, o capítulo 7 apresenta as conclusões e aponta caminhos para trabalhos futuros. As contribuições do projeto, as lições aprendidas durante seu desenvolvimento e as direções para pesquisas e melhorias que podem expandir o impacto desta iniciativa são resumidas e sugeridas.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta as teorias que fundamentam a pesquisa e guiam as escolhas do *driver* SPARQL para o Metabase. Inicia-se pelos princípios de Business Intelligence e plataformas de BI de autosserviço e, em seguida, passa pela revisão dos fundamentos da Web Semântica, como RDF, ontologias e a linguagem de consulta SPARQL, com foco nos formatos de resposta em JSON e nos tipos de dados retornados. Ao final, é discutida a estratégia de mapeamento grafo → tabela que serve como base para a representação de classes e propriedades na interface visual da plataforma de BI escolhida. Essa linha lógica fornece a base necessária para compreender as decisões de projeto desenvolvidas nos capítulos seguintes.

2.1 BUSINESS INTELLIGENCE

Business Intelligence, ou Inteligência de Negócios, refere-se ao conjunto de práticas que visam transformar dados brutos em informações valiosas, capazes de proporcionar maior agilidade na tomada de decisões (CHAUDHURI; DAYAL; NARASAYYA, 2011). Transformar grandes volumes de dados operacionais, transacionais e analíticos em conhecimento útil e acionável é de extrema importância nesse processo, porém oferecer autonomia para os usuários é igualmente importante. Isso torna o uso dos dados e do BI mais democrático, até para usuários menos familiarizados com ferramentas analíticas.

Um fluxo típico de uma plataforma de BI envolve etapas de extração, transformação e carga dos dados (em inglês, *extract, transform and load*, ETL). Esse processo consiste em extrair dados de determinada fonte, transformá-los em um formato adequado para análise e carregá-los em um destino centralizado, e, por fim, visualizá-los em *dashboards* ou outros sistemas de apoio à decisão.

Esses processos variam conforme a abordagem adotada. Há casos em que ferramentas de visualização se conectam diretamente a bancos de dados operacionais ou a réplicas de leitura. Em arquiteturas baseadas em eventos, também é possível realizar transformações em tempo real ou sob demanda, no momento da consulta. Embora a proposta apresentada neste trabalho não implemente um fluxo de transformação de dados tradicional (ingestão, transformação e carregamento), o *driver* executa as transformações sob demanda, conforme as consultas dos usuários, com o objetivo de exibir os resultados na plataforma de visualização.

Essas transformações sob demanda são realizadas por plataformas de BI de autosserviço (*self-service BI*), que são ferramentas que permitem aos analistas criar relatórios, consultas e visualizações sem depender de equipes de programadores ou profissionais de dados altamente especializados (ALPAR; SCHULZ, 2016). Elas empoderam usuários dos

mais diferentes níveis técnicos a utilizar e a obter informações importantes por meio da exploração de dados e da construção de gráficos a partir dos dados consultados. Essa autonomia oferecida por essas ferramentas é muito importante para a democratização do acesso aos dados, pois reduz as etapas necessárias até que se obtenha acesso à informação desejada.

As plataformas de BI de autosserviço oferecem interfaces visuais intuitivas, como construtores de *dashboards* por arrastar e soltar (*drag-and-drop*), bibliotecas de visualizações pré-configuradas, possibilidade de construir consultas a partir de blocos interativos e capacidade de conexão direta a diversas fontes de dados. Essa abordagem permite que profissionais com diferentes níveis de conhecimento técnico possam acessar, explorar e interpretar os dados de forma independente e mais ágil.

Dentre as soluções mais adotadas por empresas e instituições, destacam-se ferramentas proprietárias como Tableau[1], Power BI[2] e Looker Studio[3]. No contexto de soluções de código aberto, existem diversas alternativas que oferecem funcionalidades comparáveis às das plataformas proprietárias, com a vantagem adicional de permitirem modificações e extensões conforme necessidades específicas, como é o caso da proposta deste trabalho, dado que seus códigos são abertos e podem ser modificados para atender a diferentes contextos.

É importante destacar que processos de ETL tradicionais continuam sendo fundamentais em cenários que envolvem integração de múltiplas fontes, limpeza de dados, governança e desempenho em ambientes analíticos dedicados. No contexto deste trabalho, a abordagem de consulta direta aos *endpoints* SPARQL é vantajosa em cenários como: (i) análise exploratória de dados públicos de grafos RDF sem necessidade de transformação ou persistência; e (ii) desenvolvimento direto de visualizações. Nesses casos, evitar a criação de *scripts* manuais e transformações intermediárias amplia o número de pessoas capazes de acessar e interpretar os dados RDF.

2.2 MODELO DE DADOS RDF

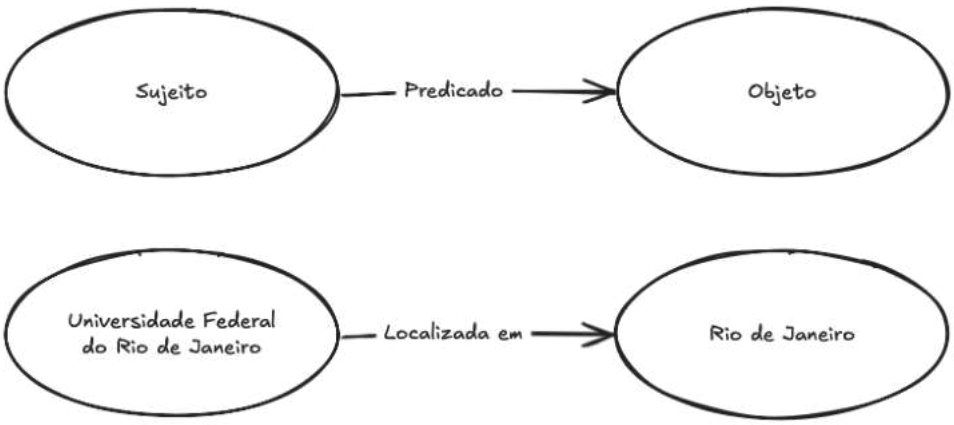
O *Resource Description Framework* (RDF) é um modelo de representação de dados da web definido pelo *World Wide Web Consortium* (W3C) (WOOD; LANTHALER; CYGANIAK, 2014). Uma tripla RDF é caracterizada por sujeito, predicado e objeto, que pode ser representada como $t = (s, p, o)$.

No exemplo da figura 6, o sujeito é a entidade "`Universidade Federal do Rio de Janeiro`", o predicado é a relação "`localizada em`" e o objeto é a entidade "`Rio de Janeiro`". Observa-se a estrutura de rede formada por um conjunto de triplas RDF quando, estendendo o grafo de exemplo, é possível verificar que o Rio de Janeiro está

[1] https://www.tableau.com/
[2] https://powerbi.microsoft.com/
[3] https://lookerstudio.google.com/

localizado no Brasil, país em que a Universidade Federal do Rio de Janeiro (UFRJ) também se encontra.

Figura 6 – Representação de uma tripla RDF genérica e com valores do mundo real



Fonte: Elaborado pelo autor

Na figura 7, a primeira tripla liga a universidade ao estado em que ela se localiza e a segunda liga o estado ao país onde ele está situado. Observe que o estado ora é sujeito de uma tripla, ora é objeto.

Figura 7 – Representação de duas triplas RDF evidenciando a relação entre três entidades

Fonte: Elaborado pelo autor

$t_1 =$ (‘`Universidade Federal do Rio de Janeiro`’, ‘`Localizada em`’, ‘`Rio de Janeiro`’)

$t_2 =$ (‘`Rio de Janeiro`’, ‘`Localizada em`’, ‘`Brasil`’)

Um sujeito pode ser representado como uma *Internationalized Resource Identifier* (IRI) ou nó em branco (*blank node*). A IRI é um identificador único, por exemplo, `<http://dbpedia.org/resource/Federal_University_of_Rio_de_Janeiro>`. O nó em branco, por sua vez, representa um recurso ainda sem identificador. Já o predicado deve, obrigatoriamente, ser uma IRI, como `<http://dbpedia.org/property/state>`. Por fim, o objeto pode ser novamente uma IRI ou um nó em branco, ou ainda um literal, por exemplo “Brasil”.

2.3 WEB SEMÂNTICA E DADOS CONECTADOS

O termo *Web Semântica* foi cunhado por Berners-Lee, Hendler e Lassila (2001), ao proporem uma nova abordagem para disponibilizar conteúdos na internet, de modo que as máquinas pudessem compreender mais facilmente os dados disponíveis. Para atingir esse objetivo, a proposta foi utilizar tecnologias que já haviam sido desenvolvidas nos anos anteriores, como o RDF, cujo primeiro rascunho surgiu em 1997 e cuja versão 1.0 foi publicada em 1999, além do *eXtensible Markup Language* (XML), utilizado como linguagem de marcação para serializar triplas RDF.

No mesmo artigo em que foi introduzido o termo *Web Semântica*, também foi incorporado o conceito de ontologias, entendido como uma forma de descrever formalmente os conceitos e suas relações dentro de um domínio específico (BERNERS-LEE; HENDLER; LASSILA, 2001). As ontologias têm como base a lógica descritiva (*Description Logic*) que contém dois conceitos principais: o *Terminological Box* (*TBox*) e o *Assertional Box* (*ABox*). O *TBox* define o esquema conceitual da ontologia, incluindo classes, propriedades, hierarquias e restrições. Já o *ABox* contém informações sobre as instâncias (ou indivíduos), representando entidades concretas de um domínio modelado (USCHOLD, 2018).

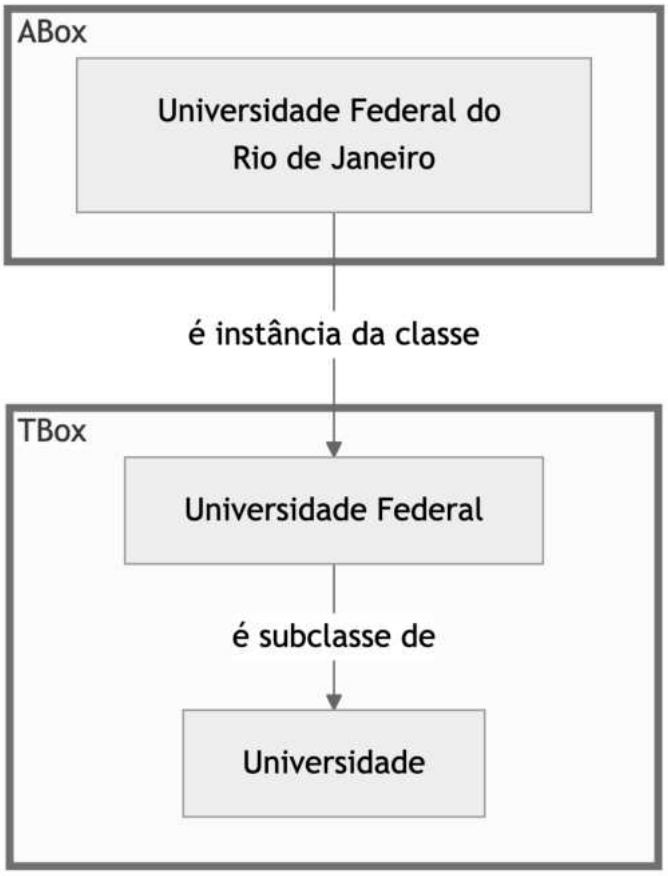
Por exemplo, a entidade `Universidade Federal do Rio de Janeiro` pode ser modelada como uma instância da classe `Universidade Federal`, que por sua vez pode ser definida como uma subclasse da classe mais genérica `Universidade`. Nesse caso, a instância pertenceria ao *ABox*, enquanto as classes e suas hierarquias fariam parte do *TBox*, como representado na figura 8.

As definições modeladas pelas ontologias representam, em grande maioria, um modelo do mundo real e diferentes ontologias podem modelar um mesmo domínio. Isso cria uma relação de alinhamento entre ontologias, na qual um único indivíduo na web pode estar representado de diversas maneiras. Por exemplo, a classe *Universidade* é representada na DBpedia como `<http://dbpedia.org/ontology/University>` e no schema.org como `<https://schema.org/CollegeOrUniversity>`. Esse alinhamento de ontologias permite a interoperabilidade dos dados que elas representam, isto é, viabiliza o reuso e a integração dos dados por diferentes sistemas, favorecendo cruzamentos e aplicações semânticas mais ricas.

Apesar das tecnologias já estarem desenvolvidas, era preciso definir boas práticas para compartilhamento de informações na web. Foi nesse contexto que surgiu o conceito de Dados Conectados. Berners-Lee (2006) propôs 4 princípios para publicar dados na Web Semântica:

1. Usar IRI para nomear recursos;
2. Usar URLs como IRIs para que o recurso possa ser localizado na Web;

Figura 8 – Representação do TBox e ABox



Fonte: Elaborado pelo autor

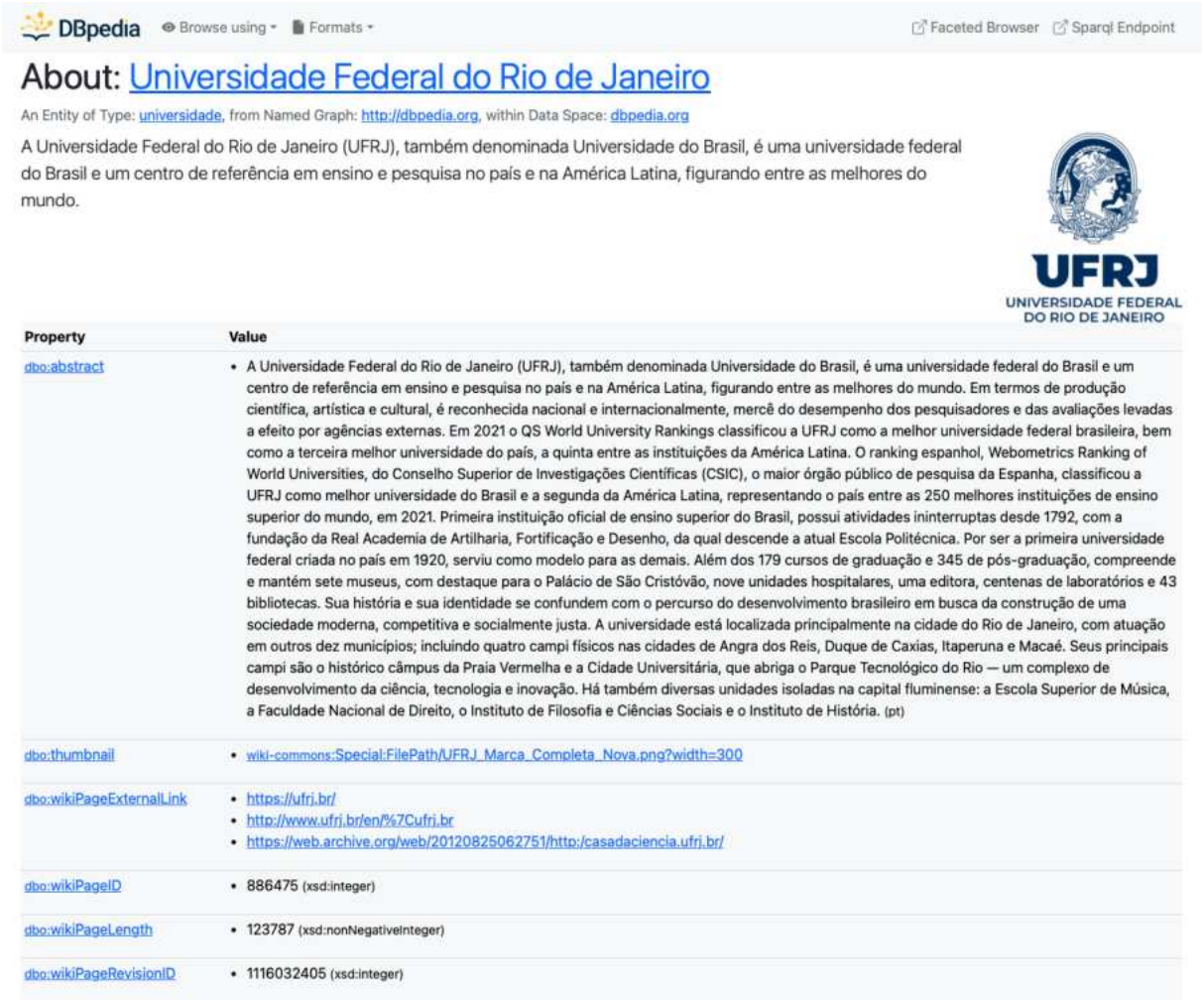
3. Prover informações úteis quando um usuário acessar a IRI que também é uma URL, pelo princípio 2;

4. Incluir links para outras IRIs.

Todos esses princípios podem ser verificados com a IRI da UFRJ na DBpedia (figura 9). A universidade tem uma IRI bem definida `<http://dbpedia.org/resource/Federal_University_of_Rio_de_Janeiro>` atendendo ao primeiro princípio. Sua IRI representa uma URL (*Uniform Resource Locator*) que pode ser acessada através de um navegador web. Ela apresenta informações úteis ao usuário. E, por fim, ela apresenta links para outros recursos através de IRIs.

É necessário um meio formal de consulta desses dados, tanto para humanos quanto para máquinas. O SQL (*Structured Query Language*) atua bem nesse papel, uma linguagem bem definida e amplamente utilizada para bancos de dados relacionais. Para a Web Semântica, por sua vez, foi criada a linguagem SPARQL.

Figura 9 – Página na DBpedia sobre a UFRJ



Fonte: https://dbpedia.org/page/Federal_University_of_Rio_de_Janeiro

2.4 LINGUAGEM DE CONSULTA SPARQL

O SPARQL pode ser definido como a linguagem de consulta e um protocolo da Web Semântica. Sua especificação completa da versão 1.1 foi definida pelo W3C em 2013 (W3C SPARQL Working Group, 2013), porém sua primeira versão, a 1.0, surgiu em 2008. Uma consulta SPARQL se assemelha a uma consulta utilizando o SQL, porém seu foco é obter resultados sobre o grafo RDF e não sobre o banco de dados relacional, como a maioria dos analistas de dados está acostumada.

O tipo de consulta mais popular do SPARQL é o `SELECT` (DUCHARME, 2013), porém há outros 3 tipos: `ASK`, `DESCRIBE` e `CONSTRUCT`. A seguir, cada um deles será apresentado junto com um exemplo:

- `SELECT`: Consultas desse tipo retornam um conjunto de colunas e linhas, semelhante a uma consulta SQL. Ela pode conter transformações, como agregações, filtros e

ordenações. O exemplo do código 1 lista os 3 estados mais populosos do Brasil com seus respectivos números de habitantes.

Código 1 – Exemplo de uma consulta `SELECT` em SPARQL (DBpedia)

```
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbp: <http://dbpedia.org/property/>
PREFIX dct: <http://purl.org/dc/terms/>
PREFIX dbc: <http://dbpedia.org/resource/Category:>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT DISTINCT ?state ?population WHERE {
  ?state dct:subject dbc:States_of_Brazil .
  OPTIONAL { ?state dbo:populationTotal ?p1 }
  OPTIONAL { ?state dbp:populationEst ?p2 }
  BIND (COALESCE(?p1, ?p2) AS ?populationRaw)
  FILTER (BOUND(?populationRaw))
  BIND (xsd:integer(?populationRaw) AS ?population)
}
ORDER BY DESC(?population)
LIMIT 3
```

- `ASK`: Consultas desse tipo retornam um valor booleano, ou seja, verdadeiro ou falso, indicando se no grafo existe uma tripla ou um conjunto de triplas que satisfaça a consulta. O exemplo do código 2 verifica se o Rio de Janeiro é uma cidade no grafo da DBpedia.

Código 2 – Exemplo de uma consulta `ASK` em SPARQL (DBpedia)

```
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbr: <http://dbpedia.org/resource/>

ASK
WHERE {
  dbr:Rio_de_Janeiro a dbo:City .
}
```

- `DESCRIBE`: Retorna um grafo RDF contendo triplas relacionadas ao recurso especificado na consulta. A seleção das triplas é definida pelo próprio *endpoint* e pode incluir informações diretamente associadas à entidade, assim como de recursos conectados a ela. O exemplo do código 3 descreve informações sobre a Universidade Federal do Rio de Janeiro.

Código 3 – Exemplo de uma consulta `DESCRIBE` em SPARQL (DBpedia)

```
PREFIX dbr: <http://dbpedia.org/resource/>

DESCRIBE dbr:Federal_University_of_Rio_de_Janeiro
```

- `CONSTRUCT`: O retorno desse tipo de consulta é um grafo RDF que representa o resultado, definido pelo padrão de triplas especificado pelo usuário no `CONSTRUCT`. O exemplo do código 4 constrói um grafo com informações de população de cidades brasileiras.

Código 4 – Exemplo de uma consulta `CONSTRUCT` em SPARQL (DBpedia)

```
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbr: <http://dbpedia.org/resource/>

CONSTRUCT {
  ?city dbo:populationTotal ?population .
}
WHERE {
  ?city a dbo:City ;
        dbo:country dbr:Brazil ;
        dbo:populationTotal ?population .
}
LIMIT 10
```

Como definido nos objetivos deste trabalho, são utilizadas apenas consultas dos tipos `SELECT` e `ASK`, que permitem obter dados em formato tabular. Consultas que retornam grafos RDF, como `CONSTRUCT` e `DESCRIBE`, não são abordadas, pois, no contexto deste trabalho, não apresentam utilidade para os tipos de gráficos propostos.

O SPARQL oferece uma gama de formatos para o retorno de consultas, incluindo XML (HAWKE et al., 2013a), JSON (HAWKE et al., 2013b) e *Comma-Separated Values* (CSV) (HAWKE et al., 2013c), além de opções para retornar grafos RDF dependendo do tipo de consulta, como as consultas `DESCRIBE` e `CONSTRUCT`. No entanto, o formato JSON é o foco principal deste trabalho, dada sua ampla adoção no ecossistema web, facilitando a integração e a desserialização de respostas de consultas SPARQL por meio de bibliotecas específicas para a correta interpretação de dados em formato JSON.

O JSON presente em código 5 representa o retorno da consulta apresentada no código 1. No caminho `head` → `vars` do resultado, estão os nomes das colunas que estavam na consulta SPARQL: `state` e `population`. No caminho `results` → `bindings`, estão representados os dados que satisfazem a consulta. É possível observar, desta forma, que já podemos representar o resultado da consulta no formato tabular, como mostrado na tabela 1.

Código 5 – Exemplo de retorno de consultas SPARQL em JSON

```
{
 "head":{"link":[],"vars":["state","population"]},
 "results":{"distinct": false , "ordered": true , "bindings":[
  {"state":{
    "type":"uri",
    "value":"http://dbpedia.org/resource/Sao_Paulo_(state)"},
   "population":{
    "type":"typed-literal",
    "datatype":"http://www.w3.org/2001/XMLSchema#integer",
    "value":"41262199"}},
  {"state":{
    "type":"uri",
    "value":"http://dbpedia.org/resource/Minas_Gerais"},
   "population":{
    "type":"typed-literal",
    "datatype":"http://www.w3.org/2001/XMLSchema#integer",
    "value":"21411923"}},
  {"state":{
    "type":"uri",
    "value":"http://dbpedia.org/resource/Rio_de_Janeiro_(state)"},
   "population":{
    "type":"typed-literal",
    "datatype":"http://www.w3.org/2001/XMLSchema#integer",
    "value":"15989929"}}
 ]}
}
```

Tabela 1 – Resultados da consulta SPARQL de população de estados brasileiros

state	**population**
http://dbpedia.org/resource/Sao_Paulo_(state)	41262199
http://dbpedia.org/resource/Minas_Gerais	21411923
http://dbpedia.org/resource/Rio_de_Janeiro_(state)	15989929

Fonte: Elaborado pelo autor com dados do DBpedia

Com a representação tabular dos dados, falta definir os tipos. Em uma entidade presente nos resultados representados em código 5, temos chaves com nome `type` e `datatype`, onde aquele representa o tipo de nó do grafo RDF e este o formato do dado caso o tipo seja um literal. A tabela 2 mostra as possibilidades de representações possíveis de acordo com a especificação do SPARQL 1.1 (HAWKE et al., 2013b).

Os tipos de dados (*datatypes*) podem variar conforme a representação desejada. Por exemplo, uma data pode ser representada no formato “2025-07-10” ou no formato “2025-

Tabela 2 – Conversão de termos RDF para formato JSON de acordo com SPARQL 1.1

RDF Term	**JSON form**
IRI ***I***	`{"type": "uri", "value": "I"}`
Literal ***S***	`{"type": "literal", "value": "S"}`
Literal ***S*** with language tag ***L***	`{"type": "literal", "value": "S", "xml:lang": "L"}`
Literal ***S*** with datatype IRI ***D***	`{"type": "literal", "value": "S", "datatype": "D"}`
Blank node, label ***B***	`{"type": "bnode", "value": "B"}`

Fonte: (HAWKE et al., 2013b)

07-10T00:00:00". Os tipos de dados podem ser definidos de diversas maneiras, porém a mais comum de encontrarmos nas bases de dados RDF é a definida pela recomendação do W3C: *XML Schema: Datatypes* (BIRON; MALHOTRA; GROUP, 2004).

Uma das maneiras mais comuns de representação de tipos de dados no RDF é a utilização do XML Schema Definition (*XSD*). Por definição, tudo o que está em um tipo literal é uma *string* comum sem formato definido, mas muitas vezes é necessário representar tipos comuns de dados, como inteiro, ponto flutuante, data ou um booleano.

Outros tipos de dados podem ser utilizados para representação do dado, por exemplo, código postal ou número de telefone. Podemos citar a *Quantities, Units, Dimensions and Data Types Ontologies* (QUDT)[4]; ela é usada para representar unidades de medida e tem o tipo lógico *UNITS* com a IRI `<http://qudt.org/3.1.4/vocab/unit>`, contrastando com os tipos definidos pelo XSD.

Para efeitos deste trabalho, são mapeados apenas os tipos básicos do XSD para os formatos internos da plataforma de BI escolhida no capítulo 4. Como as possibilidades e variedades de representações de tipos de dados disponíveis podem ser muito grandes, a conversão foca somente nesse conjunto de representações de tipos de dados.

2.5 MAPEAMENTO GRAFO-TABELA

Um aspecto fundamental para o desenvolvimento do *driver* é o mapeamento grafo-tabela, que serve como ponte entre o modelo de dados RDF e o modelo tabular utilizado na interface da plataforma de BI escolhida. Esse tipo de mapeamento é complexo, pois envolve escolhas que podem afetar diretamente a experiência do usuário. O usuário precisa ver na plataforma de BI os principais elementos do grafo RDF representados como tabelas e colunas; caso contrário, o *driver* serviria apenas para executar consultas SPARQL em uma interface amigável, e o objetivo é ir além disso.

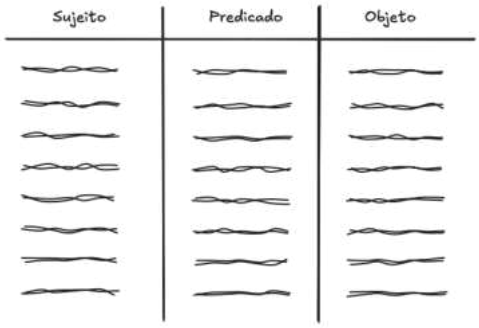
É necessário implementar uma conversão visual que permita, no mínimo, que o usuário veja os principais elementos do grafo RDF diretamente na interface. Para isso, a literatura sobre armazenamento de dados RDF em bancos de dados relacionais é utilizada (YUAN

[4]https://www.qudt.org

et al., 2023). Esses estudos não focam no mapeamento visual aqui proposto; porém, como a literatura específica é limitada, esses estudos são adotados para embasar as escolhas e a implementação do *driver*. A seguir, são apresentados os principais tipos de mapeamentos que podem ser utilizados para esse fim.

Uma das abordagens mais comuns para transformar dados em formato de grafo para o modelo tabular é a tabela de triplas (*triple table*) (ALEXAKI et al., 2001). Esse modelo consiste em uma única tabela com três colunas, uma para o sujeito, outra para o predicado e outra para o objeto, como pode ser visto no esboço apresentado na figura 10.

Figura 10 – Tabela de Triplas (*Triple Table*)



Fonte: Elaborado pelo autor

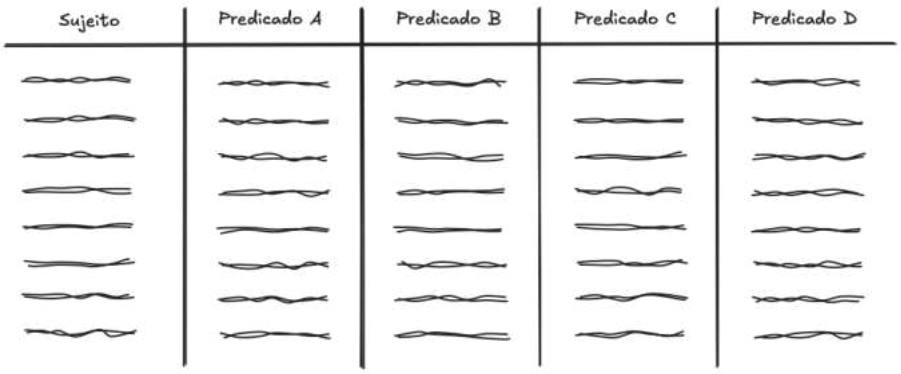
Embora essa forma de representação seja uma das mais básicas a serem implementadas para o modelo, ela pouco representa o grafo RDF que será conectado pelo *driver*, pois não apresenta as classes e propriedades. As classes e propriedades exibidas pela interface visual poderão ajudar o usuário a entender melhor o grafo a ser consultado.

Outra abordagem de mapeamento de dados RDF para o modelo relacional é a tabela de propriedades estendida (*Wide Property Table*). Ela é caracterizada por ser uma única tabela composta de uma coluna com os sujeitos e as demais colunas são os predicados, como pode ser visto no esboço apresentado na figura 11. Uma observação importante a ser feita é que esse tipo de tabela pode ser altamente esparsa, pois nem todos os sujeitos possuem todos os predicados.

A tabela de propriedades estendida, apesar de representar bem os predicados, deixa de lado a representação das classes. Então, ela ainda não representa muito bem o objetivo desta proposta de representação visual.

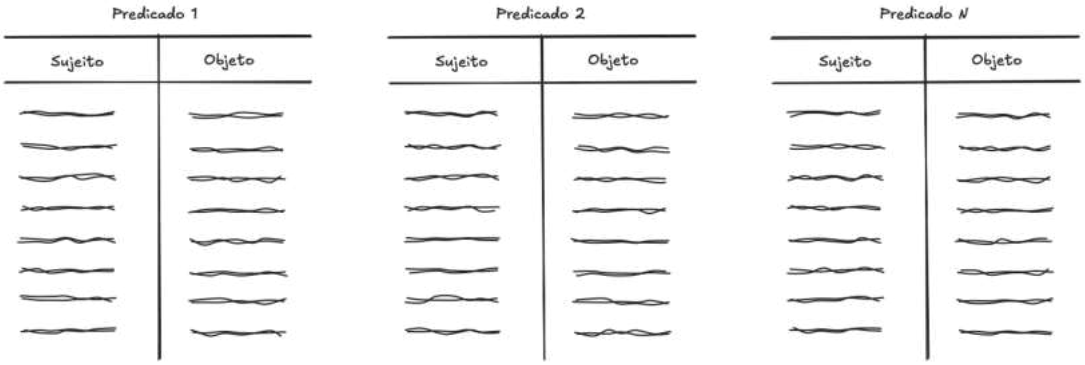
Diferente das abordagens anteriores de tabela única, a abordagem de particionamento vertical da tabela de propriedades (*Property Table Vertical Partitioning*) é composta de múltiplas tabelas representando cada predicado presente no grafo. Por sua vez, cada tabela é composta de apenas duas colunas: a de sujeito e a de objeto (ABADI et al., 2007), conforme mostra a figura 12. Esta, apesar de ser composta de múltiplas tabelas, ainda não extrai a classe para ser visualizada na interface da plataforma, apenas os predicados.

Figura 11 – Tabela de Propriedades Estendida (*Wide Property Table*)



Fonte: Elaborado pelo autor

Figura 12 – Particionamento Vertical da Tabela de Propriedades (*Property Table Vertical Partitioning*)

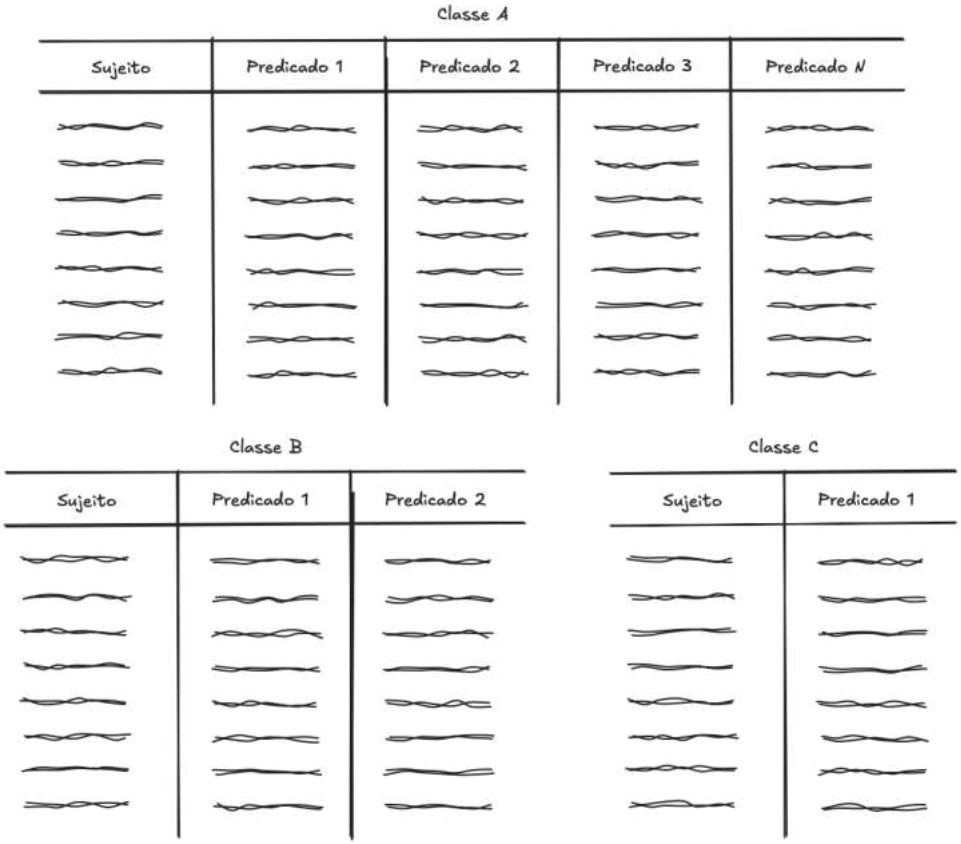


Fonte: Elaborado pelo autor

Proposta para o Apache Jena 2 (WILKINSON et al., 2003), a tabela de propriedades particionada por classes (*Property Class Table*) se mostra uma ideia promissora para este trabalho. Ela é caracterizada por conter todos os predicados relacionados ao sujeito na tabela, semelhante à tabela de propriedades estendida, porém com uma diferença única: é particionada por classes, conforme mostra a figura 13.

Dessa forma, a tabela de propriedades particionada por classes garante que a representação visual adotada será por classes, como tabelas, e predicados como colunas das tabelas, em complemento à coluna de sujeito. Esta abordagem será adotada neste trabalho para o mapeamento entre grafos RDF e a plataforma de BI, sem armazenar dados em formato tabular ou realizar transformações relacionais. As ideias do mapeamento servirão apenas de insumo para a representação visual dos elementos do grafo na ferramenta de BI.

Figura 13 – Tabela de Propriedades Particionada por Classes (*Property Class Table*)



Fonte: Elaborado pelo autor

3 TRABALHOS RELACIONADOS

A integração entre dados no formato RDF e representações gráficas tem sido explorada por poucas ferramentas. A maioria dessas soluções adota uma abordagem web, permitindo que o usuário interaja por meio de um navegador, o que facilita a criação de visualizações mesmo sem conhecimentos avançados em programação. Considerando que as plataformas de BI também possuem uma interface web, esse fator foi decisivo na seleção das ferramentas analisadas. Neste capítulo, são apresentados trabalhos que buscam gerar gráficos a partir de dados consultados via SPARQL, analisando suas contribuições e as lacunas que justificam a proposta deste trabalho.

3.1 D3SPARQL

O D3SPARQL[1] é uma biblioteca JavaScript voltada para a visualização de resultados de consultas SPARQL (KATAYAMA, 2014) por meio da biblioteca de geração de gráficos em páginas web, D3.js[2]. A partir de uma consulta SPARQL executada em um *endpoint*, a ferramenta transforma o resultado obtido em formato JSON, via chamada AJAX (*Asynchronous JavaScript and XML*), e o converte em imagens vetoriais SVG[3] (*Scalable Vector Graphics*), permitindo criar gráficos interativos diretamente em páginas web, como mostra a figura 14.

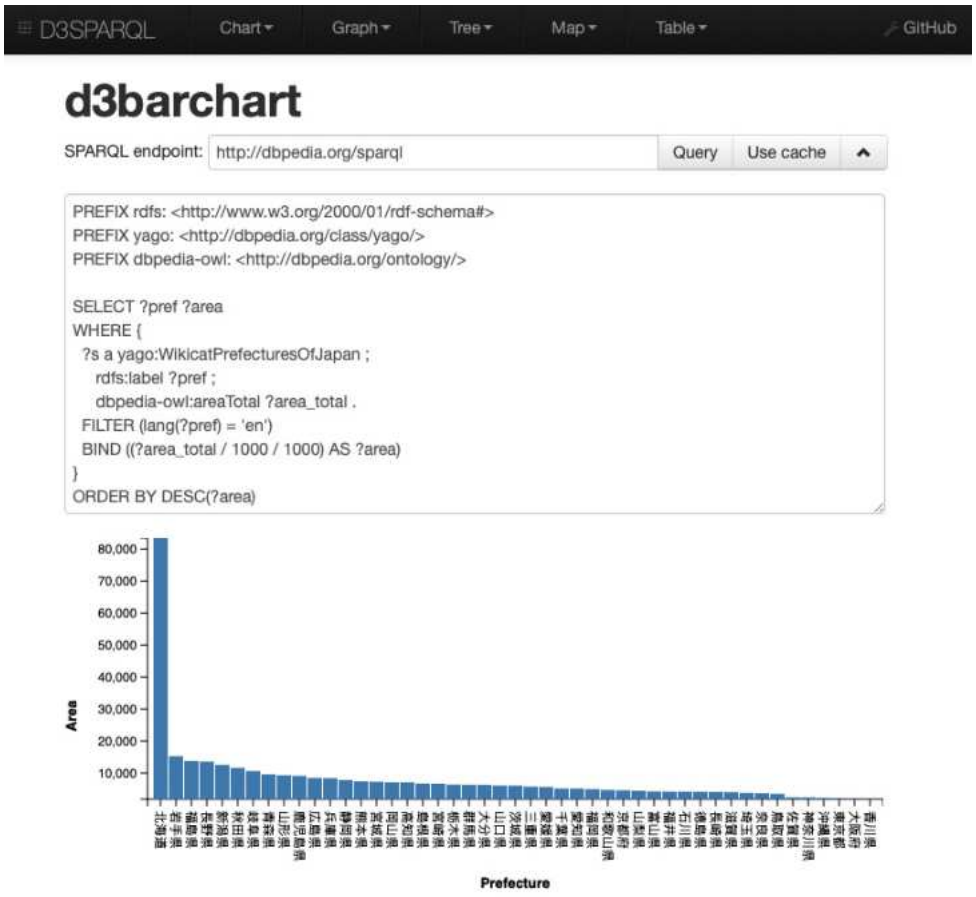
A biblioteca oferece suporte aos seguintes tipos de visualização: gráficos de barras, de pizza e de dispersão; grafos do tipo *force* e *Sankey*; visualizações hierárquicas como dendrograma, *treemap*, *sunburst* e *circle packing*; além de mapas geográficos e tabelas HTML. A figura 14 mostra um gráfico de barras gerado pela ferramenta e ilustra um dos tipos de visualização que o *driver* deve viabilizar.

Apesar das possibilidades que uma biblioteca JavaScript proporciona, seu uso pode não ser intuitivo para analistas sem experiência em desenvolvimento web. Na construção de *dashboards*, é esperada uma interface que facilite a criação dessas visualizações sem exigir conhecimentos profundos de programação.

De forma geral, o D3SPARQL se mostra eficiente para protótipos de aplicações para usuários com conhecimento em desenvolvimento web, porém, ainda pode ser de difícil adoção devido à sua carga operacional de gerenciamento. Por se tratar de uma biblioteca JavaScript, o D3SPARQL não provê elementos típicos de uma plataforma de BI, como salvar consultas e gráficos e criar *dashboards* interativos.

[1] https://github.com/ktym/d3sparql
[2] https://d3js.org
[3] https://pt.wikipedia.org/wiki/SVG

Figura 14 – Gráfico de barras gerado pelo D3SPARQL



Fonte: https://biohackathon.org/d3sparql/ selecionando o menu "d3barchart" e clicando em "Use cache"

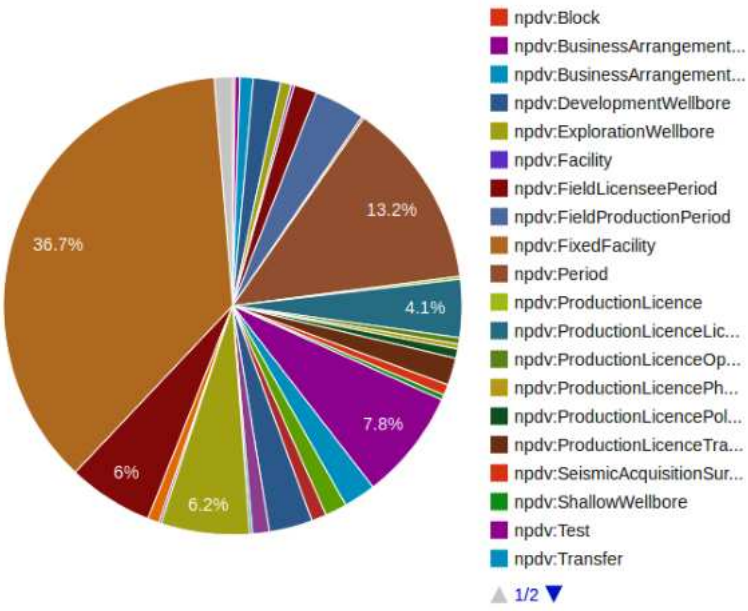
3.2 SGVIZLER

Outra ferramenta que se assemelha ao D3SPARQL é o Sgvizler[4]. O mecanismo por trás dela é servir de ponte entre consultas SPARQL e gráficos para a Web. Ela propõe que os usuários escrevam menos código em JavaScript, como uma camada de abstração; basta definir a consulta e escolher o tipo de gráfico (SKJÆVELAND, 2015).

A biblioteca Sgvizler lê valores e definições colocados em atributos HTML, envia a consulta via AJAX para o *endpoint* e espera o retorno em JSON. Assim que a resposta é obtida, transforma tudo em um objeto *DataTable* do Google Charts[5] e, em seguida, o gráfico é renderizado no lugar onde o elemento HTML foi inserido. A vantagem disso é que ela utiliza o Google Charts como gerador de gráficos, então sua única responsabilidade é transformar o JSON no formato aceito pela ferramenta final. A figura 15 mostra um exemplo gerado pela biblioteca; ela consiste em um gráfico de pizza da distribuição de instâncias por classe no grafo RDF.

[4]https://github.com/mgskjaeveland/sgvizler
[5]https://developers.google.com/chart/interactive/docs/datatables_dataviews?hl=pt-br

Figura 15 – Gráfico de pizza utilizando o Sgvizler



Fonte: https://github.com/mgskjaeveland/sgvizler

Mesmo assim, ainda há obstáculos para quem não domina HTML, CSS ou JavaScript: inserir classes, ajustar atributos, lidar com erros da requisição em páginas Web. Mais uma vez, isso é pouco intuitivo para analistas sem experiência em desenvolvimento web.

3.3 LDVIZ

O Linked Data Visualizer, ou simplesmente LDViz, foi proposto por Menin et al. (2023) e segue o padrão de visualização de dados composto basicamente por cinco etapas. O fluxo dos dados, desde o *endpoint* SPARQL até a visualização final, é processado por um conjunto de *softwares* que se complementam na execução de cada etapa.

1. Importação dos dados: Os dados são importados do *endpoint* SPARQL e convertidos para um formato JSON. Uma interface é disponibilizada para que o usuário possa realizar uma análise exploratória, como representada na figura 16.

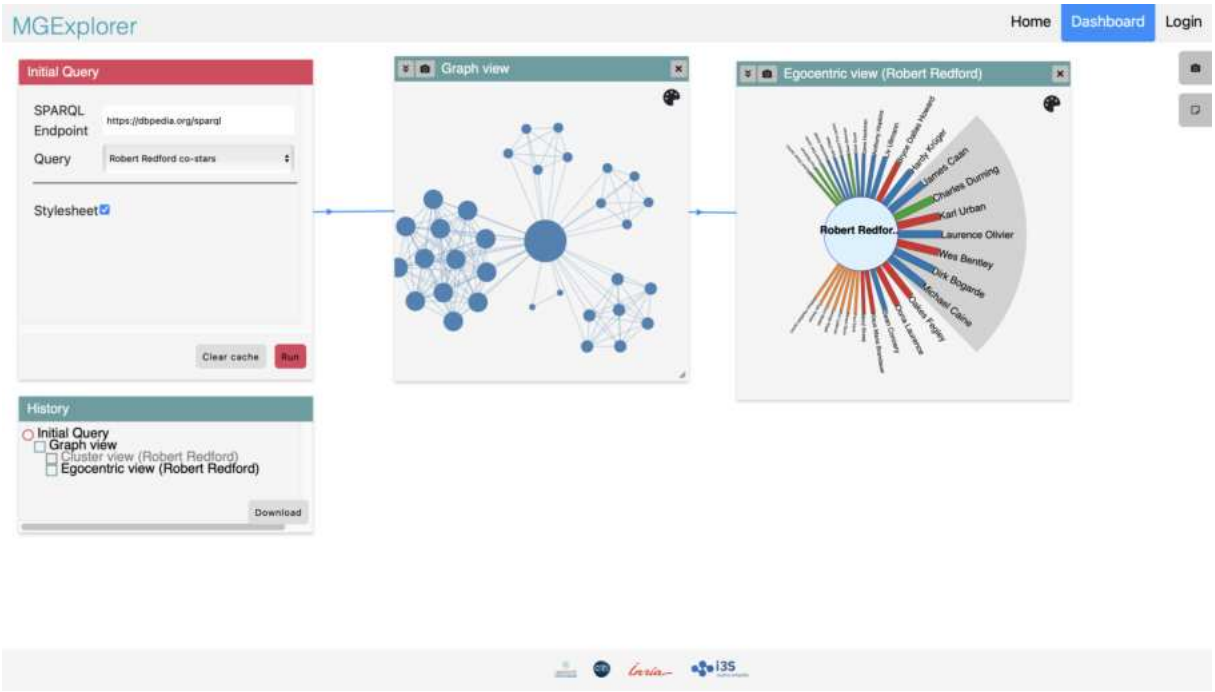
2. Transformação: Nessa etapa ocorre a modelagem dos dados para a estrutura que a ferramenta em questão espera.

3. Mapeamento: Aqui as variáveis do modelo de dados próprio do LDViz são mapeadas para o uso na visualização final.

4. Renderização: O software MGExplorer é usado para exibir o gráfico final utilizando a biblioteca JavaScript *D3.js*.

5. Interação: Por último, a ferramenta permite que o usuário interaja com o gráfico final.

Figura 16 – Interface do LDViz/MGExplorer



Fonte: https://dataviz.i3s.unice.fr/mgexplorer/dashboard

O LDViz se mostra eficiente na exploração de grafos, porém apresenta geração limitada de gráficos e não permite a criação de *dashboards*. Isso faz com que o LDViz não atinja os objetivos propostos neste trabalho.

É importante notar que, nos testes realizados com os *endpoints* SPARQL pelo LDViz, foi obtida uma taxa de sucesso de conexão e retorno de um resultado validado de aproximadamente 42%, ou seja, menos da metade dos 419 *endpoints* SPARQL que foram testados (MENIN et al., 2023). Isso indica que ainda é necessário cuidado adicional nas etapas de conexão do *driver* e na validação de funcionalidades de cada *endpoint* SPARQL. Esse aspecto é aprofundado na discussão da implementação do *driver*.

3.4 LINKDAVIZ

A abordagem do LinkDaViz é diferente das demais por propor uma ontologia específica para visualização de dados (THELLMANN et al., 2015). A ideia por trás da ontologia é reduzir a carga cognitiva do usuário em escolher o tipo de visualização e a forma como os dados serão representados e deixar essa decisão para o próprio algoritmo de recomendação de visualização.

O fluxo de trabalho proposto é baseado em três fases: a primeira de exploração e seleção dos dados é onde o usuário navega pelo grafo RDF através de uma árvore de classes e propriedades, selecionando o que deseja visualizar. A segunda fase envolve a análise do tipo de visualização que é recomendado ao usuário, baseado em uma ontologia específica para visualização de dados. Já a terceira fase consiste na geração do gráfico por parte da ferramenta de acordo com as recomendações do algoritmo e as preferências do usuário. Ao final, o usuário pode exportar o gráfico gerado para um formato de imagem, como PNG (*Portable Network Graphics*).

A aplicação desenvolvida se assemelha às demais no lado do cliente: é baseada em uma abordagem web utilizando a linguagem JavaScript e a biblioteca D3 para visualização. Diferentemente das demais, ela possui uma parte sendo executada no lado do servidor, onde são processadas as estatísticas e recomendações do algoritmo.

Infelizmente, o link reportado no artigo (THELLMANN et al., 2015) não está mais disponível na internet e não foi possível verificar o funcionamento da ferramenta. Isso reforça a necessidade de disponibilizar publicamente o *driver* resultante deste trabalho, para que estudos e trabalhos futuros possam fazer uso do artefato.

3.5 CONSIDERAÇÕES

A análise dos trabalhos relacionados mostra um cenário em que a integração entre dados RDF e visualizações permanece em sua maioria restrita a soluções de *baixo nível* voltadas a desenvolvedores web. Bibliotecas como D3SPARQL e Sgvizler reduzem o atrito na geração de gráficos a partir de consultas SPARQL, mas ainda exigem manipulação de HTML/JavaScript e desenvolvimento de páginas web. Ferramentas como o LDViz oferecem um processo mais guiado de exploração, porém apresentam tipos de gráficos limitados e não oferecem suporte a *dashboards*. Já o LinkDaViz propõe uma mudança no paradigma ao empregar uma ontologia para recomendar visualizações, mas exige trabalho do analista para alinhar os dados à ontologia de visualização. Além disso, sua indisponibilidade pública inviabiliza a reprodução e uma avaliação mais aprofundada.

Do ponto de vista prático, a ausência de recursos típicos de plataformas de BI se destaca como uma limitação: persistência de consultas e gráficos, construção de *dashboards*, parametrização, compartilhamento e controle de acesso. A tabela 3 resume comparativamente os trabalhos analisados.

Em resumo, as soluções existentes são relevantes para os cenários a que se propõem, mas não entregam, de forma integral, a experiência esperada por analistas em plataformas de BI. Este comparativo fundamenta e justifica a estratégia adotada: utilizar uma ferramenta de BI já existente e amplamente difundida como base para desenvolver um *driver* SPARQL. O próximo capítulo explora as ferramentas de BI *open-source* disponíveis e justifica a escolha da plataforma.

Tabela 3 – Comparativo dos trabalhos analisados

Ferramenta	**Abordagem**	**Exigência de programação**	**Dashboards**	**Observações**
D3SPARQL	HTML + biblioteca JavaScript + D3	Sim	Não	Flexível; sem persistência nativa de consultas/gráficos.
Sgvizler	HTML + biblioteca JavaScript + Google Charts	Sim	Não	Abstrai JSON → DataTable; depende do ecossistema Google Charts.
LDViz	Fluxo (importar/transformar/mapear/-renderizar)	Sim	Não	Foco em exploração; repertório de gráficos limitado.
LinkDaViz	Ontologia de recomendação + HTML + biblioteca JavaScript + D3	Sim	Não	Assistência automática de visualização; protótipo indisponível publicamente.

Fonte: Elaborado pelo autor.

4 ESCOLHA DA PLATAFORMA DE BI

Este capítulo apresenta a escolha da plataforma de BI utilizada como base para a implementação do *driver* SPARQL. Começa com uma revisão das plataformas de BI *open-source* disponíveis nos repositórios do GitHub, destacando as principais características e funcionalidades de cada uma. Em seguida, apresenta o processo de seleção, que envolve a análise de diversos critérios, como suporte a linguagens de consulta, gama de gráficos disponíveis, usabilidade e possibilidade de expansão para novas fontes de dados. Por fim, são discutidas as decisões tomadas para a escolha do Metabase como a ferramenta de BI e detalhes sobre a plataforma.

4.1 PLATAFORMAS DE BI OPEN-SOURCE

A proposta deste trabalho é utilizar uma ferramenta *open-source*, portanto, a fonte de pesquisa foi o GitHub[1], um repositório git amplamente conhecido pela comunidade de desenvolvedores e simpatizantes da cultura de código livre e aberto. Ele armazena uma ampla variedade de códigos-fonte divididos em repositórios, que atendem aos mais diversos objetivos.

Cada repositório pode atribuir tópicos, para tornar o código melhor categorizado dentro do site. Portanto, foi utilizada pesquisa por tópicos para filtrar os repositórios que têm relação com o objetivo deste trabalho. Os tópicos pesquisados estão em inglês, seguindo uma convenção da comunidade, e, para a pesquisa, foram usados os seguintes termos:

- *dashboard*
- *dataviz*
- *data-visualization*
- *business-intelligence*

Além dos 4 tópicos citados, são excluídos os repositórios privados; são mantidos apenas repositórios com mais de dez mil estrelas, ou seja, bem conceituados dentro do GitHub e pela comunidade; que não estejam arquivados, ou seja, seu desenvolvimento ainda está ativo; e limita a busca aos primeiros 20 repositórios ordenados em ordem decrescente pelo número de estrelas. A ferramenta de linha de comando[2] do próprio GitHub é utilizada para realizar a pesquisa na plataforma.

[1] https://github.com
[2] https://cli.github.com

Código 6 – Comando do GitHub CLI para busca de repositórios relevantes

```
gh search repos \
  --topic "dashboard OR dataviz OR data-visualization OR
     business-intelligence" \
  --visibility public \
  --stars ">=10000" \
  --archived=false \
  --order desc \
  --sort stars \
  --limit 20 \
  --json fullName,description,stargazersCount \
  --template '{{range
     .}}{{.fullName}},"{{.description}}",{{.stargazersCount}}{{printf
     "\n"}}{{end}}'
```

Por meio da execução no terminal do código 6 é possível obter os 20 repositórios mais relevantes para os tópicos pesquisados, resultando na tabela 4. Na tabela, há três destaques, correspondentes aos repositórios selecionados através de uma análise qualitativa e manual para filtrar o que faz sentido ao trabalho. É importante observar que os demais repositórios não têm relação com o objetivo aqui exposto, então são ignorados.

O primeiro destaque é para o Apache Superset[3], uma ferramenta *open-source* de visualização e exploração de dados. Ela suporta integrações com diversas fontes, como PostgreSQL, MySQL e Snowflake. Suas opções de gráficos são diversificadas, oferecendo gráficos comuns, como de pizza e de barras, até gráficos mais elaborados, como de dispersão ou histograma. A não escolha do Superset nesta implementação se deve ao uso interno da biblioteca SQLAlchemy[4], que limita as consultas a bancos de dados SQL, dificultando a implementação de conectores para fontes que não utilizam SQL, como endpoints SPARQL.

O Redash[5], embora não documentado oficialmente em sua página[6] até a data de 24 de julho de 2025, oferece um *query runner*, nome dado aos componentes que se conectam ao banco de dados, para SPARQL. Porém, por mais que o Redash seja a ferramenta mais antiga entre os três destaques, a figura 3 mostra que ela vem declinando sua taxa de crescimento de popularidade em comparação com o Apache Superset e o Metabase. Além disso, ela oferece uma menor gama de gráficos (13 tipos[7] contra 18 do Metabase) e não possui uma interface que auxilie a construção visual de consultas.

Por fim, o Metabase, também de código aberto e voltado para exploração e visualizações de dados tabulares, possui uma rica gama de gráficos e uma interface amigável para

[3]https://github.com/apache/superset
[4]https://www.sqlalchemy.org
[5]https://github.com/getredash/redash
[6]https://redash.io/help/data-sources/querying/supported-data-sources/
[7]https://redash.io/help/user-guide/visualizations/visualization-types/

Tabela 4 – Repositórios GitHub relacionados à visualização de dados e dashboards.

Nome	**Descrição do Repositório no GitHub**	**Estrelas**
d3/d3	Bring data to life with SVG, Canvas and HTML.	110921
PanJiaChen/vue-element-admin	A magical vue admin	89505
netdata/netdata	The fastest path to AI-powered full stack observability, even for lean teams.	74874
grafana/grafana	The open and composable observability and data visualization platform. Visualize metrics, logs, and traces from multiple sources like Prometheus, Loki, Elasticsearch, InfluxDB, Postgres and many more.	68654
strapi/strapi	Strapi is the leading open-source headless CMS. It's 100% JavaScript/TypeScript, fully customizable, and developer-first.	67120
apache/superset	Apache Superset is a Data Visualization and Data Exploration Platform	66766
apache/echarts	Apache ECharts is a powerful, interactive charting and data visualization library for browser	63812
pi-hole/pi-hole	A black hole for Internet advertisements	52336
pixijs/pixijs	The HTML5 Creation Engine: Create beautiful digital content with the fastest, most flexible 2D WebGL renderer.	45289
metabase/metabase	The easy-to-use open-source Business Intelligence and Embedded Analytics tool that lets everyone work with data	42450
streamlit/streamlit	Streamlit - A faster way to build and share data apps.	40065
tabler/tabler	Tabler is free and open-source HTML Dashboard UI Kit built on Bootstrap	39617
gradio-app/gradio	Build and share delightful machine learning apps, all in Python. Star to support our work!	38730
ant-design/ant-design-pro	Use Ant Design like a Pro!	37230
directus/directus	The flexible backend for all your projects Turn your DB into a headless CMS, admin panels, or apps with a custom UI, instant APIs, auth & more.	30981
microsoft/Data-Science-For-Beginners	10 Weeks, 20 Lessons, Data Science for All!	29789
getredash/redash	Make Your Company Data Driven. Connect to any data source, easily visualize, dashboard and share your data.	27459
academic/awesome-datascience	An awesome Data Science repository to learn and apply for real world problems.	26674
akveo/ngx-admin	Customizable admin dashboard template based on Angular 10+	25537
glanceapp/glance	A self-hosted dashboard that puts all your feeds in one place	25429

Fonte: Elaborado pelo autor com base no resultado do código 6. Dados coletados em 30/06/2025.

criar consultas de maneira visual, atenuando a curva de aprendizado. Sua arquitetura modular permite que a comunidade contribua, criando os mais variados tipos de *drivers*, que são componentes especializados em se conectar com os bancos de dados externos à aplicação. Devido a essa característica modular dos *drivers* e sua ampla gama de tipos de gráficos disponíveis, essa ferramenta foi escolhida como base desta implementação.

4.2 METABASE

Segundo a própria documentação do Metabase (METABASE, 2025):

> *Metabase is a "Business intelligence" (BI) platform that gives you a bunch of tools to understand and share your data. Companies typically use Metabase to give their teams an easy way to query data, or to embed Metabase in their application to let customers explore data on their own.*

Ou seja, é uma plataforma de BI capaz de capacitar analistas e até usuários sem conhecimento de linguagens de consulta, como o SQL, a obter valor a partir de seus dados.

Apesar de o foco em fontes relacionais, o Metabase não se limita ao SQL. Por exemplo, os próprios desenvolvedores da plataforma suportam o *driver* do MongoDB[8], um banco de dados NoSQL. Isso é um indício da viabilidade da proposta aqui apresentada, que busca executar consultas utilizando a linguagem SPARQL.

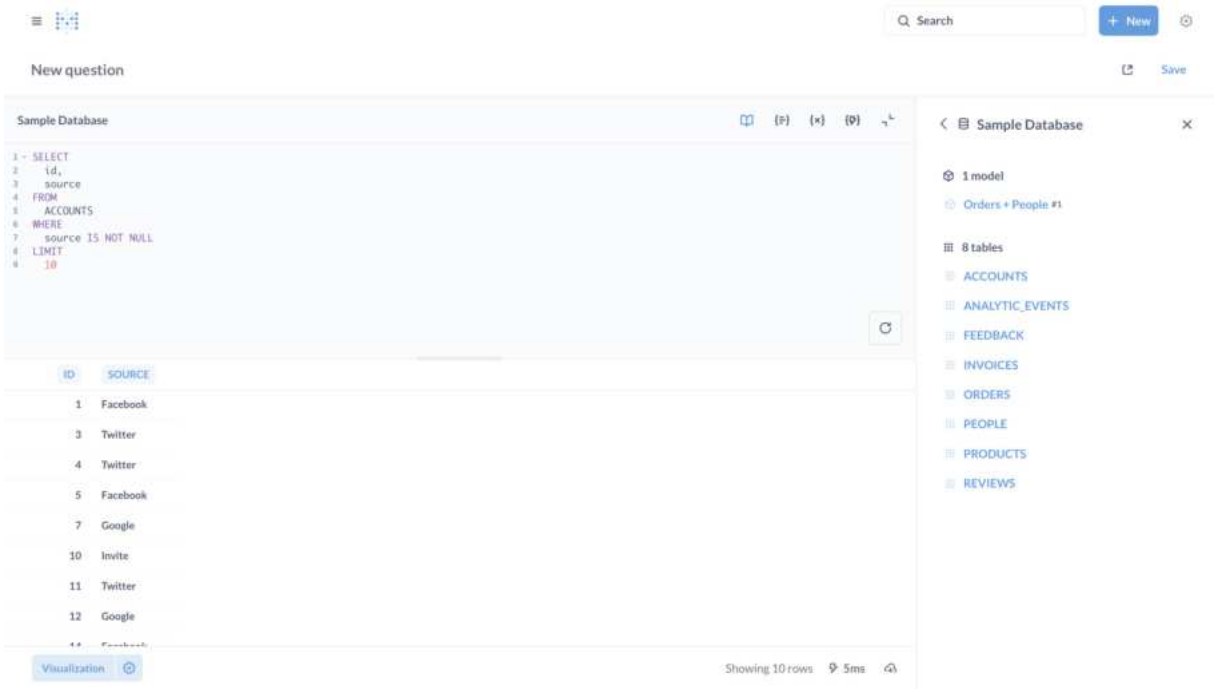
A interface de consultas do Metabase está representada na figura 17. Nela, há um banco de dados de amostra que acompanha a instalação básica. Na lateral direita da imagem, é possível observar uma listagem de tabelas presentes na fonte. Na parte central, em cinza claro, está localizada a caixa de texto onde o usuário pode inserir a consulta que será executada. E na parte inferior, temos o resultado da consulta em formato tabular, ou seja, dividido em linhas e colunas.

O formato do dado representado em cada coluna é obtido através dos metadados do resultado da consulta processada pelo *driver* em questão. Por exemplo, se uma coluna é do tipo data no esquema da tabela, então é mapeada para o tipo data do Metabase, assim como pode acontecer com o tipo numérico ou textual. Portanto, é de responsabilidade do *driver* fazer esse mapeamento de cada tipo da fonte de dados para o tipo correspondente no Metabase.

A parte visual é bastante rica em possibilidades de visualização. Na figura 18 são apresentados os tipos de gráficos suportados pela ferramenta, sendo estes compostos de mais de 18 tipos de visualizações. No capítulo 6, são exemplificadas algumas dessas visualizações, juntamente com as respectivas consultas na linguagem SPARQL.

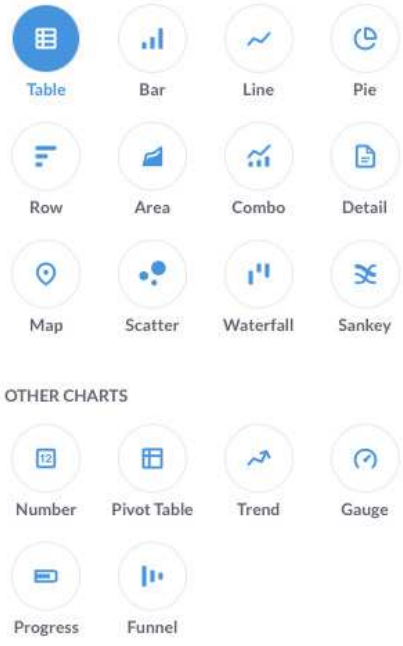
[8]https://www.mongodb.com

Figura 17 – Interface de consulta do Metabase



Fonte: Captura de tela feita pelo autor

Figura 18 – Tipos de visualizações suportadas pelo Metabase



Fonte: Captura de tela feita pelo autor

4.2.1 Drivers suportados

É possível dividir em dois grupos os *drivers* implementados para uso do Metabase: os oficiais e os desenvolvidos pela comunidade. A estrutura de interfaces implementada na linguagem Clojure permite que desenvolvedores criem um *driver* específico para se comunicar com alguma fonte de dados de interesse. Esses pequenos componentes podem ser empacotados em artefatos JAR, distribuídos e instalados em qualquer instância do Metabase de maneira muito simples.

Tabela 5 – *Drivers* mantidos pelo repositório oficial do Metabase

Driver	**Linguagem de Consulta**
Amazon Athena	SQL
BigQuery	SQL
ClickHouse	SQL
Databricks	SQL
Druid	SQL
Druid JDBC	SQL
Spark SQL	SQL
MongoDB	MQL
Oracle	SQL
Presto	SQL
Amazon Redshift	SQL
Snowflake	SQL
SQLite	SQL
SQL Server	SQL
Starburst (Trino)	SQL
Vertica	SQL

Fonte: https://github.com/metabase/metabase. Listas consultadas em 24/07/2025.

Como mencionado anteriormente, há um destaque para o *driver* do MongoDB na tabela 5. É relevante para o trabalho evidenciar que este é o único *driver* mantido oficialmente pelo Metabase que não utiliza o SQL como linguagem de consulta, e sim a linguagem própria do MongoDB, a *MongoDB Query Language* (MQL).

Outro destaque encontra-se na tabela 6. O Neo4j[9] é um banco de dados baseado em grafos de propriedades e utiliza a Cypher como linguagem de consulta. Apesar de ele utilizar uma biblioteca Java que conecta ao Neo4j via JDBC, é evidenciada a viabilidade técnica da utilização de banco de dados não-relacionais (especificamente, grafos de conhecimento).

A forma como o Metabase é estruturado permite à comunidade implementar abstrações comuns dos *drivers* que são necessárias para o funcionamento correto da plataforma. Essa

[9]https://neo4j.com

Tabela 6 – *Drivers* mantidos pela comunidade *open-source*

Driver	**Linguagem de Consulta**	**Repositório no GitHub**
CSV	Não se aplica	Markenson/csv-metabase-driver
Databend	SQL	databendcloud/metabase-databend-driver
IBM i	SQL	damienchambe/metabase-ibmi-driver
Dremio	SQL	Baoqi/metabase-dremio-driver
DuckDB	SQL	motherduckdb/metabase_duckdb_driver
Firebolt	SQL	firebolt-db/metabase-firebolt-driver
Firebird	SQL	evosec/metabase-firebird-driver
GreptimeDB	SQL	GreptimeTeam/greptimedb-metabase-driver
Impala	SQL	brenoae/metabase-impala-driver
Materialize	SQL	MaterializeInc/metabase-materialize-driver
Neo4j	Cypher	StronkMan/metabase-neo4j-driver
NetSuite	SQL	ericcj/metabase-netsuite-driver
Peaka	SQL	peakacom/metabase-driver
Teradata	SQL	swisscom-bigdata/metabase-teradata-driver

Fonte: https://www.metabase.com/docs/v0.55/developers-guide/community-drivers - Listas consultadas em 24/07/2025

forma de abstração é graças a um conceito chamado de multimétodo[10], uma maneira elegante de utilizar polimorfismo na linguagem de programação Clojure.

O polimorfismo é uma maneira pela qual determinadas funções podem responder à mesma interface de acordo com os argumentos e tipos de dados recebidos como entrada. Por exemplo, o multimétodo `can-connect?` que o Metabase define impõe que os *drivers* implementem a própria lógica de testar a conexão com o banco de dados.

No exemplo do código 7 é possível verificar a chamada ao multimétodo específico de cada *driver* e a implementação de dois *drivers* distintos, um do SQLite e outro do MongoDB. De acordo com o banco de dados que o usuário estiver consultando no momento, a respectiva implementação é chamada.

[10]https://clojure.org/reference/multimethods

Código 7 – Multimétodos do SQLite e MongoDB e a chamada genérica

```
(defn can-connect-with-details?
 ^Boolean [driver details-map & [throw-exceptions]]
 {:pre [(keyword? driver) (map? details-map)]}
 (if throw-exceptions
  (try
   (u/with-timeout (driver.settings/db-connection-timeout-ms)
    (or (driver/can-connect? driver details-map) ;; <--- Chamada do
        multimétodo
...)
;; Implementação do multimétodo para sqlite
(defmethod driver/can-connect? :sqlite [driver details]
 (if (confirm-file-is-sqlite (:db details))
  (sql-jdbc.conn/can-connect? driver details) false))
...)
;; Implementação do multimétodo para mongo
(defmethod driver/can-connect? :mongo [_ db-details]
 (mongo.connection/with-mongo-client [^MongoClient c db-details]
  (let [db-names (mongo.util/list-database-names c)
...)
```

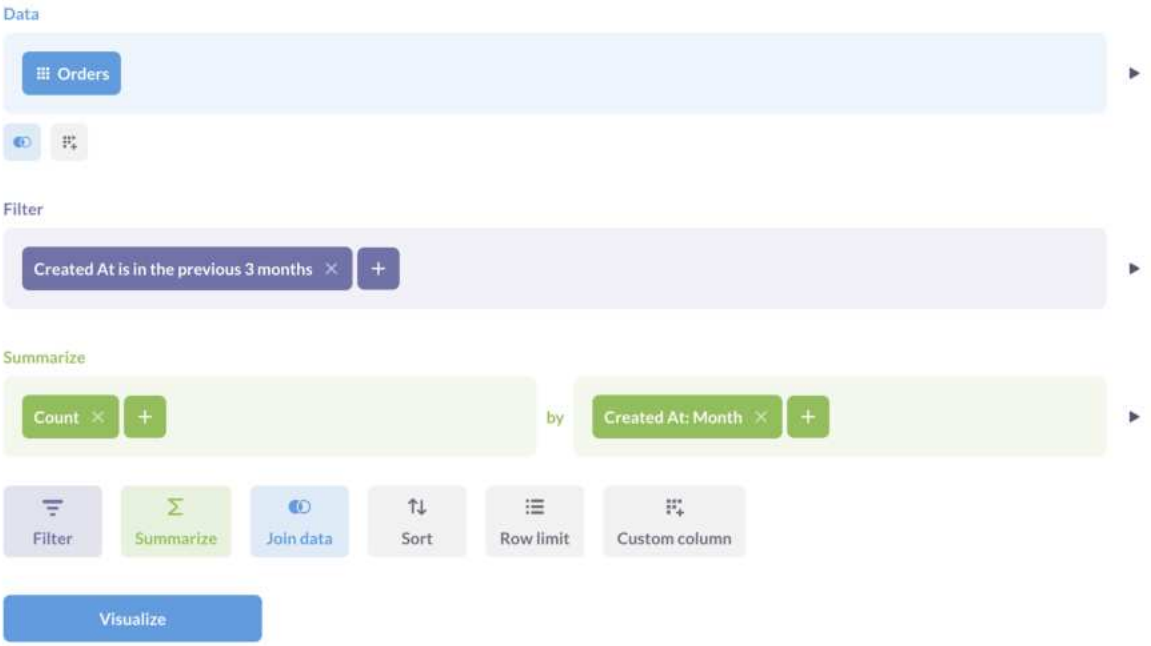
Essa abordagem na arquitetura oferece várias vantagens, como: facilitar a implementação de novos *drivers* sem modificar o código existente da plataforma; separar claramente a responsabilidade de cada implementação; e manter a consistência da interface, fazendo com que todos os *drivers* tenham que implementar os mesmos métodos. Dessa forma, os multimétodos são essenciais para manter a modularidade e a escalabilidade do Metabase.

Um dos principais multimétodos a ser implementado é o relacionado à construção visual de consultas. A plataforma possui a funcionalidade em que o usuário pode construir consultas por meio de blocos interativos em uma interface web. Essa funcionalidade é denominada *query builder* (figura 19), a qual será referenciada pelo seu nome em inglês quando citada. O *query builder* necessita do multimétodo `mbql->native` para converter a sintaxe do MBQL em uma consulta SPARQL.

4.2.2 Metabase Query Language

A MBQL, linguagem interna específica do Metabase para representar consultas, é de suma importância nesta implementação. Na figura 19 é possível observar a interface do *query builder*, que consiste em blocos visuais que geram uma consulta de acordo com o *driver* utilizado. Na maioria dos *drivers* já implementados é utilizada a conversão para SQL; no caso do MongoDB, gera MQL; nesta implementação, é esperada a geração da consulta SPARQL.

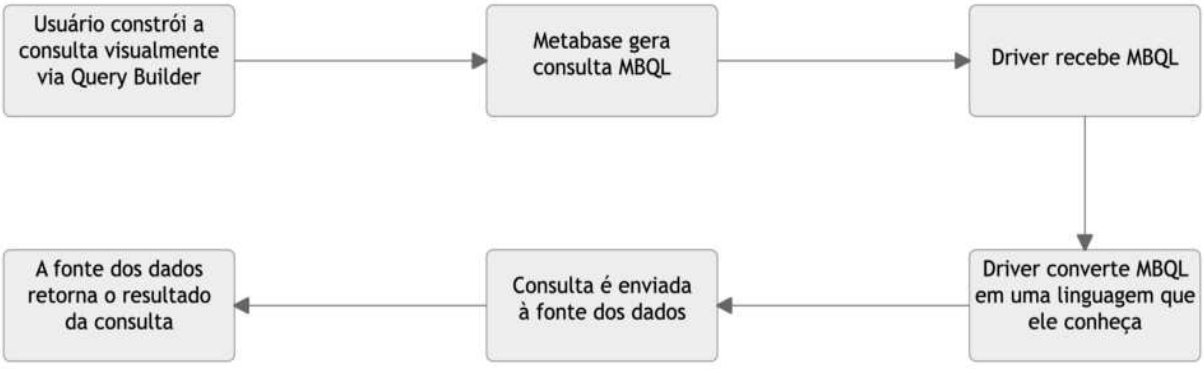
Figura 19 – Query Builder do Metabase



Fonte: https://www.metabase.com/docs/latest/questions/query-builder/editor

O *query builder* permite que usuários sem proficiência na linguagem de consulta em questão possam obter dados e construir visualizações de maneira intuitiva e visual. Essa consulta construída via interface deve ser convertida internamente pelo *driver* para a linguagem que a fonte de dados aceite e entenda, no caso deste trabalho será o SPARQL. O processo segue um fluxo muito bem definido: o usuário monta a consulta visualmente na interface, o Metabase estrutura a consulta em um formato interno da plataforma, MBQL, e a envia para o *driver* converter na linguagem apropriada, em seguida, o *driver* envia a consulta gerada para a fonte de dados e, por fim, a fonte de dados retorna os resultados esperados.

Figura 20 – Diagrama do fluxo do *Metabase Query Language* para um *driver* genérico



Fonte: Elaborado pelo autor

A maioria dos *drivers* do Metabase não implementa essa conversão diretamente, pois herda da implementação padrão dos métodos já existentes do SQL. Nesta implementação, tal aproveitamento não é possível, como se apresenta no capítulo 5; portanto, é necessária a implementação dessa conversão entre a estrutura do MBQL e o SPARQL. No próximo capítulo é apresentada a proposta de implementação do *driver* SPARQL para o Metabase.

5 PROPOSTA E IMPLEMENTAÇÃO DO DRIVER

Neste capítulo, são descritas as fases de um projeto de engenharia de *software* adaptadas à realidade deste trabalho. Foi adotada uma abordagem enxuta e flexível, adequada ao tamanho da equipe, ao cronograma e ao escopo. Como destaca (PRESSMAN; MAXIM, 2021, p. 143), "[o processo de software] não precisa estar completo desde o início, mas é preciso saber o seu objetivo antes de dar o primeiro passo". Portanto, a seguir apresenta-se o planejamento de cada etapa, ainda que ela represente apenas uma visão geral que guiará o desenvolvimento.

1. **Concepção** - Envolve a análise da viabilidade técnica e econômica, a definição de escopo e a análise de possíveis riscos que o projeto pode enfrentar durante seu desenvolvimento.

2. **Levantamento de requisitos** - Reunir as funcionalidades que o *software* deve oferecer, a descrição delas e a definição de prioridades.

3. **Critérios de aceitação** - Alinhar os critérios que o *software* deve atender para verificar se os requisitos estão sendo cumpridos.

4. **Planejamento** - Definição da metodologia adotada e cronograma, neste caso, um cronograma mais flexível baseado em entregas incrementais.

5. **Definição arquitetural** - Definição das interfaces e camadas de abstração que serão utilizadas no desenvolvimento do *driver*, acompanhadas de diagramas de sequência.

6. **Implementação** - Fase em que ocorre o desenvolvimento do *software*, revisão de código, documentação técnica, refatorações e testes. Geralmente, essa fase é composta por um ciclo, onde o *software*, após testado, volta ocasionalmente à etapa de desenvolvimento.

7. **Entrega** - Fase que envolve a entrega do *driver* de acordo com o que foi proposto no objetivo. O artefato JAR será publicado no repositório git público e disponibilizado para uso pela comunidade.

5.1 CONCEPÇÃO DO PROJETO

Esta é uma das fases mais importantes de um projeto de *software*, pois nela são definidos os critérios de aceitação e de viabilidade, juntamente com as partes interessadas no processo. Será discutida a viabilidade técnica, ou seja, se há tecnologia disponível e se é possível executar o que será planejado, assim como a viabilidade econômica, ou seja,

se o que for planejado cabe no orçamento disponível. Em seguida, o escopo do projeto será definido e delimitado de acordo com o que fará parte da entrega. E, por fim, serão apresentados os riscos que podem surgir e como eles podem ser mitigados.

5.1.1 Viabilidade Técnica

De um lado, temos o Metabase, uma plataforma de BI bem conhecida e amplamente utilizada na comunidade de dados. Como apresentado no capítulo 4, o Metabase permite o desenvolvimento de *drivers* personalizados de acordo com a fonte a ser consultada, ou seja, é possível criar novos *drivers* de acordo com as necessidades do projeto. Do outro lado, temos os *endpoints* SPARQL, que permitem consultas sobre dados em RDF.

Endpoints SPARQL recebem consultas via protocolos HTTP ou HTTPS (*Hypertext Transfer Protocol Secure*), que representam, respectivamente, comunicações inseguras e seguras. Embora a maior parte dos *drivers* desenvolvidos para o Metabase utilize SQL por meio do JDBC, existe um voltado para o MongoDB, que adota um modelo orientado a documentos e uma linguagem própria de consultas, distinta da relacional. Apesar das diferenças entre HTTP e o protocolo do MongoDB, é justamente essa variedade de abordagens que demonstra a viabilidade técnica de implementar um *driver* SPARQL independente do JDBC.

Outro *driver*, citado no capítulo 4, é da plataforma Neo4j que, apesar de utilizar o protocolo JDBC para conexão, utiliza uma linguagem de consulta diferente do SQL (Cypher). O Neo4j utiliza a linguagem Cypher para consulta sobre grafos de propriedades. Em ambos os *drivers* apresentados não existe o conceito de tabelas, o que indica um potencial para a implementação do *driver* SPARQL.

Apresentada no capítulo 2, a especificação sobre o retorno JSON de *endpoints* (HAWKE et al., 2013b) padroniza o formato de resposta do servidor SPARQL e define que o formato JSON pode ser utilizado como resposta da requisição HTTP. Isso facilita a integração com bibliotecas do Clojure especializadas em lidar com esse formato de dados, como a Cheshire.

Dados os fatos apresentados, é possível inferir que a viabilidade de construir um *driver* capaz de enviar consultas SPARQL e processar o retorno do servidor é tecnicamente possível. A próxima fase deverá analisar a viabilidade econômica do projeto.

5.1.2 Viabilidade Econômica

Do ponto de vista econômico, o desenvolvimento do *driver* apresenta diversas vantagens em relação a possíveis soluções comerciais. Por ser um projeto de código aberto, não há custos de licenciamento de *software* proprietário. O Metabase é distribuído sob a licença Affero General Public License, que permite o uso, modificação, estudo e distribuição do *software*, desde que o código-fonte de eventuais modificações também seja disponibi-

lizado. Essa exigência não representa um empecilho para o trabalho, pois a distribuição pública do *driver* está alinhada aos objetivos iniciais apresentados no capítulo 1.

Outra vantagem em relação a sistemas comerciais é a possibilidade de adaptar o *driver* às necessidades específicas dos usuários. Caso falte alguma funcionalidade, desenvolvedores podem modificá-lo para atender às expectativas.

Além disso, não há necessidade de recursos financeiros para o desenvolvimento, já que toda a estrutura pode ser construída usando *softwares* de código aberto. O principal recurso exigido é humano, ou seja, o tempo dedicado pelo desenvolvedor ao projeto. Como esta implementação está sendo feita como parte do trabalho de conclusão de curso descrito nesta monografia, o custo já está contemplado nas horas de dedicação a este trabalho.

Por fim, o *driver* também pode trazer benefícios econômicos, tanto diretos quanto indiretos, ao possibilitar que os usuários visualizem e extraiam informações valiosas de *dashboards* construídos com dados RDF. Embora seja difícil metrificar financeiramente esse retorno, o valor gerado, ainda que subjetivo, é relevante e não deve ser desconsiderado.

5.1.3 Escopo

O escopo do projeto é algo muito importante, pois ele define quais funcionalidades devem ser entregues dentro do cronograma disponível. Na seção 1.1 foram apresentados os objetivos deste trabalho e do artefato a ser entregue. De forma geral, o escopo consiste na criação de um *driver* que permita ao Metabase se conectar a *endpoints* SPARQL por meio do protocolo HTTP.

Alguns desses *endpoints* oferecem conexão segura via HTTPS/TLS (Transport Layer Security), porém com certificados vencidos ou inválidos, o que pode gerar problemas de conectividade entre o *driver* e o servidor. Por isso, o *driver* também precisa, quando configurado, permitir ignorar a validação do certificado TLS para possibilitar a execução de consultas mesmo em ambientes com HTTPS inválido.

No que se refere às funcionalidades, o *driver* oferece suporte apenas a consultas SPARQL dos tipos `SELECT` e `ASK`. A primeira retorna dados em formato tabular, enquanto a segunda fornece um valor booleano, indicando a existência ou não de determinada informação no grafo RDF sendo consultado. Já as consultas SPARQL dos tipos `CONSTRUCT` e `DESCRIBE` estão fora do escopo, pois não produzem dados tabulares adequados à geração de gráficos no Metabase.

Também não faz parte da implementação a autenticação em *endpoints* SPARQL protegidos, que podem exigir autenticação básica ou autenticação via *token*. Isso não impede que seja implementado em trabalhos futuros, mas não é um requisito do projeto neste momento.

5.1.4 Riscos

O risco do projeto é a falta de conhecimento aprofundado do desenvolvedor em Clojure, linguagem de programação funcional que foi utilizada no desenvolvimento do *driver*. Esse risco pode ser mitigado com estudos e pesquisas para aprofundar o conhecimento necessário, embora isso possa gerar atrasos na entrega do *driver*. O estudo da linguagem foi realizado durante o desenvolvimento. Portanto, é possível que, em um primeiro momento, o código não seja o mais performático, mas deve ser funcional conforme os testes apresentados no capítulo 6.

Outro fator de risco é o cronograma de desenvolvimento desta monografia. Apesar de flexível, o desenvolvimento do *driver* pode levar mais tempo do que o previsto, impactando a escrita do texto. Esse risco pode ser mitigado com uma definição clara dos requisitos mínimos e dos critérios de aceitação, garantindo que as funcionalidades essenciais sejam desenvolvidas de forma eficiente.

O mapeamento entre o modelo de dados RDF e o formato tabular pode apresentar elevada complexidade, principalmente em grafos de conhecimento que não seguem as recomendações do W3C, apresentadas no Capítulo 2. Além disso, a necessidade de compatibilidade com diferentes versões do SPARQL, como 1.0 e 1.1, aumenta a complexidade do desenvolvimento. Diversas funcionalidades podem não ser suportadas pelos *endpoints*, variando de servidor para servidor (BUIL-ARANDA et al., 2013), tornando necessário implementar um mecanismo eficiente de descoberta de capacidades.

Por fim, é importante ressaltar que o desenvolvimento do *driver* não se encerra com a entrega desta monografia, já que o repositório será público e aberto a contribuições futuras. Todos os códigos e testes apresentados deverão referenciar uma *tag* específica no repositório git[1], garantindo a rastreabilidade completa da linha do tempo de desenvolvimento.

5.2 REQUISITOS MÍNIMOS E CRITÉRIOS DE ACEITAÇÃO

Para garantir que os objetivos do projeto sejam atendidos na implementação do *driver*, foram definidos requisitos funcionais e não funcionais, apresentados na tabela 7. Requisitos funcionais são essenciais para o correto funcionamento do *software*, que, sem eles, o *driver* não cumpriria sua finalidade, como é o caso dos requisitos *R2*, *R3*, *R4*, *R5*, *R6*, *R7*, *R8*, *R9* e *R11*. Já os requisitos não funcionais dizem respeito à qualidade do sistema, abrangendo aspectos como desempenho, manutenção e documentação, e não impactam diretamente o funcionamento correto do *software*, como em *R1*, *R10* e *R12*.

Para cada requisito foi estabelecido um critério de aceitação que permite verificar se o que foi planejado está sendo cumprido. Os resultados da validação desses critérios serão apresentados em detalhe no capítulo 6.

[1] https://github.com/jhisse/metabase-sparql-driver/tree/v0.0.4

Tabela 7 – Requisitos e critérios de aceitação

[R1] Código-fonte público sob licença AGPLv3.	**[C1]** Repositório GitHub público com arquivo `LICENSE`.
[R2] Conectar-se aos *endpoints* SPARQL via HTTPS.	**[C2]** Configurar ao menos 1 servidor SPARQL via protocolo HTTPS. A conexão deve ser verificada com sucesso, independente da latência da rede.
[R3] Conectar-se aos *endpoints* SPARQL via HTTPS, ignorando a validação do certificado TLS.	**[C3]** Configurar ao menos 1 servidor SPARQL via protocolo seguro HTTPS com certificado inválido. A conexão deve ser verificada com uma consulta `ASK` retornando verdadeiro ou falso com sucesso, independente da latência da rede.
[R4] Descobrir metadados RDF como estrutura tabular.	**[C4]** Listar automaticamente as classes RDF mais frequentes como tabelas e suas propriedades como colunas, limitando a exibição das 10 classes mais utilizadas no *endpoint*.
[R5] Executar consultas do tipo `ASK`.	**[C5]** Executar ao menos 5 consultas do tipo `ASK` em 3 *endpoints* SPARQL.
[R6] Executar consultas do tipo `SELECT`.	**[C6]** Executar ao menos 5 consultas do tipo `SELECT` em 5 *endpoints* SPARQL, validando retorno de metadados corretos e dados estruturados sem que exceções de código sejam lançadas. As consultas devem ser documentadas para poderem ser reproduzidas no futuro.
[R7] Mapear resultados para a interface tabular do Metabase.	**[C7]** Após C6, converter automaticamente tipos XSD (*integer*, *float*, *boolean*, *date*, *datetime*, *IRI*) para tipos Metabase equivalentes, preservando a semântica dos dados e garantindo que nomes de colunas sejam preservados na interface do Metabase com os tipos mapeados corretos.
[R8] Suportar parâmetros nativos em consultas SPARQL.	**[C8]** Permitir parametrização de consultas com sintaxe `{{parametro}}` e substituição correta de valores fornecidos pelo usuário na interface.
[R9] Permitir criação de gráficos a partir dos resultados.	**[C9]** Criar ao menos 8 tipos diferentes de gráficos que o Metabase disponibiliza, incluindo: tabela, barras verticais, barras horizontais, rosca, linha, bolhas, número único e mapa, a partir de consultas em SPARQL.
[R10] Fluxo de integração contínua com testes unitários e de sintaxe.	**[C10]** A cada envio de código para a ramificação principal do git, um fluxo deve executar verificação de sintaxe/boas práticas e testes unitários, além da compilação do artefato; a publicação do JAR deve ocorrer apenas quando uma tag de versão for criada.
[R11] Distribuir o *driver* como JAR instalável.	**[C11]** Copiar `metabase-sparql-driver.jar` para `plugins/` e o Metabase não deve lançar exceções na inicialização.
[R12] Documentação de instalação e uso.	**[C12]** Entrega do *driver* em uma imagem *Docker* para execução em contêiner com o Metabase, acompanhada de documentação de uso.

Fonte: Elaborado pelo autor

Idealmente, os requisitos do *driver* deveriam ser definidos em conjunto com as partes interessadas, de modo a refletir diretamente as necessidades da comunidade que irá utilizar o *driver*. No entanto, como não houve tempo hábil para essa etapa, os requisitos foram derivados de três fontes principais: (i) funcionalidades mínimas para que o Metabase reconheça o *driver*; (ii) suporte essencial para o processamento e interpretação de consultas SPARQL e seus resultados; e (iii) boas práticas em projetos de código aberto, alinhadas às restrições de cronograma.

Por fim, a disponibilização de uma imagem *Docker*[2], tecnologia de conteinerização que encapsula aplicações com suas dependências, garante que qualquer avaliador possa executar os testes em um ambiente controlado e reproduzível. Ainda assim, alguns fatores externos fogem ao controle deste autor, como a disponibilidade e o desempenho de *endpoints* públicos. Por essa razão, métricas como latência foram excluídas do escopo de avaliação do *driver*.

5.3 PLANEJAMENTO

Embora seja de interesse que o projeto continue evoluindo após a entrega deste trabalho, é importante definir prazos para materializar uma entrega condizente com nossos objetivos previamente definidos. Esse prazo deve se adequar a seis meses de desenvolvimento do *driver* paralelamente à escrita deste trabalho.

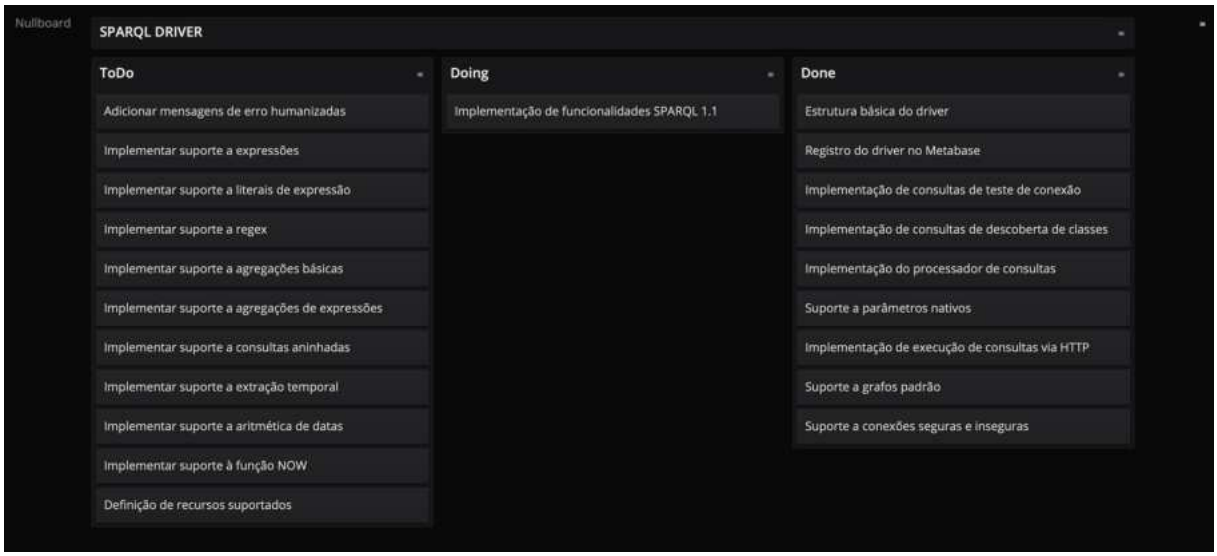
Foi definido o uso de um quadro Kanban simples, feito no *software* Nullboard[3], com três colunas: "*ToDo*", "*Doing*" e "*Done*", ou seja, o que é preciso fazer, o que está em fase de desenvolvimento e o que já foi feito. Esse quadro servirá como forma de organizar o desenvolvimento e acompanhar o que está pendente a ser feito.

A figura 21 mostra uma foto do quadro utilizado pelo autor em determinado momento do desenvolvimento. Nele, há tarefas pendentes a serem feitas na primeira coluna, tarefas que estavam sendo feitas, naquele período, na coluna central e tarefas que já foram concluídas na última coluna. É importante notar que este sistema é um fluxo orgânico em que novas tarefas podem ser incorporadas na primeira coluna a qualquer momento, de acordo com novas necessidades do projeto.

O uso do Kanban, neste projeto, segue os princípios fundamentais da metodologia ágil. Ele prioriza a visualização eficiente do fluxo de trabalho e limita as tarefas em progresso, facilitando o gerenciamento do trabalho e do tempo. Ao permitir que as atividades sejam acompanhadas e modificadas de forma dinâmica, o quadro Kanban facilita a organização das prioridades e a adaptação rápida a necessidades que ocorrem durante o desenvolvimento.

[2]https://www.docker.com
[3]https://github.com/apankrat/nullboard

Figura 21 – Quadro Kanban utilizado para o planejamento das tarefas



Fonte: Captura de tela feita pelo autor

5.4 DEFINIÇÃO ARQUITETURAL

Conforme apresentado anteriormente, a linguagem Clojure permite o uso de multimétodos para implementação de polimorfismo. O Metabase explora esse recurso para possibilitar a definição das funções necessárias ao funcionamento de cada *driver*. Grande parte desses métodos pode ser listada com o comando `clojure -M:run driver-methods docs`, executado no diretório do projeto do Metabase. A figura 22 apresenta um trecho da documentação resultante desse comando.

Figura 22 – Documentação dos multimétodos que podem ser implementados para o *driver*

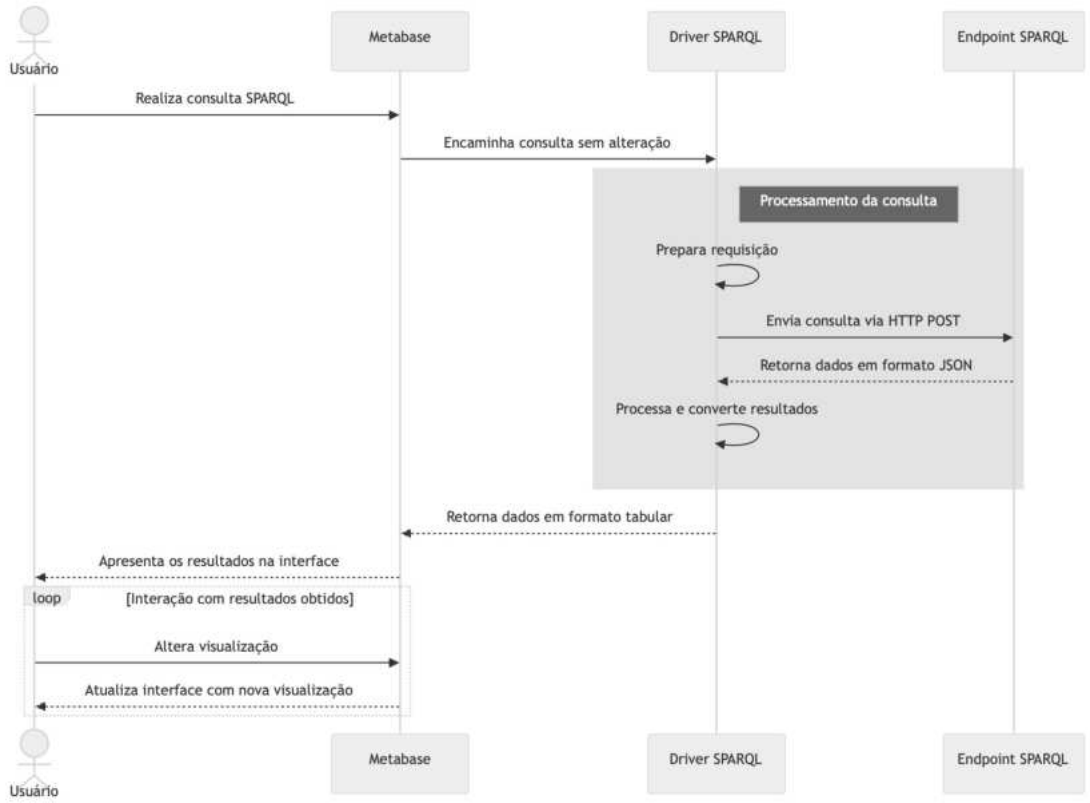
```
describe-database [driver database]
Return a map containing information that describes all of the tables in a `database`, an instance of the `Database`
  model. It is expected that this function will be peformant and avoid draining meaningful resources of the database.
  Results should match the [[metabase.sync.interface/DatabaseMetadata]] schema.

describe-fields [driver database & {:keys [schema-names table-names]}]
Returns a reducible collection of maps, each containing information about fields. It includes which keys are
  primary keys, but not foreign keys. It does not include nested fields (e.g. fields within a JSON column).

  Takes keyword arguments to narrow down the results to a set of
  `schema-names` or `table-names`.

  Results match [[metabase.sync.interface/FieldMetadataEntry]].
  Results are optionally filtered by `schema-names` and `table-names` provided.
  Results are ordered by `table-schema`, `table-name`, and `database-position` in ascending order.

describe-indexes [driver database & {:keys [schema-names table-names]}]
Returns a reducible collection of maps, each containing information about the indexes of a database.
  Currently we only sync single column indexes or the first column of a composite index. We currently only support
   indexes on unnested fields (i.e., where parent_id is null).

  Takes keyword arguments to narrow down the results to a set of
  `schema-names` or `table-names`.

  Results match [[metabase.sync.interface/FieldIndexMetadata]].
  Results are optionally filtered by `schema-names` and `table-names` provided.

describe-table [driver database table]
Return a map containing a single field `:fields` that describes the fields in a `table`. `database` will be an
  instance of the `Database` model; and `table`, an instance of the `Table` model. It is expected that this function
  will be peformant and avoid draining meaningful resources of the database. The value of `:fields` should be a set of
  values matching the [[metabase.sync.interface/TableMetadataField]] schema.
```

Fonte: Captura de tela feita pelo autor

Nem todos os multimétodos disponíveis precisam ser implementados, já que alguns não se aplicam ao contexto do SPARQL. Por exemplo, o multimétodo `truncate!` não é necessário, pois este trabalho não tem como objetivo realizar operações de deleção de dados; da mesma forma, `insert-into!` não é utilizado, uma vez que não há inserção de valores na fonte de dados. A seleção dos multimétodos a serem implementados foi uma decisão técnica, baseada nas funcionalidades efetivamente suportadas pelos *endpoints* SPARQL.

A definição desses multimétodos exige compreensão do fluxo de dados no Metabase: do clique do usuário na interface, passando pelas camadas de *frontend* e *backend*, atravessando o *driver*, até alcançar o *endpoint* SPARQL e retornar ao usuário em forma de dados tabulares. A figura 23 apresenta uma visão geral desse fluxo de alto nível, ilustrando a ida e a volta da informação.

Figura 23 – Diagrama geral da interação com o *driver*



Fonte: Elaborado pelo autor

5.5 AMBIENTE DE DESENVOLVIMENTO

Para a implementação do *driver*, foi necessário configurar um ambiente de desenvolvimento adequado às características do projeto. A linguagem Clojure foi escolhida por ser a linguagem nativa do Metabase, facilitando a integração com os multimétodos e interfaces existentes da plataforma.

O ambiente de desenvolvimento foi composto por Clojure CLI, Java Development Kit (JDK) 21 e um sistema de *build* baseado em *Makefile*. Estas ferramentas automatizaram tarefas fundamentais para o desenvolvimento do *driver*, como compilação, execução de testes e empacotamento. Para inspecionar o código-fonte do Metabase e desenvolver o *driver*, foi utilizado o Visual Studio Code[4] com a extensão Calva[5], que oferece recursos específicos para desenvolvimento em Clojure, como destaque de sintaxe, autocompletar, navegação de código e verificação de assinaturas de funções.

5.6 IMPLEMENTAÇÃO

O foco desta seção é destrinchar as principais decisões tomadas no desenvolvimento do *driver*. Desde o ponto inicial de comunicação do *driver* com *endpoints* SPARQL, até chegar à visualização dos dados no Metabase.

5.6.1 Testando a conexão com *endpoints* SPARQL

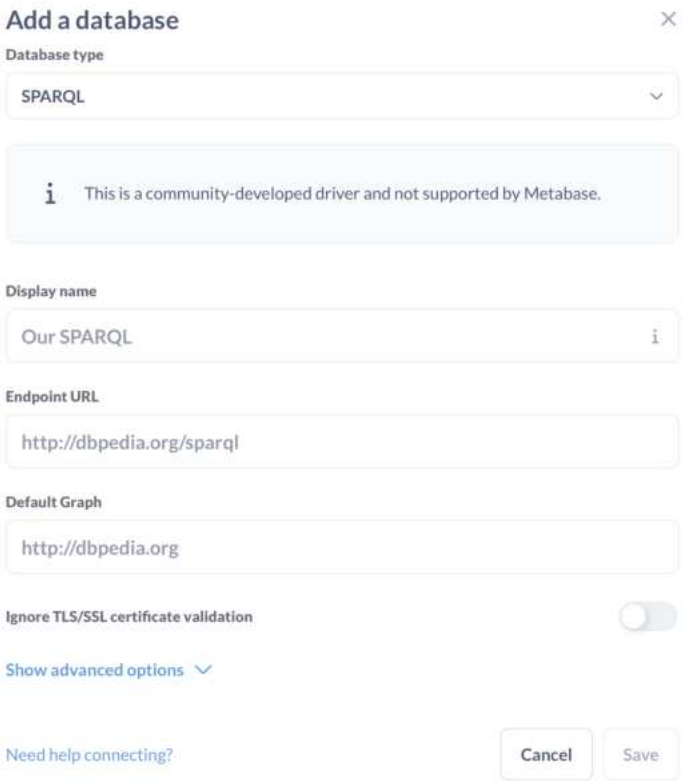
A primeira decisão que precisou ser tomada foi a escolha das configurações que o *driver* irá utilizar para se conectar a *endpoints* SPARQL. Como visto no capítulo 2, os *endpoints* SPARQL permitem consultas via protocolos HTTP e HTTPS (seguro, via TLS). Esses protocolos de rede exigem URLs que indicam a localização do servidor com o qual se pretende estabelecer a comunicação. Portanto, dois campos básicos devem estar presentes na configuração do *driver*: URL do servidor e a opção de ignorar a validação do certificado TLS.

A URL deve ser um campo obrigatório do tipo *string*, e a possibilidade de ignorar a validação do certificado TLS deve ser um campo opcional do tipo booleano, ou seja, o usuário poderá permitir que o *driver* não valide o certificado quando necessário. Além dessas duas configurações, o protocolo SPARQL 1.1 permite que seja definido um grafo para a consulta, ou seja, é necessário um terceiro campo do tipo *string* que indica o grafo que será consultado.

Na figura 24, é possível observar a interface que o usuário utilizará para adicionar uma nova conexão com um servidor SPARQL. Ela contém um campo para o nome da conexão, outro para a URL do *endpoint* SPARQL e um terceiro para o grafo padrão, sendo

[4]https://code.visualstudio.com
[5]https://marketplace.visualstudio.com/items?itemName=betterthantomorrow.calva

Figura 24 – Interface de adição de conexão no Metabase



Fonte: Captura de tela feita pelo autor

esse último opcional. Além disso, há a opção de ignorar conexões seguras, conforme os objetivos iniciais. Porém, não basta que a interface de consulta seja definida; é preciso que os multimétodos responsáveis por testar a conexão sejam implementados.

O primeiro multimétodo a ser executado quando uma nova conexão é criada é o `can-connect?`. Ele tem a função de verificar se o *endpoint* está acessível assim que o botão "*Save*" é clicado. Devido à variedade de formas de se testar uma conexão a um servidor SPARQL (LEHMANN et al., 2017), algumas estratégias foram adotadas ao longo do desenvolvimento desse método até que a mais eficiente do ponto de vista do *driver* fosse escolhida.

A primeira delas consistia em executar uma consulta simples do tipo `ASK`, "`ASK WHERE {?s ?p ?o}`". Essa consulta verifica se existe qualquer tripla RDF no grafo; existindo, ela retorna verdadeiro, caso contrário, falso. Mas seria necessária uma pequena observação ao retorno dessa consulta: o servidor SPARQL pode estar disponível, mas ainda não ter nenhuma tripla armazenada. Tendo como consequência uma conexão feita com sucesso, mas retornaria falso. O retorno falso não seria um problema em si, pois um servidor pode estar disponível, mas ainda estar sem triplas, caso seja uma instância nova ou um novo grafo.

Se fosse necessário receber um valor no retorno da consulta, poderia ser utilizada uma consulta simples de `SELECT` com o `BIND`. O `BIND` atribui um valor a uma variável na consulta, por exemplo, na consulta "`SELECT ?ping WHERE {BIND('pong' AS ?ping)}`" a variável `ping` receberia o valor `pong`. Desta forma, a consulta sempre teria um retorno, porém, o `BIND` só foi introduzido no SPARQL 1.1, então essa solução só seria viável para *endpoints* com a versão 1.1 do SPARQL.

A decisão final foi utilizar a versão com `ASK` mais simples que a primeira apresentada. A consulta "`ASK { }`" pode ser utilizada para teste, pois sempre retorna verdadeiro. Além dessa consulta funcionar nas versões 1.0 e 1.1 do SPARQL, ainda funciona na versão 1.2 que está em fase de rascunho[6]. A vantagem da consulta `ASK` é o desempenho que ela entrega. Quanto antes a consulta retornar o resultado e quanto menor for a quantidade de dados trafegados na rede, menor é a latência nessa operação.

Na tabela 8, é apresentada a lista de *endpoints* SPARQL que foram utilizados para teste de conexão. Ao todo, temos oito *endpoints* públicos, selecionados com foco em obter uma variedade de configurações diferentes. A última coluna indica quando é necessário que o usuário configure o *driver* para ignorar a validação do certificado TLS, quando o servidor tiver certificado inválido. Essa opção deve ser evitada sempre que possível, pois reduz a segurança da conexão, sendo indicada apenas para testes. O objetivo em utilizá-los é ter como base para atender aos requisitos e critérios de aceitação apresentados.

Tabela 8 – *Endpoints* SPARQL utilizados para testes

Nome	**URL**	**Domínio**	**Ignorar TLS**
Wikidata	https://query.wikidata.org/sparql	Base de conhecimento geral	Não
DBpedia	https://dbpedia.org/sparql	Dados estruturados da Wikipedia	Não
UniProt	https://sparql.uniprot.org/sparql	Informações sobre proteínas	Não
AgroVoc	https://agrovoc.fao.org/sparql	Vocabulário agrícola controlado (FAO)	Não
Datos España	https://datos.gob.es/virtuoso/sparql	Dados governamentais espanhóis	Sim
LinkedGeoData	https://linkedgeodata.org/sparql	Dados geográficos OpenStreetMap	Não
CoyPu	https://copper.coypu.org/coypu/	Análise de crises e tendências	Não
OSM - Sophox	https://sophox.org/sparql	Consultas SPARQL sobre OpenStreetMap	Não

Fonte: Elaborado pelo autor

[6]https://www.w3.org/TR/sparql12-query/

5.6.2 Verificando funcionalidades disponíveis no servidor

Como apresentado nos testes de conexão, uma das nuances de trabalhar com *endpoints* SPARQL é a variedade de comportamentos que eles podem apresentar (LEHMANN et al., 2017). Apesar das recomendações do W3C, diversos fatores podem provocar diferenças de implementação, seja por decisões dos administradores que habilitam ou desabilitam determinadas funcionalidades, seja por características específicas de cada servidor SPARQL. Uma das principais consequências dessas diferenças é o impacto no funcionamento do *query builder* do Metabase.

No Metabase, algumas funções podem ser habilitadas ou desabilitadas de acordo com os recursos que a fonte de dados suporta, como agregações, parâmetros, expressões regulares ou funções de data e hora. Para essa verificação, o *query builder* utiliza o multimétodo "`database-supports?`", executado quando uma nova conexão é adicionada. A estratégia adotada consiste em testar a compatibilidade de cada funcionalidade por meio de uma consulta SPARQL que a utilize diretamente.

Por exemplo, para verificar se o servidor suporta a função de hora atual, foi utilizada a consulta: `SELECT ?now WHERE { BIND( now()AS ?now ). }` Se houver retorno de resultado, conclui-se que o servidor aceita essa funcionalidade, e o multimétodo "`database-supports?`" deve retornar verdadeiro para ela. Essa abordagem foi aplicada de forma semelhante para as demais funcionalidades. Na versão desenvolvida por este trabalho, foi implementada apenas a verificação de suporte à função de hora atual, `NOW()`. As demais verificações permanecem como trabalhos futuros.

Para manter o sistema modular, foi utilizada a estratégia de polimorfismo, explicada no capítulo 4. Um multimétodo para testes de funcionalidades do servidor SPARQL foi implementado, ao qual poderão ser implementados, no futuro, os métodos específicos de cada funcionalidade que o *query builder* suporta e faz sentido no contexto do SPARQL.

5.6.3 Obtendo a lista de classes e a frequência de uso

Conforme destrinchado na seção 2.5, a tabela de propriedades particionada por classes é utilizada para representar as classes como tabelas na interface do Metabase. Essa abordagem tem o objetivo de auxiliar a exploração do grafo de maneira visual.

Para representar classes como tabelas, é necessário considerar que os dados da Web Semântica são extremamente heterogêneos. Para ilustrar essa diversidade, basta observar o número de classes disponíveis na DBpedia ou no UniProt por meio de uma consulta SPARQL. Quando a consulta apresentada no código 8 é executada, ela retorna 1.568 classes na DBpedia e 470.306 no UniProt[7]. Esses valores dão uma dimensão do desafio de lidar com grandes volumes de dados e representá-los como tabelas.

[7]Consulta executada em 04 de agosto de 2025

Código 8 – Consulta SPARQL para obter a quantidade de classes em um servidor

```
PREFIX owl:  <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT (COUNT(DISTINCT ?class) AS ?total_classes)
WHERE {
  ?class a ?type .
  FILTER(?type IN (rdfs:Class, owl:Class))
}
```

Diante disso, foi adotada a estratégia de priorizar as classes mais utilizadas, ordenando-as da mais frequente para a menos frequente. Dessa forma, foi possível limitar a quantidade de classes que serão exibidas como tabelas na interface do Metabase. O código 9 apresenta a consulta usada para listar as classes mais frequentes. É importante destacar que essa limitação é apenas visual; todas as classes continuam disponíveis para consultas no grafo RDF.

Código 9 – Consulta SPARQL para listar as classes mais frequentes

```
SELECT ?class (COUNT(?s) AS ?count)
WHERE {
  ?s a ?class .
}
GROUP BY ?class
ORDER BY DESC(?count)
LIMIT 20
```

Na figura 25, é possível verificar uma classe específica sendo representada como tabela no Metabase. Devido à limitação da ferramenta, as classes são representadas somente por sua IRI e não é utilizado nenhum outro rótulo com um nome mais amigável, pois isso exigiria modificações da base de código principal e não só do *driver*.

Como não é possível interferir nos recursos utilizados nos *endpoints* SPARQL públicos, pode ocorrer um longo tempo para ser finalizada esta etapa, ou seja, dependendo da quantidade de triplas e suas classes, a consulta de agregação e ordenamento de classes mais usadas pode demorar demasiadamente. Para evitar esse problema na obtenção das classes mais usadas, foi implementada uma opção para desativar a obtenção das "tabelas" e, consequentemente, das "colunas". Isso permite que consultas SPARQL ainda sejam feitas, porém com o custo de não ter as classes e propriedades disponíveis na interface do Metabase. Essa opção pode ser encontrada na seção de opções avançadas da tela de configuração da fonte de dados (figura 26).

Figura 25 – Representação de uma classe como tabela no Metabase

Fonte: Captura de tela feita pelo autor

Figura 26 – Opção para desativar a sincronização de metadados

Fonte: Captura de tela feita pelo autor

5.6.4 Representando predicados como colunas

De forma semelhante à representação das classes como tabelas, na tabela de propriedades particionada por classes, os predicados são representados como colunas. Cada predicado encontrado nas instâncias de determinada classe se torna uma coluna correspondente na tabela de propriedades.

Para evitar a sobrecarga da interface, a mesma abordagem utilizada para listar as classes é utilizada para listar os predicados. Portanto, a estratégia de priorizar os predicados mais usados por cada instância que pertence a uma classe específica é adotada. Os menos frequentes não são exibidos na interface do Metabase.

A frequência de cada predicado é determinada pela consulta SPARQL representada no código 10. Ela contabiliza quantas vezes o predicado é utilizado para uma instância de uma classe específica. Para evitar sobrecarga no servidor, foi utilizado um limite de instâncias a serem consultadas ao obter a frequência.

Código 10 – Consulta SPARQL para obter a quantidade de propriedades mais frequentes de uma classe

```
SELECT ?property (COUNT(?instance) AS ?count)
WHERE {
  { SELECT ?instance WHERE {
      ?instance a {{class}}
    } LIMIT {{sample-size}}
  }
  ?instance ?property ?value .
}
GROUP BY ?property
ORDER BY DESC(?count)
LIMIT 20
```

A escolha por limitar a quantidade de predicados foi necessária para evitar gerar tabelas com centenas de colunas, o que tornaria a exibição pouco prática. Além disso, evita a sobrecarga do banco de dados que o Metabase utiliza para armazenar metadados, evitando degradação de desempenho da ferramenta. Por fim, essa limitação não influencia a capacidade de executar consultas sobre todo o grafo RDF.

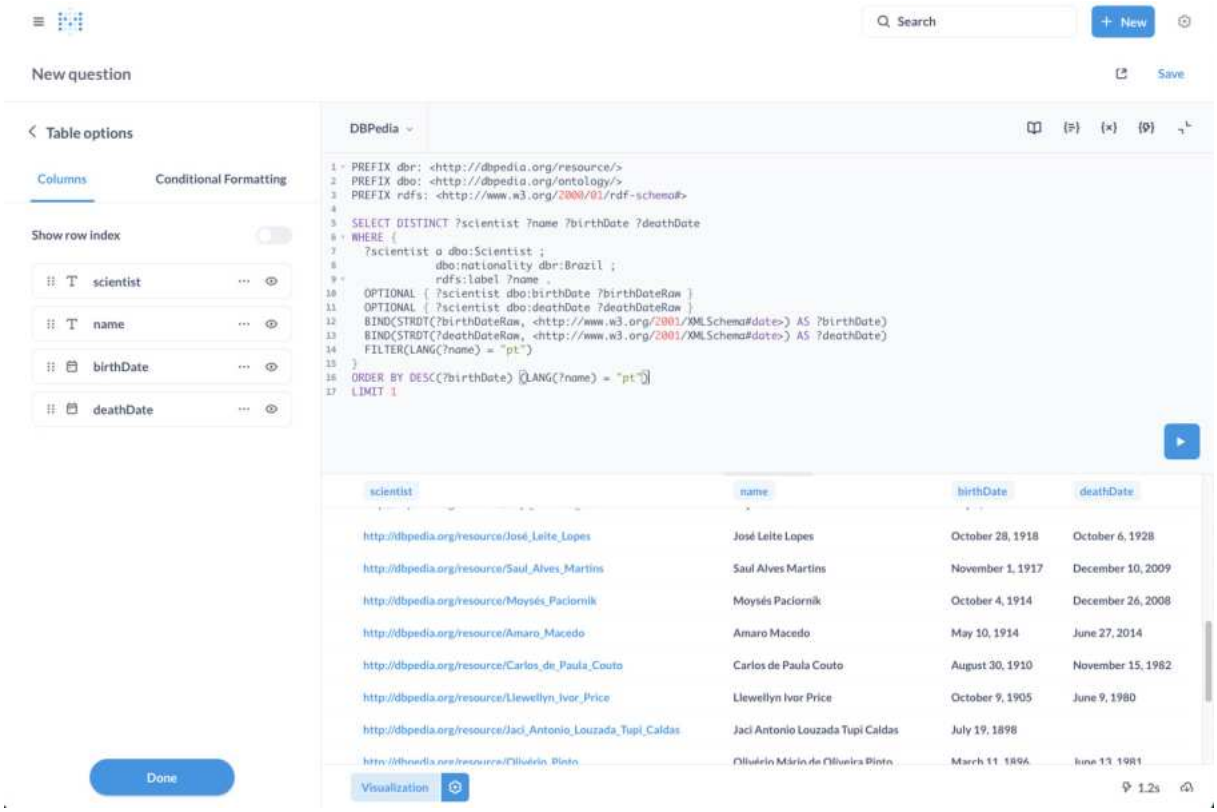
5.6.5 Executando consultas SPARQL

Uma das principais funções do Metabase é o editor de consultas nativas, que permite aos usuários escrever consultas diretamente na linguagem que a fonte de dados compreende. No contexto deste *driver*, isso significa que os usuários podem escrever consultas SPARQL completas, aproveitando todos os recursos que a linguagem oferece para explorar os dados RDF disponíveis nos *endpoints* SPARQL.

Conforme definido no escopo do projeto, o *driver* deve oferecer suporte completo aos tipos de consultas `SELECT` e `ASK` do SPARQL. As consultas `SELECT` retornam conjuntos de dados em formato tabular, ideais para a criação de visualizações e *dashboards* no Metabase. Por exemplo, uma consulta que recupera informações sobre cidades brasileiras pode retornar múltiplos registros com diferentes propriedades, permitindo a criação de gráficos de barras, rosca ou mapas. Já as consultas `ASK` fornecem respostas booleanas simples, úteis para verificações de existência ou validações condicionais que podem ser utilizadas em *dashboards* com indicadores. A figura 27 ilustra uma consulta `SELECT` básica no editor do Metabase.

A execução das consultas segue o protocolo SPARQL 1.0 e 1.1, conforme definido pelo W3C (W3C SPARQL Working Group, 2013), mas o *driver* apenas encaminha a consulta para o servidor e quem deve suportar as funcionalidades de cada versão do protocolo é o servidor SPARQL, deixando o *driver* responsável por processar o retorno da consulta. Para isso, o *driver* usa a função representada no código 11 chamada

Figura 27 – Exemplo de consulta `SELECT` básica para demonstrar a sintaxe suportada pelo *driver*



Fonte: Captura de tela feita pelo autor

`execute-reducible-query`, que pega a consulta do Metabase, prepara adequadamente o corpo da requisição e envia via HTTP POST para o servidor SPARQL. Durante esse processo, o *driver* faz várias operações: configura os cabeçalhos HTTP corretos (incluindo o `Accept: application/sparql-results+json` para garantir resposta em JSON) e trata possíveis erros de rede ou do servidor.

Para garantir que o *driver* funcione com diferentes *endpoints* SPARQL, ele precisa processar resultados JSON de duas versões do protocolo, tanto o 1.0 quanto o 1.1. Embora ambas as versões usem a mesma estrutura básica, há uma pequena diferença na forma como os dados tipados são representados, o que afeta como o *driver* processa essas informações. Na versão original do SPARQL 1.0, valores com tipos específicos aparecem no JSON como objetos `typed-literal` (CLARK; FEIGENBAUM; TORRES, 2007). O código 12 mostra um exemplo de retorno real no formato SPARQL 1.0.

Código 11 – Função em Clojure responsável por enviar a consulta SPARQL via HTTP

```
(defn execute-reducible-query
 [native-query _context respond]
 (let [database (lib.metadata/database (qp.store/metadata-provider))
  endpoint (or
   (get-in native-query [:native :endpoint])
   (-> database :details :endpoint))
  sparql-query (get-in native-query [:native :query])
  options {:default-graph (-> database :details :default-graph)
           :insecure? (-> database :details :use-insecure)}
  [success result] (execute-sparql-query endpoint sparql-query options)]
 (if success
  (query-processor/process-query-results result respond)
  (respond {:cols []} []))))

(defn execute-sparql-query
 [endpoint query options]
 (try
  (let [http-options (create-http-options query options)
        response (http/post endpoint http-options)]
   (process-response response))
  (catch Exception e
   [false (.getMessage e)])))
```

Código 12 – Exemplo de resposta JSON no formato SPARQL 1.0 com `typed-literal`

```
...
    "bindings": [
        {
            "scientist": {
                "type": "uri",
                "value": "http://dbpedia.org/resource/Welington_de_Melo"
            },
            "birthDate": {
                "type": "typed-literal",
                "datatype": "http://www.w3.org/2001/XMLSchema#date",
                "value": "1946-11-17"
            }
        }
...
```

Na versão 1.1 do protocolo, isso foi simplificado, usando apenas `literal` como tipo (HAWKE et al., 2013b). Um exemplo deste caso é mostrado no código 13.

Código 13 – Exemplo de resposta JSON no formato SPARQL 1.1 com `literal`

```
...
    "bindings": [
        {
            "thing": {
                "type": "uri",
                "value": "https://data.coypu.org/commodity/ecb/BRL"
            },
            "value": {
                "type": "literal",
                "datatype": "http://www.w3.org/2001/XMLSchema#decimal",
                "value": "1.0153239210242735"
            }
        }
...
```

Para funcionar com qualquer servidor SPARQL na Web, o *driver* aceita os dois formatos. O sistema de conversão verifica automaticamente se a resposta usa `typed-literal` ou `literal` e converte os dados da mesma forma para o Metabase, garantindo que tudo funcione independentemente da versão do servidor consultado.

Além disso, o *driver* garante que o formato dos dados retornados pelo servidor SPARQL seja convertido para o formato tabular do Metabase. Para isso, foi criada a função `sparql-type->base-type`, representada no código 14, que recebe os tipos SPARQL e converte para os tipos Metabase. Por ora, a função só aceita os tipos definidos na especificação XSD, garantindo que o requisito 7 (R7) seja atendido. No futuro, a função poderá ser expandida para suportar mais tipos de dados. Demais tipos de dados são tratados como texto literal, e IRIs são tratadas como URLs.

Código 14 – Função em Clojure responsável por interpretar o tipo de dado

```
(defn sparql-type->base-type
 [sparql-type datatype]
 (let [base-type (cond
  (= sparql-type "uri") :type/URL
  (= sparql-type "bnode") :type/Text
  (or (and (= sparql-type "typed-literal") datatype)
    (and (= sparql-type "literal") datatype))
  (cond
    (or (= datatype "http://www.w3.org/2001/XMLSchema#integer")
      (= datatype "http://www.w3.org/2001/XMLSchema#int")
      (= datatype "http://www.w3.org/2001/XMLSchema#long")
      (= datatype "http://www.w3.org/2001/XMLSchema#short")
      (= datatype "http://www.w3.org/2001/XMLSchema#byte")
      (= datatype "http://www.w3.org/2001/XMLSchema#nonNegativeInteger")
      (= datatype "http://www.w3.org/2001/XMLSchema#positiveInteger")
      (= datatype "http://www.w3.org/2001/XMLSchema#nonPositiveInteger")
      (= datatype "http://www.w3.org/2001/XMLSchema#negativeInteger")
      (= datatype "http://www.w3.org/2001/XMLSchema#unsignedLong")
      (= datatype "http://www.w3.org/2001/XMLSchema#unsignedInt")
      (= datatype "http://www.w3.org/2001/XMLSchema#unsignedShort")
      (= datatype "http://www.w3.org/2001/XMLSchema#unsignedByte")) :
        type/Integer
    (or (= datatype "http://www.w3.org/2001/XMLSchema#decimal")
      (= datatype "http://www.w3.org/2001/XMLSchema#float")
      (= datatype "http://www.w3.org/2001/XMLSchema#double")) :
        type/Float
    (= datatype "http://www.w3.org/2001/XMLSchema#boolean") :
       type/Boolean
    (or (= datatype "http://www.w3.org/2001/XMLSchema#dateTime")
      (= datatype "http://www.w3.org/2001/XMLSchema#gYear")
      (= datatype "http://www.w3.org/2001/XMLSchema#gYearMonth")) :
        type/DateTime
    (or (= datatype "http://www.w3.org/2001/XMLSchema#date")
      (= datatype "http://www.w3.org/2001/XMLSchema#gMonthDay")
      (= datatype "http://www.w3.org/2001/XMLSchema#gDay")
      (= datatype "http://www.w3.org/2001/XMLSchema#gMonth")) :type/Date
    (= datatype "http://www.w3.org/2001/XMLSchema#time") :type/Time
    :else :type/Text)
  :else :type/Text)]
  base-type))
```

Outra funcionalidade do *driver* é permitir que os usuários criem consultas com parâmetros variáveis. Seguindo o padrão do Metabase, os parâmetros são escritos com duplas

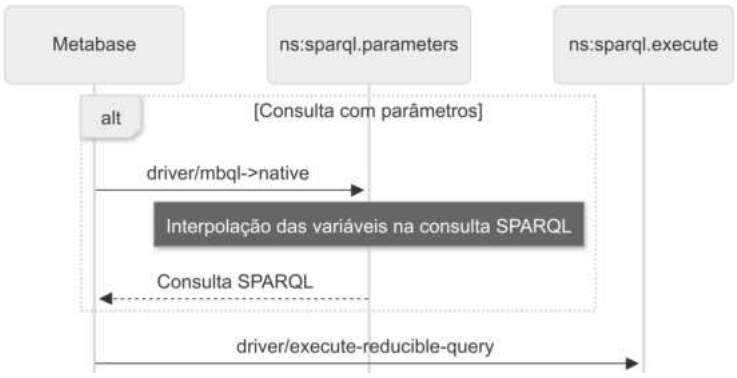
chaves `{{nome_parametro}}`, que são substituídos pelos valores que o usuário escolhe na interface. Isso é muito importante em análise de dados, pois permite reutilizar a mesma consulta com filtros diferentes, aumentando a produtividade dos analistas.

Figura 28 – Exemplo de consulta utilizando parâmetros

Fonte: Captura de tela feita pelo autor

Por exemplo, uma consulta com parâmetros permite que o usuário filtre dados por país, população mínima, idioma e limite de resultados sem reescrever a consulta inteira, conforme mostrado na figura 28. O *driver* faz isso através da função `substitute-native-parameters`, que encontra todos os marcadores de parâmetro (como `{{pais}}`) na consulta e os substitui pelos valores escolhidos pelo usuário antes de enviar para o servidor (figura 29). Dessa forma, consultas complexas podem ser facilmente adaptadas para diferentes situações de análise.

Figura 29 – Diagrama de sequência da interpolação de parâmetro



Fonte: Elaborado pelo autor

Todo esse processo pode ser representado pelo diagrama de sequência apresentado na figura 30. Desde o recebimento da consulta pelo Metabase até o retorno dos dados processados, o *driver* realiza várias operações para garantir que os dados sejam apresentados de forma consistente e adequada.

Figura 30 – Diagrama de sequência da execução de uma consulta desde a origem do Metabase até o endpoint SPARQL, englobando o processamento pelo *driver*

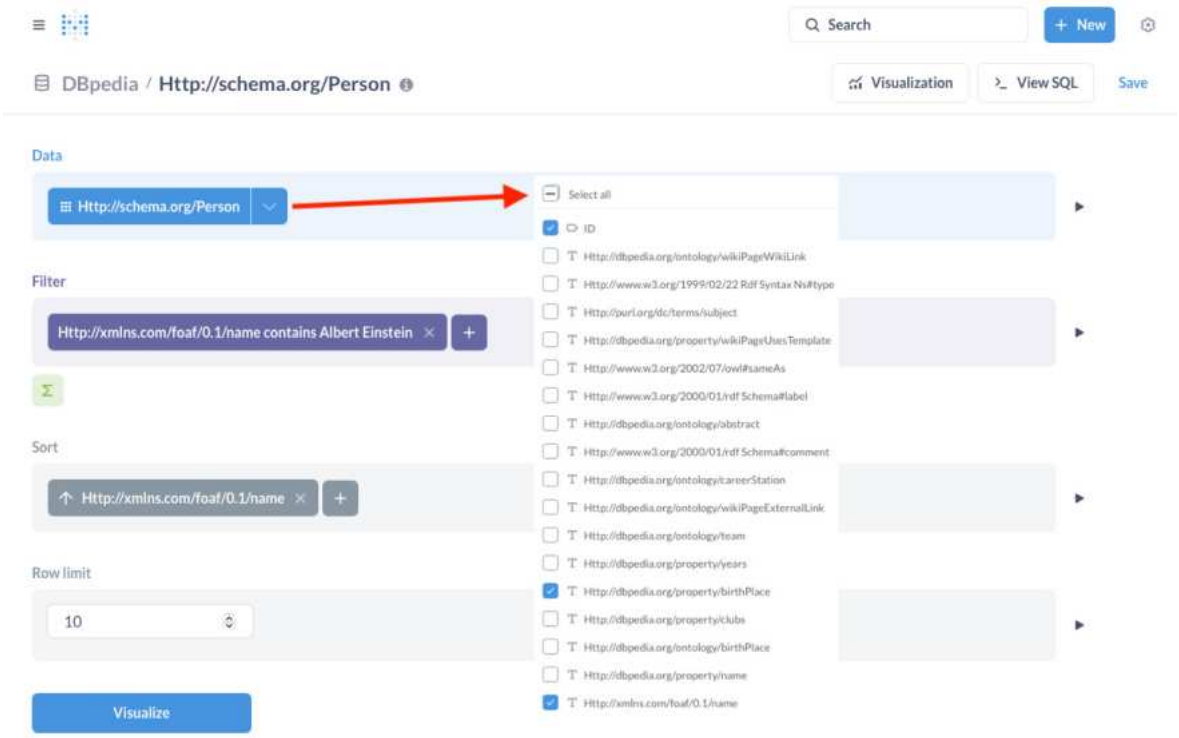
Fonte: Elaborado pelo autor

O diagrama de sequência mostra passo a passo as etapas que uma consulta passa pelo *driver*. Este, quando recebe um pedido para executar uma consulta SPARQL, precisa obter as configurações do *endpoint* cadastrado no Metabase, montar os cabeçalhos da requisição HTTP de forma correta para que o *endpoint* retorne os dados da forma esperada, fazer uma consulta *POST*, processar o JSON de retorno, identificar se o resultado corresponde a uma consulta do tipo `SELECT` ou `ASK`, para o correto processamento da resposta, converter os valores para os tipos nativos do Metabase e, por fim, exibir o resultado ao usuário.

5.6.6 Convertendo consulta MBQL em SPARQL

Como apresentado na seção 4.2.2, o Metabase permite construir consultas por meio de uma interface amigável, abstraindo a linguagem de consulta da fonte de dados. A figura 31 ilustra o *query builder* com uma consulta para recuperar IRIs de pessoas com um nome específico e a IRI do respectivo local de nascimento. A partir da construção visual, funções internas do Metabase geram uma representação declarativa em MBQL, que é então convertida pelo *driver* em uma consulta SPARQL, como ilustrado na figura 32.

Figura 31 – Demonstração do *query builder* buscando uma pessoa com nome específico



Fonte: Elaborado pelo autor

O MBQL resultante apresenta, de forma estruturada, a configuração do *endpoint* (`:database`), a classe selecionada (`:source-table`), os predicados a projetar (`:fields`), os filtros (`:filter`), as ordenações (`:order-by`) e o limite de linhas (`:limit`). Nesta im-

Figura 32 – Diagrama de sequência da conversão MBQL para SPARQL

Fonte: Captura de tela feita pelo autor

plementação, é adotada a tabela de propriedades particionada por classes (seção 2.5): classes RDF são expostas como "tabelas" e predicados como "colunas". Assim, `:source-table` é convertido para a IRI da classe e cada item em `:fields` é convertido para a IRI de um predicado associado à classe.

Código 15 – MBQL de uma consulta gerada para obter IRIs de pessoas com nome específico

```
{:database 22,
 :middleware {:js-int-to-string? true,
              :userland-query? true,
              :add-default-userland-constraints? true},
 :info {:executed-by 1,
        :context :ad-hoc,
        :query-hash #object["[B" 0x15e4b7d9 "[B@15e4b7d9"],
        :card-entity-id "$$ADHOC[oDi49WKUXQSDf23UxAnqv]"},
 :constraints {:max-results 10000, :max-results-bare-rows 2000},
 :type :query,
 :query {:source-table 181,
         :fields [[:field 7738 {:base-type :type/Text}]
                  [:field 7726 {:base-type :type/Text}]
                  [:field 7731 {:base-type :type/Text}]],
     :limit 10,
     :order-by [[:asc [:field 7726 {:base-type :type/Text}]]],
     :filter [:contains
              [:field 7726 {:base-type :type/Text}]
              [:value "Albert Einstein" {
                  :base_type :type/Text,
                  :effective_type :type/Text,
                  :coercion_strategy nil,
                  :semantic_type nil,
                  :database_type "string",
                  :name "http://xmlns.com/foaf/0.1/name"}]
              {:effective-type :type/Boolean}]}}
```

A conversão MBQL → SPARQL segue o seguinte fluxo:

- **Metadados**: `:source-table` → IRI da classe; campos → IRIs de predicado, obtidos a partir dos metadados armazenados no Metabase (código 17).
- **Variáveis e sujeito**: o sujeito é representado por `?subject`, pois a tabela de propriedades particionada por classes define a primeira coluna como sujeito; nomes de variáveis são normalizados para SPARQL, removendo caracteres especiais (código 16).
- **SELECT**: sempre inclui `?subject` para ser projetado e adiciona as variáveis dos predicados selecionados.
- **WHERE**: inclui `?s a <Classe>` e, para cada predicado referenciado em `:fields`, `:order-by` ou `:filter`, insere um padrão `OPTIONAL { ?s <predicado> ?var . }`. Filtros MBQL são mapeados para `FILTER` com operadores lógicos (`:and`, `:or`, `:not`), comparações (`:=`, `:!=`, `:>`, `:>=`, `:<`, `:<=`), existência (`:is-null` → `!BOUND`, `:not-null` → `BOUND`) e operações de texto (`:starts-with`, `:ends-with`, `:contains`, com variante *case-insensitive* via `LCASE(STR(?var))` quando `:case-sensitive false`).
- **Ordenação e limite**: `:order-by`→`ORDER BY ASC|DESC(?var)`; `:limit`→`LIMIT n`.
- **Literais**: *strings* escapadas; números e booleanos emitidos como literais SPARQL; valores `nil` tratados com `BOUND`/`!BOUND` de acordo com o operador.

Código 16 – Normalização de nomes de variáveis

```
(defn- sanitize-var-name
  "Return a SPARQL-safe variable name."
  [s]
  (let [base (-> (str s)
                 (str/replace #"[^A-Za-z0-9_]" "_")
                 (str/replace #"^([0-9])" "_$1"))]
    (if (str/blank? base) "v" base)))
```

Código 17 – Obter nome da classe a partir do `:source-table`

```
(defn- table-id->class-uri
  "Resolve RDF class URI (table name) from :source-table."
  [table-id]
  (let [uri (some-> (lib.metadata/table (qp.store/metadata-provider)
     table-id) :name)]
    (log/debugf "[mbql] Resolved class URI for table-id %s: %s"
       table-id uri)
    uri))
```

Código 18 – Trecho da função que mapeia filtros MBQL para SPARQL

```
(defn- compile-filter-expr
 "Compile a filter clause to a SPARQL boolean expression string."
 [filter-clause field-id->var]
 ...
  (case op
  := (if (nil? v)
       (format "(!BOUND(?%s))" var)
       (format "(?%s = %s)" var (literal->sparql v)))
  :!= (if (nil? v)
       (format "(BOUND(?%s))" var)
       (format "(?%s != %s)" var (literal->sparql v)))
  :> (and (some? v) (format "(?%s > %s)" var (literal->sparql v)))
  :>= (and (some? v) (format "(?%s >= %s)" var (literal->sparql v)))
  :< (and (some? v) (format "(?%s < %s)" var (literal->sparql v)))
  :<= (and (some? v) (format "(?%s <= %s)" var (literal->sparql v)))
  :starts-with (let [needle (literal->sparql v)
        expr (if insensitive?
        (format "STRSTARTS(LCASE(STR(?%s)), LCASE(%s))" var needle)
        (format "STRSTARTS(STR(?%s), %s)" var needle))]
      (str "(" expr ")"))
  :ends-with (let [needle (literal->sparql v)
        expr (if insensitive?
        (format "STRENDS(LCASE(STR(?%s)), LCASE(%s))" var needle)
        (format "STRENDS(STR(?%s), %s)" var needle))]
      (str "(" expr ")"))
  :contains (let [needle (literal->sparql v)
        expr (if insensitive?
        (format "CONTAINS(LCASE(STR(?%s)), LCASE(%s))" var needle)
        (format "CONTAINS(STR(?%s), %s)" var needle))]
      (str "(" expr ")"))
  :is-null (format "(!BOUND(?%s))" var)
  :not-null (format "(BOUND(?%s))" var)
  nil)))))))
```

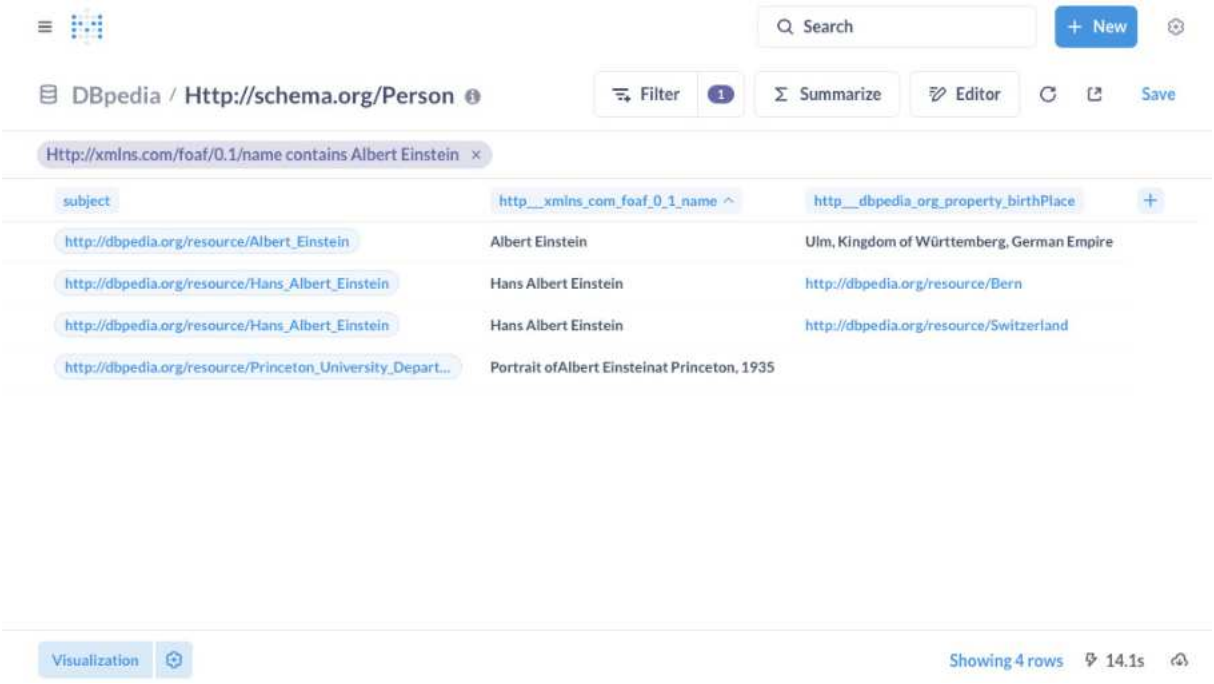
Além dos predicados presentes em `:fields`, o *driver* inclui padrões `OPTIONAL` (código 19) para campos utilizados exclusivamente em `:filter` ou `:order-by`, ainda que não sejam adicionados no `SELECT`. Essa decisão garante que filtros e ordenações funcionem corretamente sem alterar o conjunto de colunas exibidas, mantendo o *query builder* intuitivo e os resultados coerentes com a consulta construída na ferramenta visual.

Código 19 – Função que constrói o padrão `OPTIONAL`

```
(defn- ensure-triple-for-field
 "Build OPTIONAL triple pattern for property and var."
 [property-uri var-alias]
 (let [triple (format " OPTIONAL { ?s <%s> ?%s . }" property-uri
    var-alias)]
  (log/debugf "[mbql] OPTIONAL triple: property=%s var=?%s"
     property-uri var-alias)
  triple))
```

A versão atual do *driver* não cobre agregações, agrupamentos, expressões calculadas ou junções. O foco permanece na seleção de instâncias de uma classe com predicados opcionais, filtros básicos, ordenação e limite. Por essa razão, a conversão MBQL → SPARQL não compõe os requisitos e critérios de aceitação, uma vez que as visualizações originadas do *query builder* ficam limitadas ao tipo tabela. Ainda assim, a implementação é documentada como base para trabalhos futuros.

Figura 33 – Resultado da consulta feita via *query builder*



Fonte: Captura de tela feita pelo autor

Na visualização de resultados (figura 33), a coluna `subject` corresponde ao sujeito RDF de cada instância. Os nomes de variáveis derivados de predicados são normalizados para nomear as colunas de forma consistente na interface. O uso de `OPTIONAL` garante que o SPARQL seja utilizado de forma semelhante ao comportamento do SQL, onde os valores nulos são tratados como ausência de dados. Caso contrário, o não uso do `OPTIONAL`

implicaria em resultados que contêm todas as colunas preenchidas, ou dependeria somente dos filtros. Essa decisão foi tomada para acompanhar o comportamento do SQL e pode ser revista no futuro, caso não se deseje mais esse comportamento.

Figura 34 – Consulta SPARQL gerada pelo *driver* a partir do MBQL

Fonte: Captura de tela feita pelo autor

A figura 34 ilustra a consulta SPARQL gerada pelo *driver* a partir do MBQL. Para trabalhos futuros, é possível incorporar agregações (`COUNT`, `SUM`, `AVG`), `GROUP BY`/`HAVING`, expressões calculadas e expressões regulares, além de permitir consultas multi-classe por meio de *joins*.

5.6.7 Disponibilização de uma imagem Docker

A estratégia de conteinerização implementada utiliza o conceito de *multi-stage build*. Ela é uma técnica do *Docker* que otimiza o tamanho final da imagem, melhora a segurança ao separar o ambiente de construção do ambiente de execução e facilita o reuso de etapas anteriores. No código 20, é possível verificar três cláusulas `FROM` que representam os estágios que foram utilizados para a construção da imagem.

Código 20 – Exemplo de Dockerfile para o *driver* SPARQL

```
FROM clojure:temurin-21-tools-deps-trixie-slim AS builder-base

WORKDIR /app

COPY metabase/ ./metabase

COPY deps.edn Makefile ./
COPY resources/ ./resources
COPY src/ ./src

ENTRYPOINT ["make", "build"]

FROM builder-base AS builder

RUN make build

FROM metabase/metabase:v0.55.x

COPY --from=builder /app/target/sparql.metabase-driver.jar /plugins/
```

O primeiro estágio, nomeado `builder-base`, é utilizado para a construção do *driver*. Ele utiliza como base a imagem `clojure:temurin-21-tools-deps-trixie-slim` e copia os arquivos do projeto para o diretório `/app`. Nele, são copiados os arquivos do projeto e definido o `ENTRYPOINT` que será executado quando o contêiner referente a este estágio for iniciado. Isso é importante para garantir que a imagem possa ser utilizada também para a compilação do *driver* em uma etapa do fluxo de integração contínua.

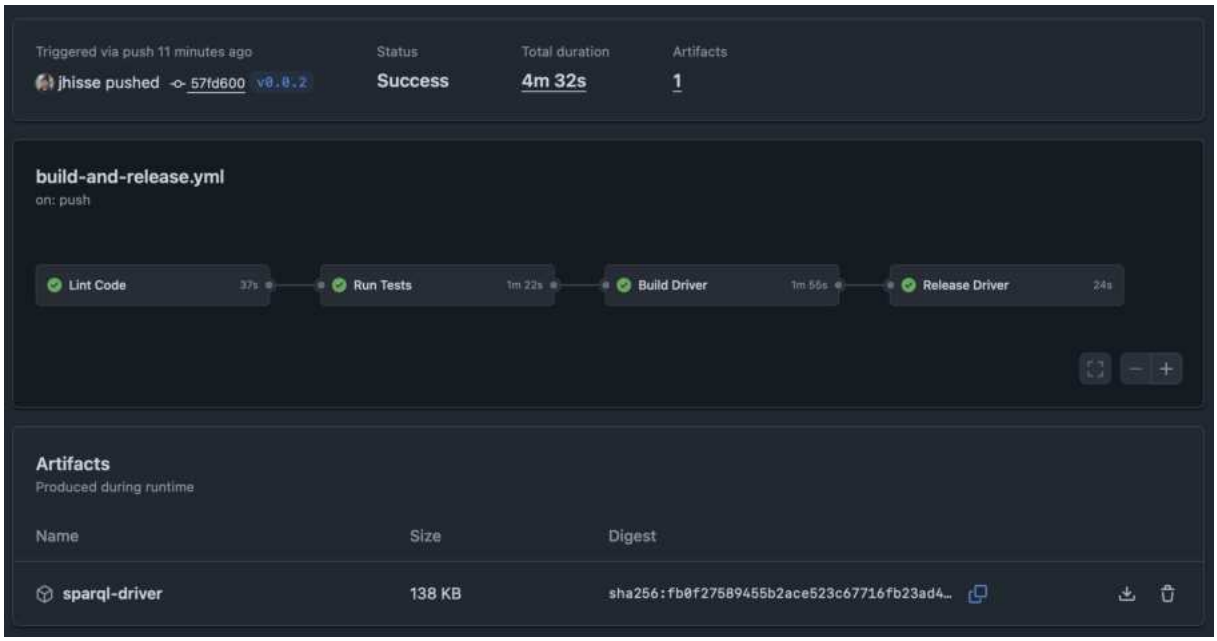
O segundo estágio, nomeado `builder`, é auxiliar para o fluxo de integração contínua. Ele herda do estágio anterior e executa o comando `make build` para a construção do *driver*. Ele é auxiliar à terceira etapa e só é necessário para compilar o *driver* quando a imagem completa é construída.

O terceiro estágio utiliza a imagem oficial que o Metabase disponibiliza. Nele, é copiado o artefato compilado do segundo estágio, o JAR. Essa última etapa é necessária para garantir que o *driver* seja instalado no Metabase e habilita a execução da plataforma já com o *driver* instalado.

5.7 INTEGRAÇÃO CONTÍNUA E DISPONIBILIZAÇÃO DO DRIVER

Para garantir a entrega do artefato aos usuários de maneira rápida e garantindo a qualidade do código, foi desenvolvido um fluxo de integração contínua no repositório do projeto. Esse fluxo conta com 4 etapas que são executadas em sequência, que vão desde a análise do código até a publicação do artefato JAR no GitHub (figura 35).

Figura 35 – Fluxo de integração contínua feita no GitHub



Fonte: https://github.com/jhisse/metabase-sparql-driver/actions/runs/16781643561

O fluxo foi implementado utilizando o GitHub Actions[8], funcionalidade gratuita da própria plataforma para repositórios hospedados nela. As etapas são processadas em máquinas virtuais fornecidas pelo GitHub, que executam o sistema operacional Ubuntu. Toda a definição do fluxo é realizada por meio de um arquivo no formato *YAML*, como demonstrado em trecho no código 21.

A primeira etapa consiste em verificar se o *software* está seguindo as melhores práticas da linguagem Clojure. Para isso, foram utilizadas ferramentas conhecidas como *linters*, especializadas em análise estática de código. A clj-kondo[9] e a Splint[10] foram as escolhidas para essa função. Elas permitem validar o código antes que seja compilado ou executado, bloqueando o fluxo se existir algum erro na sintaxe ou algo que poderia estar melhor organizado ou otimizado.

Após a garantia de que o código não possui erros de sintaxe, são executados os testes unitários. Eles consistem em checagens se determinadas funções produzem o resultado esperado de acordo com entradas pré-determinadas. Isso garante que o código funcione de acordo com o que foi projetado. Caso algum teste não produza o resultado esperado, então o fluxo não deve seguir adiante para a próxima etapa.

[8] https://docs.github.com/pt/actions
[9] https://github.com/clj-kondo/clj-kondo
[10] https://github.com/NoahTheDuke/splint

Código 21 – Trecho do fluxo de integração contínua em formato YAML utilizado no GitHub Actions

```yaml
name: Build and Release Driver
on:
  push:
    branches: [ main ]
    tags: [ 'v*.*.*' ]
  pull_request:
    branches: [ main ]
jobs:
  lint:
    name: Lint Code
    runs-on: ubuntu-latest
    steps:
    - name: Checkout code
      uses: actions/checkout@v4
    - name: Setup Java
      uses: actions/setup-java@v4
      with:
        distribution: 'temurin'
        java-version: '21'
    - name: Setup Clojure Tools
      uses: DeLaGuardo/setup-clojure@13.4
      with:
        cli: latest
    - name: Run Linting
      run: make lint
    - name: Run Static Analysis
      run: make splint
  tests:
    name: Run Tests
    runs-on: ubuntu-latest
    needs: lint
    steps:
...
```

A etapa seguinte corresponde à compilação do *driver*. Uma vez que todas as validações anteriores são executadas com sucesso, o artefato JAR é gerado com o objetivo de ser utilizado em testes adicionais caso o desenvolvedor queira fazer o download. Nessa fase, o artefato ainda não é publicado como uma nova versão oficial.

A disponibilização pública do *driver* ocorre apenas quando uma nova *tag* é criada no repositório Git. Esse evento aciona a criação de um novo *release*, o que resulta na publicação automática do artefato na seção de *releases*[11] do GitHub.

[11] https://github.com/jhisse/metabase-sparql-driver/releases

6 AVALIAÇÃO E DISCUSSÃO DOS RESULTADOS

Neste capítulo são apresentados os testes propostos na implementação, assim como o alinhamento aos objetivos inicialmente mapeados. Além disso, são apresentadas as evidências de que o *software* atingiu os critérios de aceitação definidos na seção 5.2.

O objetivo principal foi permitir que o Metabase se conectasse a *endpoints* SPARQL via protocolo de rede e, a partir dos resultados obtidos, apresentasse esses dados na interface, possibilitando a criação de visualizações. Os critérios de aceitação deixam isso explícito quando se propõem a validar se os requisitos apontados foram contemplados de forma satisfatória ou não.

Para o desenvolvimento do *driver* foi utilizado o Metabase versão 0.55.5.1[1] (versão lançada em 24 de junho de 2025). Além disso, o *driver* foi desenvolvido em Clojure versão 1.12.1.1550 e o OpenJDK 21.0.7.

6.1 DISPONIBILIZAÇÃO E INSTALAÇÃO DO DRIVER

Nesta seção, é avaliada a disponibilização pública e a instalação do *driver*. Para isso, são detalhados os seguintes critérios de avaliação:

- `C1` - Repositório GitHub público com arquivo `LICENSE`
- `C11` - Copiar `metabase-sparql-driver.jar` para `plugins/` e o Metabase não deve lançar exceções na inicialização
- `C12` - Entrega do *driver* em uma imagem *Docker* para execução em contêiner com o Metabase, acompanhada de documentação de uso

6.1.1 Validação do Critério C1 - Repositório Público e Licenciamento

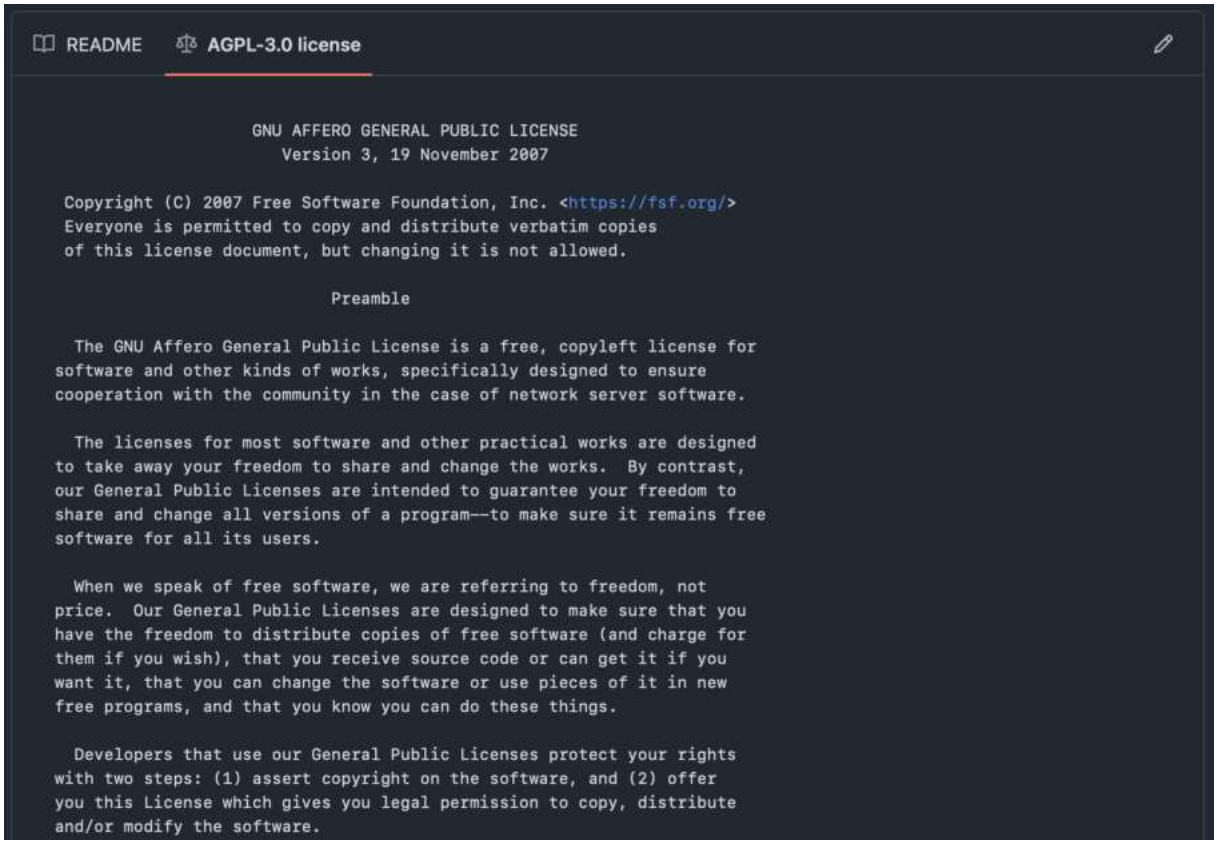
É notável que não basta a simples disponibilização pública do código-fonte; é preciso garantir a qualidade do código e uma documentação bem estruturada. Isso facilita a adoção do *driver* pela comunidade e permite que mais desenvolvedores possam contribuir com o projeto.

Para validar o critério `C1`, foi criado um repositório público no GitHub[2] seguindo as melhores práticas de projetos *open-source*. A escolha da Affero General Public License (AGPL) v3.0 se deu principalmente por esta se tratar da licença sob a qual o Metabase é distribuído originalmente. Além disso, a licença AGPL impõe que futuras modificações do código-fonte sejam disponibilizadas publicamente sob a mesma licença, garantindo que o

[1] https://github.com/metabase/metabase/releases/tag/v0.55.5.1
[2] https://github.com/jhisse/metabase-sparql-driver

driver continue sendo livre e aberto para todos. A figura 36 apresenta o arquivo de licença no repositório.

Figura 36 – Arquivo de licença do *driver* no GitHub



Fonte: https://github.com/jhisse/metabase-sparql-driver?tab=AGPL-3.0-1-ov-file

Além do arquivo de licença, foi criado um arquivo `README.md` onde são detalhadas todas as informações necessárias para a instalação e uso do *driver*. O arquivo possui instruções passo a passo, imagens ilustrativas e exemplos de consultas SPARQL, permitindo que novos usuários compreendam o funcionamento de maneira intuitiva. Essa documentação também serve para desenvolvedores que desejam contribuir com o projeto. Com esses recursos, é possível validar que o critério `C1` foi satisfeito.

6.1.2 Validação do Critério C11 - Distribuição como JAR Instalável

De acordo com a própria documentação do Metabase[3]:

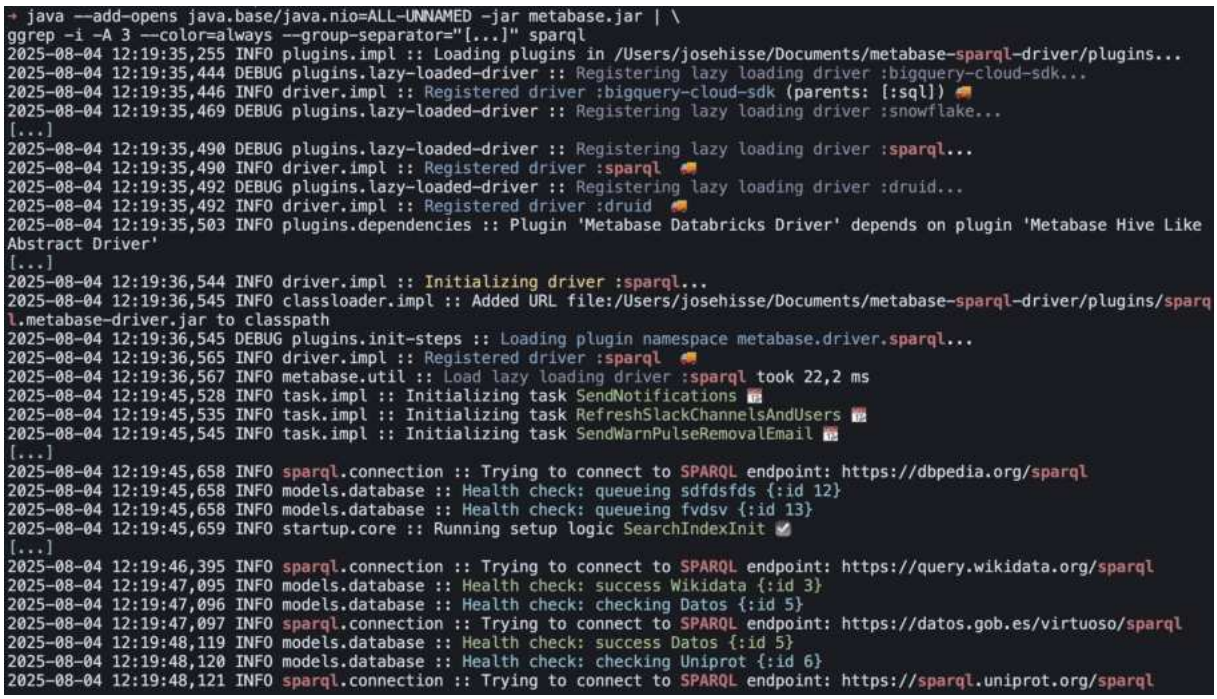
> *For Metabase to use your driver, all you need to do is put the driver JAR you built into the /plugin directory, which you'll find in the same directory where you run your metabase.jar.*

[3]https://www.metabase.com/docs/latest/developers-guide/drivers/basics

Ou seja, para que o Metabase possa utilizar o *driver*, basta que o usuário coloque o arquivo `JAR` no diretório `plugins/`, que deve estar no mesmo diretório que o `metabase.jar`, sendo este o arquivo principal da plataforma.

A partir do momento em que o `JAR` é adicionado ao diretório `plugins/` e o Metabase é inicializado, o *driver* deve inicializar sem lançar qualquer exceção. Além disso, a plataforma deve apresentar a mensagem confirmando que o *driver* foi carregado com sucesso.

Figura 37 – Mensagem do terminal de que o *driver* foi carregado com sucesso



Fonte: Captura de tela feita pelo autor

Nesse sentido, conclui-se que o critério `C11` foi atendido com sucesso, vide figura 37. Além disso, é possível verificar, nos *logs*, que o *driver* realizou, com sucesso, conexões com *endpoints* SPARQL.

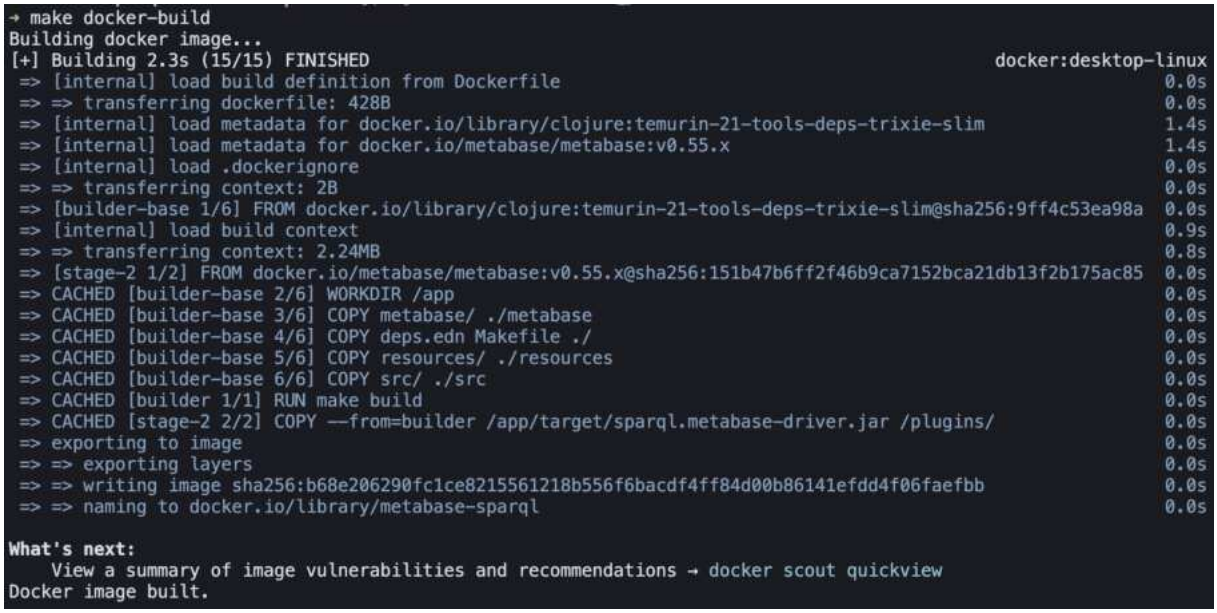
6.1.3 Validação do Critério C12 - Distribuição como imagem Docker

A abordagem de distribuir uma imagem *Docker* que possa ser usada para executar o Metabase junto ao *driver* é uma evolução significativa em relação à simples entrega do artefato `JAR`. Ela permite que usuários mais familiarizados com esta tecnologia possam utilizar o *driver* sem precisar se preocupar com a instalação do Metabase e do *driver*, bastando apenas executar o contêiner.

A combinação com o *Makefile* do projeto permitiu uma simplificação na construção da imagem *Docker*. Além disso, ela foi construída utilizando a abordagem de *multi-stage builds*, como apresentada na seção 5.6.7.

O *Dockerfile* está disponível no repositório do projeto[4] e, na figura 38, é possível verificar o processo de construção da imagem *Docker* no terminal, sendo executado o comando `make docker-build` para a construção da imagem. Conclui-se, assim, que o critério `C12` foi atendido com sucesso.

Figura 38 – Mensagem do terminal mostrando o processo de construção da imagem Docker



Fonte: Captura de tela feita pelo autor

Dessa forma, é possível concluir que todos os requisitos de avaliação referentes à disponibilização e instalação do *driver* foram atendidos de forma satisfatória. A combinação de uma licença aberta, a disponibilização do *driver* como JAR e como imagem *Docker*, além de uma documentação clara e detalhada, garante que o *driver* possa ser utilizado por uma ampla gama de usuários e que a sua adoção seja facilitada pela comunidade.

6.2 CONECTIVIDADE E DESCOBERTA DE METADADOS

O primeiro passo para o funcionamento correto do *driver* é o teste inicial de conectividade, ou seja, se o *driver* consegue se conectar a um servidor SPARQL com sucesso. Caso essa etapa não seja realizada com sucesso, o Metabase não permite nem que a nova fonte de dados seja adicionada. Portanto, o teste de conexão HTTP/HTTPS é fundamental para garantir que o *driver* funcione corretamente.

Logo após o teste de conectividade, o Metabase solicita ao *driver* que ele realize a descoberta de metadados. Essa etapa é fundamental para que o Metabase possa entender a estrutura do banco de dados e criar as tabelas e colunas necessárias para a visualização

[4] https://github.com/jhisse/metabase-sparql-driver/blob/main/Dockerfile

dos dados. Portanto, as validações a seguir verificarão os testes de conectividade com os *endpoints* e a descoberta de metadados sobre o grafo RDF.

6.2.1 Validação dos Critérios C2 e C3 - Conectividade via Protocolo HTTP/HTTPS

Foram utilizados, para os testes de conectividade, os *endpoints* apresentados previamente na tabela 8. Vale ressaltar que há 7 *endpoints* que suportam HTTPS/TLS e 1 opera com certificado TLS inválido. De acordo com os critérios de aceitação `C2` e `C3`, essa quantidade não foi um problema, pois todos os requisitos foram cobertos quando o *driver* conseguiu se conectar a todos os *endpoints*.

Figura 39 – Teste de conectividade com os 8 *endpoints* SPARQL

```
2025-08-04 17:41:39,575 DEBUG sparql.execute :: Endpoint: https://copper.coypu.org/coypu/
2025-08-04 17:41:39,575 DEBUG sparql.execute :: Ignore SSL validation: false
2025-08-04 17:41:39,575 DEBUG sparql.execute :: SPARQL query: ASK { }
2025-08-04 17:41:39,576 DEBUG sparql.execute :: SPARQL query execution time: 818 ms
2025-08-04 17:41:40,992 DEBUG sparql.execute :: Endpoint: https://dbpedia.org/sparql
2025-08-04 17:41:40,993 DEBUG sparql.execute :: Ignore SSL validation: false
2025-08-04 17:41:40,993 DEBUG sparql.execute :: SPARQL query: ASK { }
2025-08-04 17:41:40,993 DEBUG sparql.execute :: SPARQL query execution time: 2276 ms
2025-08-04 17:41:41,725 DEBUG sparql.execute :: Endpoint: https://query.wikidata.org/sparql
2025-08-04 17:41:41,726 DEBUG sparql.execute :: Ignore SSL validation: false
2025-08-04 17:41:41,726 DEBUG sparql.execute :: SPARQL query: ASK { }
2025-08-04 17:41:41,726 DEBUG sparql.execute :: SPARQL query execution time: 731 ms
2025-08-04 17:41:44,182 DEBUG sparql.execute :: Endpoint: https://datos.gob.es/virtuoso/sparql
2025-08-04 17:41:44,182 DEBUG sparql.execute :: Ignore SSL validation: true
2025-08-04 17:41:44,183 DEBUG sparql.execute :: SPARQL query: ASK { }
2025-08-04 17:41:44,183 DEBUG sparql.execute :: SPARQL query execution time: 2455 ms
2025-08-04 17:41:45,002 DEBUG sparql.execute :: Endpoint: https://sparql.uniprot.org/sparql
2025-08-04 17:41:45,002 DEBUG sparql.execute :: Ignore SSL validation: false
2025-08-04 17:41:45,002 DEBUG sparql.execute :: SPARQL query: ASK { }
2025-08-04 17:41:45,002 DEBUG sparql.execute :: SPARQL query execution time: 817 ms
2025-08-04 17:41:45,923 DEBUG sparql.execute :: Endpoint: https://linkedgeodata.org/sparql
2025-08-04 17:41:45,923 DEBUG sparql.execute :: Ignore SSL validation: false
2025-08-04 17:41:45,923 DEBUG sparql.execute :: SPARQL query: ASK { }
2025-08-04 17:41:45,923 DEBUG sparql.execute :: SPARQL query execution time: 918 ms
2025-08-04 17:41:46,246 DEBUG sparql.execute :: Endpoint: https://agrovoc.fao.org/sparql
2025-08-04 17:41:46,247 DEBUG sparql.execute :: Ignore SSL validation: false
2025-08-04 17:41:46,247 DEBUG sparql.execute :: SPARQL query: ASK { }
2025-08-04 17:41:46,247 DEBUG sparql.execute :: SPARQL query execution time: 322 ms
2025-08-04 17:41:47,030 DEBUG sparql.execute :: Endpoint: https://copper.coypu.org/coypu/
2025-08-04 17:41:47,030 DEBUG sparql.execute :: Ignore SSL validation: false
2025-08-04 17:41:47,030 DEBUG sparql.execute :: SPARQL query: ASK { }
2025-08-04 17:41:47,031 DEBUG sparql.execute :: SPARQL query execution time: 781 ms
```

Fonte: Captura de tela feita pelo autor

A partir da figura 39, é possível verificar que o *driver* conseguiu se conectar a todos os *endpoints* com sucesso, validando o critério `C2` com sucesso. Cabe destacar que o *endpoint* `https://datos.gob.es/virtuoso/sparql` tem o certificado TLS inválido, mas o *driver* conseguiu se conectar a ele, cobrindo assim o critério `C3` com sucesso.

6.2.2 Validação do Critério C4 - Descoberta Automática de Metadados

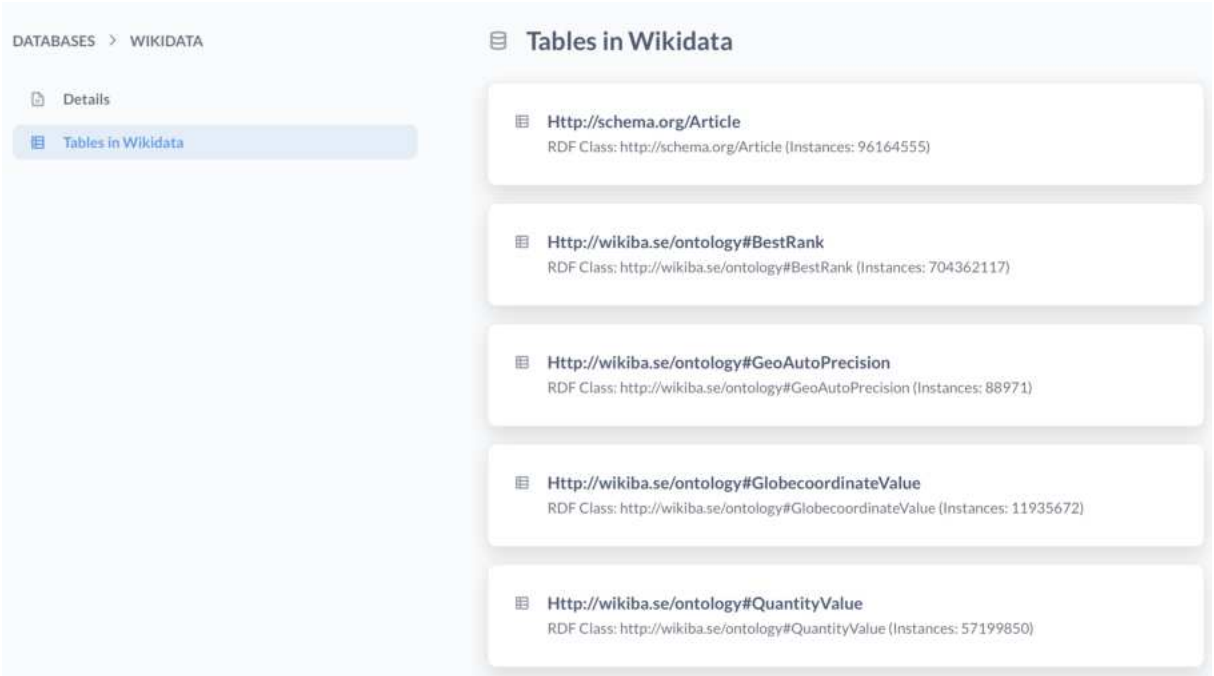
Nesta etapa, o mapeamento grafo-tabela é avaliado, como proposto na seção 2.5. Para satisfazer o critério `C4`, é necessário que o *driver* seja capaz de mapear as classes mais frequentes do grafo RDF, assim como as propriedades mais frequentes de cada classe.

Todo o mapeamento é realizado durante a fase de sincronização de metadados, momento em que o Metabase solicita ao *driver* que execute essa tarefa e repasse o resultado para que a plataforma possa armazená-lo internamente. Esses metadados são utilizados na interface do *query builder* para exibir opções de tabelas e colunas ao usuário. Vale ressaltar que a funcionalidade de desativar a sincronização de metadados implica na não exibição de tabelas e colunas na interface do Metabase, além de não fornecer metadados suficientes para o *query builder* realizar consultas.

É importante notar que a recuperação das classes e propriedades mais frequentes de um grafo impõe uma limitação no *query builder*, pois nem todas as classes e propriedades do grafo são exibidas como opções de consulta. Ainda assim, essa restrição ocorre apenas na interface visual, já que o *driver* continua apto a executar consultas SPARQL em qualquer classe ou propriedade do grafo por meio de consultas nativas.

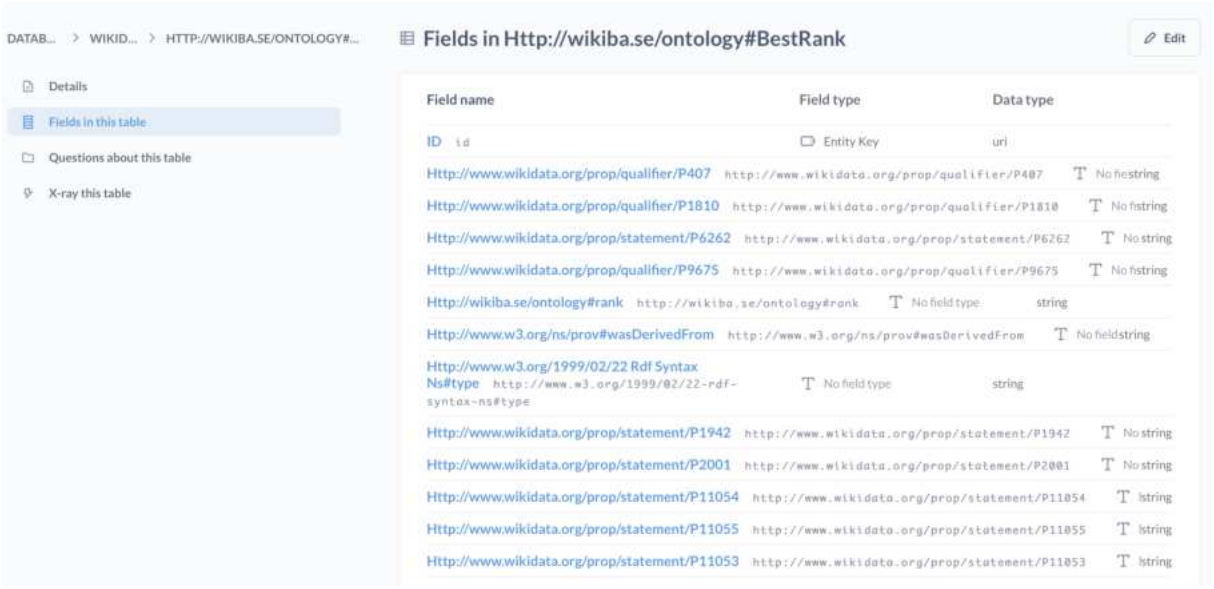
Na figura 40, é possível observar que o *driver* conseguiu mapear as classes mais frequentes do grafo RDF, enquanto na figura 41 são apresentadas as propriedades mais comuns de cada classe. Enquanto a figura 40 mostra as classes como tabelas na interface do Metabase, a figura 41 mostra as propriedades como colunas.

Figura 40 – Mapeamento de classes como tabelas na interface do Metabase



Fonte: Captura de tela feita pelo autor

Figura 41 – Mapeamento de propriedades como colunas na interface do Metabase



Fonte: Captura de tela feita pelo autor

Conforme evidenciado pela figura 40 e figura 41, o *driver* conseguiu mapear corretamente as classes e propriedades mais frequentes do grafo RDF, atendendo com sucesso ao critério `C4`. Isso demonstra que o processo de descoberta de metadados foi implementado de forma a permitir que o Metabase possa utilizar essas informações no *query builder*. O mapeamento também contribui para a usabilidade da interface, permitindo ao usuário visualizar um subconjunto dos elementos do grafo baseado nos usos mais frequentes, evitando a sobrecarga visual que ocorreria com a exibição completa de todas as classes e propriedades disponíveis em grandes grafos. Além disso, o fato de o mapeamento ter sido obtido de forma automatizada elimina a necessidade de intervenções manuais para configurar as tabelas e colunas a serem apresentadas na interface.

6.3 EXECUÇÃO DE CONSULTAS SPARQL

A capacidade do *driver* de executar SPARQL é de extrema importância neste trabalho, pois viabiliza a criação de visualizações e *dashboards*. Nesta seção, verificamos os critérios relacionados à execução: `ASK` e `SELECT`, conversão de tipos de dados e parametrização.

6.3.1 Validação do Critério C5 - Execução de Consultas `ASK`

Como já relatado em seção 2.4, o suporte à consulta do tipo `ASK` é fundamental para verificar se determinado grafo RDF possui uma tripla, ou um conjunto de triplas, que satisfaça as condições definidas. Para validar o critério `C5`, é necessário que o *driver* execute esse tipo de operação, que retorna verdadeiro ou falso.

Foram realizadas cinco consultas, com estruturas e filtros distintos, distribuídas em três *endpoints* diferentes, que são apresentadas no apêndice A. Na figura 42, observa-se um exemplo, no qual, utilizando o *endpoint* da DBpedia, é verificado se a UFRJ possui mais de 10 mil alunos de graduação. O resultado obtido é verdadeiro, condizente com o fato representado.

Figura 42 – Consulta `ASK` que verifica na DBpedia se a UFRJ tem mais de 10 mil alunos

Fonte: Captura de tela feita pelo autor

Durante os testes, todas as consultas foram processadas corretamente pelo *driver*. A tabela 9 resume os sucessos por *endpoint* e as principais observações. Além disso, o simples retorno verdadeiro ou falso valida a conexão com os *endpoints* SPARQL e a correta interpretação do JSON de resposta. Assim, o critério `C5` foi atingido com sucesso.

Tabela 9 – Resumo da validação de C5 por *endpoint*

Endpoint	**Consultas**	**Sucesso**	**Observações**
UniProt	1	1	`ASK` por criação em data específica (`xsd:date`).
AgroVoc	2	2	Filtros com `LCASE`/`STR` sobre rótulos.
DBpedia	2	2	Uso literal com *datatype* (`xsd:date`) e filtro numérico (`>`).

Fonte: Elaborado pelo autor

6.3.2 Validação do Critério C6 - Execução de Consultas `SELECT`

Para validar o critério `C6`, foram executadas cinco consultas do tipo `SELECT` em cinco *endpoints* SPARQL: Wikidata, OSM - Sophox, CoyPu, UniProt e AgroVoc. O objetivo foi verificar os resultados, o processamento correto pelo *driver* e a coerência dos tipos de dados com o retorno esperado (esse último tendo interseção com `C7`). A tabela 10 resume o número de execuções por *endpoint*, os sucessos e observações relevantes.

Tabela 10 – Resumo da validação de C6 por *endpoint*

Endpoint	**Consultas**	**Sucessos**	**Observações**
Wikidata	1	1	Uso de `SERVICE wikibase:label`; ordenação por população; tipos numéricos reconhecidos pelo Metabase.
OSM Sophox	1	1	Funções `CONTAINS` e `REPLACE` com `BIND`; retorno textual.
CoyPu	1	1	`GROUP_CONCAT` e `COUNT(DISTINCT)`; `GROUP BY` por entidade com múltiplos rótulos.
UniProt	1	1	Agregação por fonte com `COUNT`; ordenação descendente com estabilidade de metadados.
AgroVoc	1	1	Seleção com filtros e rótulos; *strings* com definição de idioma; resultados reprodutíveis com `LIMIT`.

Fonte: Elaborado pelo autor

As consultas realizadas (apêndice B) cobriram cenários utilizados em contextos reais: projeção simples, filtros com `FILTER`, ordenação com `ORDER BY`, limitação de resultados com `LIMIT`, agregações com `COUNT` e `GROUP BY`, concatenação de rótulos com `GROUP_CONCAT`, além de funções de manipulação de texto `REPLACE`. Embora a execução das consultas valide a capacidade dos *endpoints* SPARQL, foi possível verificar que o *driver* se comportou da maneira esperada na execução de todas as consultas de teste.

As 5 consultas do tipo `SELECT` propostas como cenário de testes tiveram seu resultado renderizado com sucesso na interface do Metabase. Portanto, é seguro dizer que o *driver* cumpre com sucesso o critério `C6`.

6.3.3 Validação do Critério C7 - Conversão de Tipos de Dados

A conversão correta dos tipos de dados provenientes do retorno JSON para os tipos aceitos pelo Metabase é fundamental para o funcionamento adequado das visualizações na ferramenta. Conforme discutido na seção 5.6.5, valores literais retornados nas consultas podem trazer um *datatype* associado, indicando como aquele valor deve ser interpretado. Por isso, mapear, de maneira precisa, os tipos definidos pelo SPARQL para os tipos nativos do Metabase é essencial para garantir a coerência dos dados apresentados e a criação de gráficos adequados.

Para validar o critério `C7`, foram utilizadas três abordagens. A primeira foi a verificação visual, diretamente pela interface do Metabase, checando se valores como inteiros, decimais, booleanos, datas e IRIs estavam sendo reconhecidos e exibidos corretamente. Em seguida, foi feita uma análise dos *logs* de *debug* gerados pelo *driver*, observando como o processo de conversão estava sendo realizado de forma granular no código, possibilitando identificar a conversão de cada registro em específico. A figura 43 mostra um trecho dos *logs*, no qual é possível acompanhar o mapeamento do tipo `xsd:date` retornado pelo *endpoint* SPARQL para o tipo de data reconhecido pelo Metabase.

Figura 43 – *Logs* de *debug* para a função de conversão

```
DEBUG sparql.conversion :: sparql-type: uri, datatype: null -> base-type: :type/URL
DEBUG sparql.conversion :: sparql-type: uri, datatype: null -> base-type: :type/URL
DEBUG sparql.conversion :: sparql-type: uri, datatype: null -> base-type: :type/URL
DEBUG sparql.conversion :: sparql-type: uri, datatype: null -> base-type: :type/URL
DEBUG sparql.conversion :: sparql-type: uri, datatype: null -> base-type: :type/URL
DEBUG sparql.conversion :: sparql-type: uri, datatype: null -> base-type: :type/URL
DEBUG sparql.conversion :: sparql-type: uri, datatype: null -> base-type: :type/URL
DEBUG sparql.conversion :: sparql-type: uri, datatype: null -> base-type: :type/URL
DEBUG sparql.conversion :: sparql-type: uri, datatype: null -> base-type: :type/URL
DEBUG sparql.conversion :: sparql-type: uri, datatype: null -> base-type: :type/URL
DEBUG sparql.conversion :: sparql-type: literal, datatype: null -> base-type: :type/Text
DEBUG sparql.conversion :: sparql-type: literal, datatype: null -> base-type: :type/Text
DEBUG sparql.conversion :: sparql-type: literal, datatype: null -> base-type: :type/Text
DEBUG sparql.conversion :: sparql-type: literal, datatype: null -> base-type: :type/Text
DEBUG sparql.conversion :: sparql-type: literal, datatype: null -> base-type: :type/Text
DEBUG sparql.conversion :: sparql-type: literal, datatype: null -> base-type: :type/Text
DEBUG sparql.conversion :: sparql-type: literal, datatype: null -> base-type: :type/Text
DEBUG sparql.conversion :: sparql-type: literal, datatype: null -> base-type: :type/Text
DEBUG sparql.conversion :: sparql-type: literal, datatype: null -> base-type: :type/Text
DEBUG sparql.conversion :: sparql-type: literal, datatype: null -> base-type: :type/Text
DEBUG sparql.conversion :: sparql-type: typed-literal, datatype: http://www.w3.org/2001/XMLSchema#date -> base-type: :type/Date
DEBUG sparql.conversion :: sparql-type: typed-literal, datatype: http://www.w3.org/2001/XMLSchema#date -> base-type: :type/Date
DEBUG sparql.conversion :: sparql-type: typed-literal, datatype: http://www.w3.org/2001/XMLSchema#date -> base-type: :type/Date
DEBUG sparql.conversion :: sparql-type: typed-literal, datatype: http://www.w3.org/2001/XMLSchema#date -> base-type: :type/Date
DEBUG sparql.conversion :: sparql-type: typed-literal, datatype: http://www.w3.org/2001/XMLSchema#date -> base-type: :type/Date
DEBUG sparql.conversion :: sparql-type: typed-literal, datatype: http://www.w3.org/2001/XMLSchema#date -> base-type: :type/Date
DEBUG sparql.conversion :: sparql-type: typed-literal, datatype: http://www.w3.org/2001/XMLSchema#date -> base-type: :type/Date
DEBUG sparql.conversion :: sparql-type: typed-literal, datatype: http://www.w3.org/2001/XMLSchema#date -> base-type: :type/Date
DEBUG sparql.conversion :: sparql-type: typed-literal, datatype: http://www.w3.org/2001/XMLSchema#date -> base-type: :type/Date
DEBUG sparql.conversion :: sparql-type: typed-literal, datatype: http://www.w3.org/2001/XMLSchema#date -> base-type: :type/Date
DEBUG sparql.conversion :: sparql-type: typed-literal, datatype: http://www.w3.org/2001/XMLSchema#date -> base-type: :type/Date
```

Fonte: Captura de tela feita pelo autor

Por fim, foram desenvolvidos testes unitários para a função de conversão, levando em conta diferentes cenários, incluindo casos de ausência de *datatype* explícito ou recebimento de tipos não suportados pelo Metabase, que, neste último caso, foram interpretados como texto comum. Ainda na figura 43, é possível observar a conversão desse tipo textual, além da conversão de IRIs para URLs.

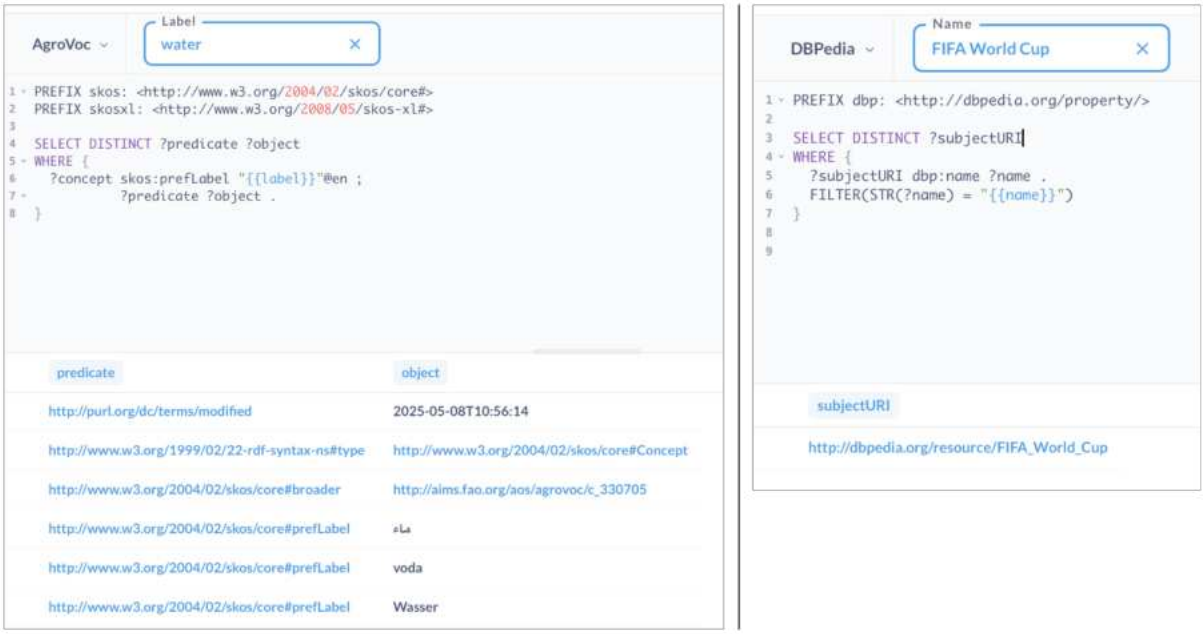
As três abordagens demonstram de maneira complementar que os tipos estão sendo convertidos corretamente e que os dados representados na interface visual respeitam o tipo original retornado pelo *endpoint* SPARQL. Com isso, o critério `C7` é considerado atendido com sucesso, garantindo a conversão correta dos tipos de dados na interface visual.

6.3.4 Validação do Critério C8 - Execução de Consultas com Parâmetros

Como mostrado na seção 5.6.5, o Metabase interpreta parâmetros definidos na consulta por meio da sintaxe "`{{...}}`". A responsabilidade de substituir esses parâmetros é da função `substitute-native-parameters`, que atua na etapa anterior à execução da consulta final. Para validar o critério `C8`, foram utilizados dois métodos complementares: a execução de consultas parametrizadas diretamente pela interface do Metabase e a execução de testes unitários.

Para os testes via interface, foram utilizadas duas consultas que incluem parâmetros que deveriam ser substituídos por valores fornecidos pelo usuário, como mostra a figura 44. Era esperado que cada uma dessas consultas retornasse pelo menos uma linha como resultado. Caso contrário, o critério deveria ser considerado não atingido, pois indicaria falha na substituição ou na execução da consulta com o parâmetro fornecido.

Figura 44 – Dois exemplos de consultas SPARQL utilizando parâmetros



Fonte: Capturas de tela feitas pelo autor

Já, nos testes unitários, o foco foi validar a lógica da função de substituição, como mostra um teste específico no código 22. Para isso, foram utilizadas consultas simples contendo parâmetros, e o objetivo não foi executá-las, mas sim verificar se, após o processamento, a consulta gerada com os valores interpolados corresponde exatamente ao esperado. Assim, mesmo sem o envio da consulta SPARQL ao *endpoint* SPARQL, é possível garantir que a substituição dos parâmetros ocorre corretamente.

Código 22 – Trecho de um dos testes unitários da função `substitute-native-parameters`

```
(deftest test-substitute-native-parameters
  (testing "Handle single parameter substitution"
    (let [inner-query {
      :query "SELECT * WHERE { ?s rdfs:label {{name}} }"
      :template-tags {"name" {:name "name"
                              :display-name "Name"
                              :type "text"}}
      :parameters [{:type "text"
                    :target [:variable [:template-tag "name"]]
                    :value "\"Alice\""}]}
          result (parameters/substitute-native-parameters :sparql
              inner-query)]
      (is (= "SELECT * WHERE { ?s rdfs:label \"Alice\" }" (:query
          result))))))
```

Enquanto os testes via interface garantem que a funcionalidade está disponível e funcionando para o usuário final, os testes unitários checam a corretude da implementação responsável por processar as consultas parametrizadas. A combinação dessas abordagens permite afirmar que o *driver* está funcional no que diz respeito a lidar com parâmetros em consultas SPARQL, atendendo ao critério `C8`.

6.4 VALIDAÇÃO DO CRITÉRIO C9 - VISUALIZAÇÕES E GRÁFICOS

Esta seção apresenta as evidências visuais obtidas a partir da execução de consultas SPARQL no Metabase. O objetivo é demonstrar que a plataforma é capaz de exibir diferentes tipos de visualizações a partir dos resultados tabulares retornados pelos *endpoints*. Serão apresentadas as visualizações mais comuns que são utilizadas em análises exploratórias, cada uma acompanhada de breve descrição do propósito analítico e dos códigos das consultas correspondentes.

O critério `C9` foi validado por meio da criação de oito visualizações no Metabase a partir de consultas SPARQL pré-definidas. Conforme apresentado no capítulo 4 e ilustrado em figura 18, a ferramenta suporta diferentes tipos de gráficos aplicáveis a cenários analíticos variados. Para permitir reprodução, as consultas utilizadas se encontram em apêndice C.

Primeiramente, a visualização do tipo *tabela* foi empregada para conferir os resultados. Foi executada uma consulta (código 33 sobre a Wikidata) para retornar eventos recentes, com colunas nomeadas e valores de tipos distintos, a fim de verificar a correspondência entre variáveis e colunas exibidas no Metabase. Essa verificação inicial visa garantir que os dados sejam apresentados corretamente na interface antes da aplicação de agregações ou filtros. Foram adotadas ordenação temporal e limitação de linhas, quando apropriado, para facilitar a leitura dos resultados. A figura 45 mostra o resultado em formato de tabela.

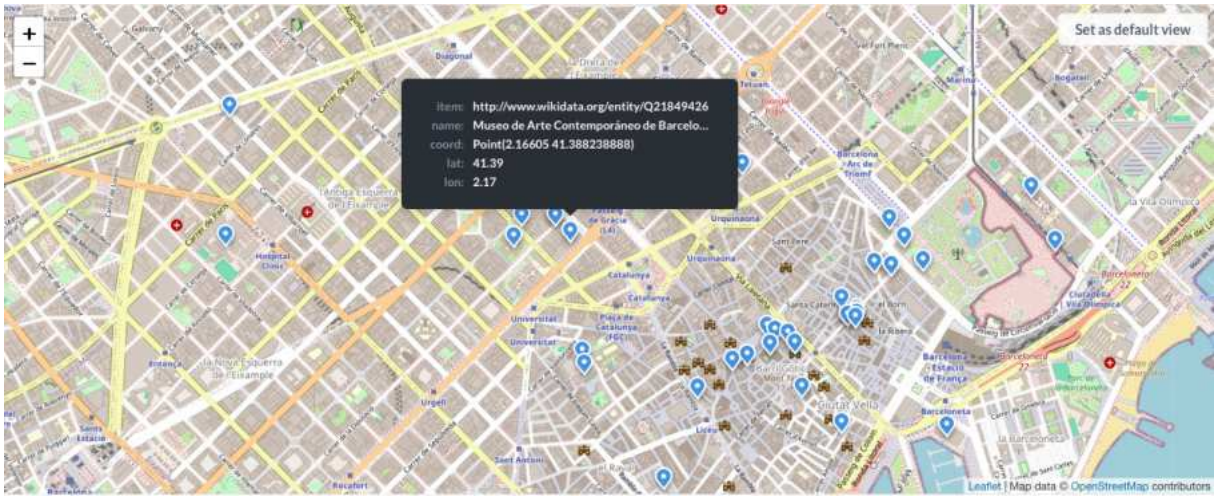
Figura 45 – Tabela de eventos recentes obtidos via consulta SPARQL

event	eventLabel	date
http://www.wikidata.org/entity/Q135488475	Coding History. Perspektiven einer digitalen Landesgeschichte	August 4, 2025
http://www.wikidata.org/entity/Q135653464	Tropical Storm Dexter	August 4, 2025
http://www.wikidata.org/entity/Q135644172	2025 Sydney Harbour Bridge protest	August 3, 2025
http://www.wikidata.org/entity/Q134126878	Miss Grand Nicaragua 2025	August 3, 2025
http://www.wikidata.org/entity/Q135012548	Logie Awards of 2025	August 3, 2025
http://www.wikidata.org/entity/Q135617477	Badminton at Die Finals – Dresden 2025	August 3, 2025
http://www.wikidata.org/entity/Q135640719	Swimming at the 2025 World Aquatics Championships – Women's 4 × 100 metre medley relay	August 3, 2025
http://www.wikidata.org/entity/Q135640675	Swimming at the 2025 World Aquatics Championships – Men's 4 × 100 metre medley relay	August 3, 2025
http://www.wikidata.org/entity/Q135483428	Swimming at the 2025 World Aquatics Championships – Women's 400 metre individual medl...	August 3, 2025
http://www.wikidata.org/entity/Q135626336	Pro dvizheniye	August 2, 2025
http://www.wikidata.org/entity/Q135603776	2025 Copa América Femenina Final	August 2, 2025

Fonte: Captura de tela feita pelo autor com o resultado da consulta em código 33

Em seguida, o *mapa de pontos* foi utilizado para visualizar a distribuição geográfica de museus em Barcelona (usando a consulta apresentada no código 34 sobre a Wikidata). As coordenadas (latitude/longitude) foram extraídas diretamente da consulta, permitindo a localização precisa no mapa. Essa visualização facilita perceber a distribuição espacial dos elementos, além de possibilitar inspeção pontual por meio de interações do usuário. A figura 46 apresenta o resultado em um recorte do mapa da cidade de Barcelona.

Figura 46 – Mapa de pontos - museus em Barcelona



Fonte: Captura de tela feita pelo autor com o resultado da consulta em código 34

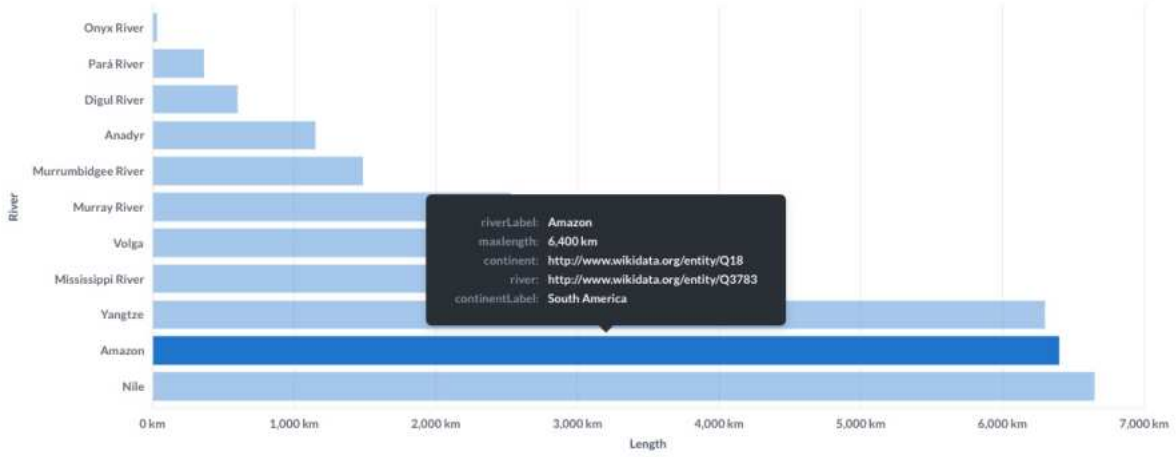
Para o gráfico do tipo *número único*, foi utilizada uma consulta que agrega o total de isoformas de um proteoma no *endpoint* UniProt (código 35). Esse indicador destaca um valor único, útil para representar um agregado que não seja uma dimensão categórica. A figura 47 mostra o resultado com o total de isoformas de um proteoma específico.

Figura 47 – Número único - total de isoformas

Fonte: Captura de tela feita pelo autor com o resultado da consulta em código 35

O gráfico de *barras* horizontal foi gerado a partir da consulta apresentada no código 36, sobre a Wikidata, comparando os rios mais longos de cada sub-região do planeta. A ordenação decrescente e a limitação de categorias favorecem a legibilidade quando há cauda longa de valores. A figura 48 apresenta o resultado renderizado como barras horizontais.

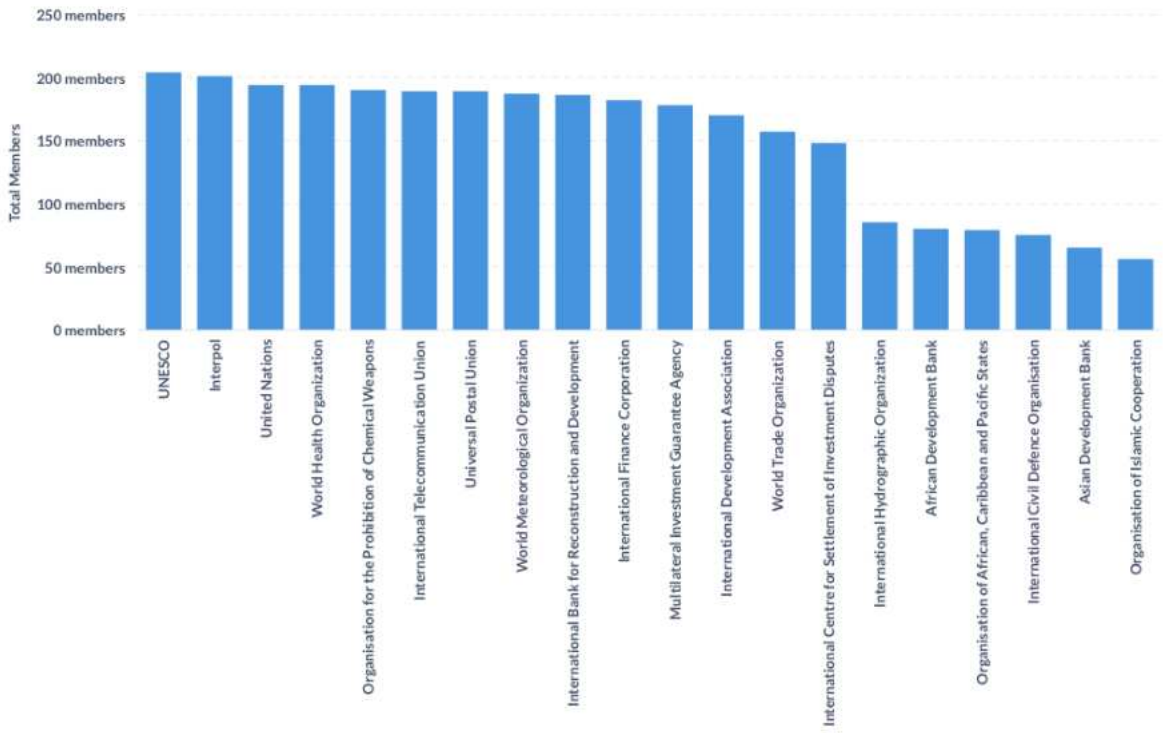
Figura 48 – Barras horizontais - rios mais longos por sub-região do planeta



Fonte: Captura de tela feita pelo autor com o resultado da consulta em código 36

O gráfico de *barras verticais* é apropriado para comparar categorias, como o total de membros por organização internacional (consulta apresentada no código 37 realizada no *endpoint* CoyPu). Mesmo com nomes longos das categorias, neste caso nomes das organizações internacionais, a visualização mantém boa legibilidade, como mostra a figura 49.

Figura 49 – Barras verticais - total de membros por organização internacional



Fonte: Captura de tela feita pelo autor com o resultado da consulta em código 37

Gráficos do tipo *linha* podem representar tendências ao longo do tempo. Foi utilizada

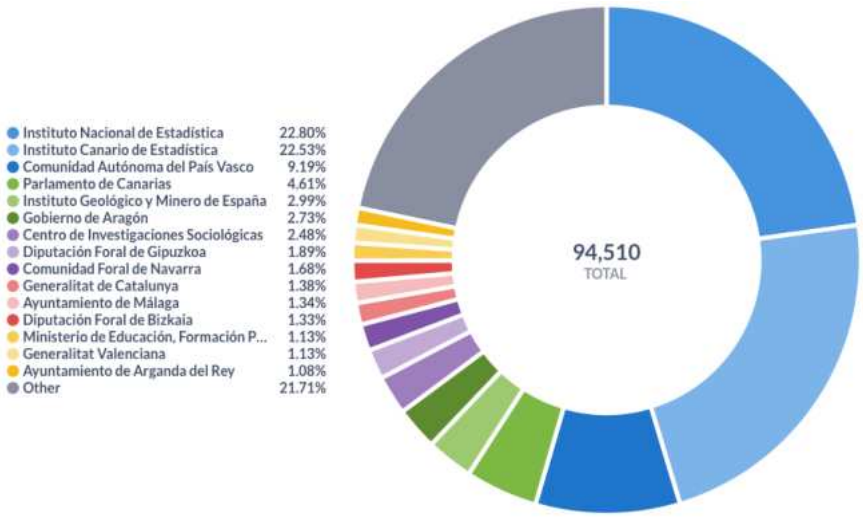
uma série temporal de população por ano (usando a consulta apresentada no código 38 sobre a Wikidata), destacando a evolução populacional do Suriname ao longo de décadas. A granularidade anual e a ordenação temporal tornam a variação e as tendências de longo prazo mais evidentes, como mostra a figura 50.

Figura 50 – Linha - evolução da população do Suriname ao longo do tempo

Fonte: Captura de tela feita pelo autor com o resultado da consulta em código 38

Para relações parte–todo, foi utilizado o gráfico de *rosca*, representando a distribuição em porcentagem de entidades que publicaram conjuntos de dados, resultado da consulta no *endpoint* de dados abertos da Espanha (código 39). A figura 51 apresenta o resultado obtido a partir da consulta.

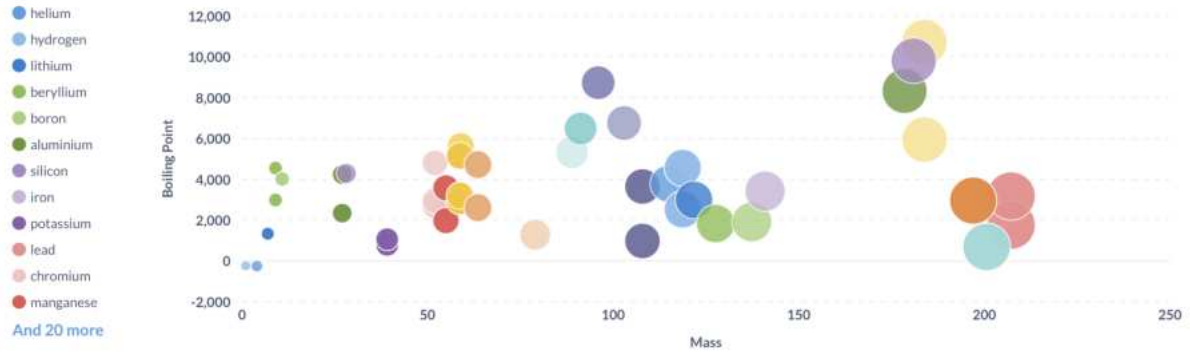
Figura 51 – Rosca - distribuição de publicadores de conjuntos de dados



Fonte: Captura de tela feita pelo autor com o resultado da consulta em código 39

Por fim, foi utilizado o gráfico do tipo *bolhas*. No Metabase, são necessárias três variáveis: uma categórica para identificar cada entidade (eixo X) e duas métricas (eixos Y e tamanho da bolha). Na figura 52, é mostrada a relação entre a massa de um elemento químico (eixo X) e seu ponto de ebulição (eixo Y), permitindo comparar os elementos e avaliar possíveis correlações. Ressaltando que a consulta utilizada para gerar o gráfico está disponível no código 40, feita sobre a Wikidata.

Figura 52 – Bolhas - relação entre massa e ponto de ebulição



Fonte: Captura de tela feita pelo autor com o resultado da consulta em código 40

Em resumo, o conjunto de visualizações apresentadas: tabela, mapa de pontos, número único, barras (horizontal e vertical), linha, rosca e bolhas, evidenciam que resultados tabulares com origem em consultas SPARQL podem ser explorados no Metabase sem adaptações adicionais, satisfazendo o critério de aceitação `C9`. A diversidade de gráficos, juntamente com a conversão consistente de tipos (`C7`), viabiliza análises comparativas em diferentes domínios. Além disso, a partir das consultas documentadas em apêndice C, é possível reproduzir os resultados obtidos.

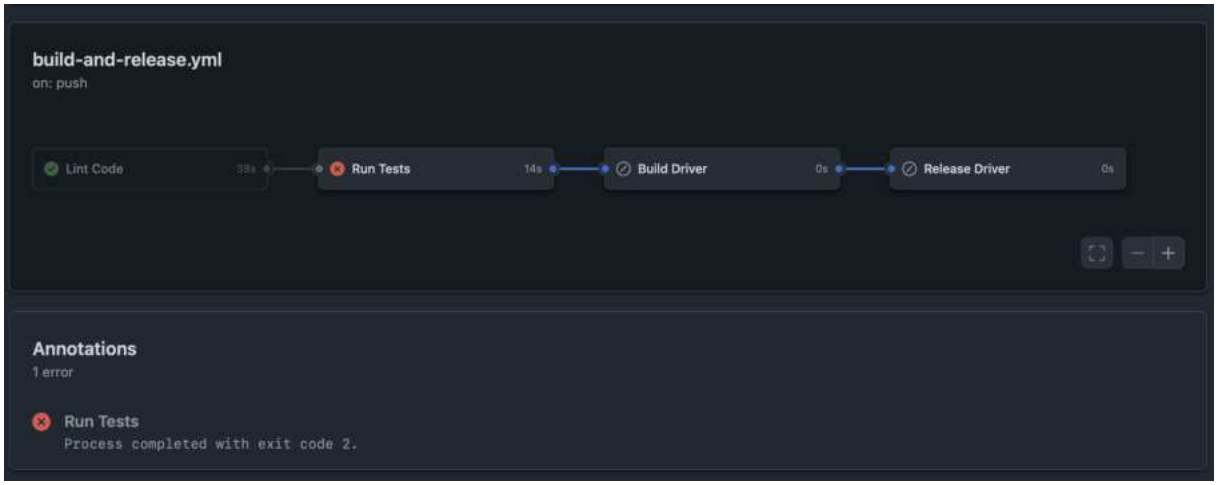
6.5 VALIDAÇÃO DO CRITÉRIO C10 - INTEGRAÇÃO CONTÍNUA E QUALIDADE DO CÓDIGO

Esta seção aborda as estratégias que foram adotadas para manter a qualidade do código-fonte durante o desenvolvimento do *driver*. O fluxo automático de integração contínua, implementado via GitHub Actions, facilita a manutenção do repositório público e assegura que o código permaneça minimamente organizado, seguindo boas práticas de desenvolvimento.

O critério `C10` busca garantir a qualidade do código do *driver*. Toda vez que uma nova contribuição é enviada ao repositório, o fluxo de integração contínua deve executar com sucesso todas as etapas previstas: verificação de sintaxe; verificação de boas práticas; execução de testes unitários e empacotamento do artefato final.

Foram realizados diversos *commits* ao longo do desenvolvimento do projeto, os quais acionaram automaticamente o fluxo de CI. É possível verificar, por meio da figura 53, que o processo foi interrompido corretamente quando os testes unitários falharam. Esse comportamento pode ser interpretado como algo positivo, pois demonstra que o fluxo impediu que um artefato com falhas fosse disponibilizado ao público.

Figura 53 – Falha no teste unitário interrompendo o fluxo de CI



Fonte: https://github.com/jhisse/metabase-sparql-driver/actions/runs/16781430072

Da mesma forma, a figura 54 mostra que um trecho do código poderia ser simplificado. A ferramenta Splint identificou corretamente essa oportunidade de melhoria e bloqueou a continuação do fluxo até que a sugestão fosse incorporada.

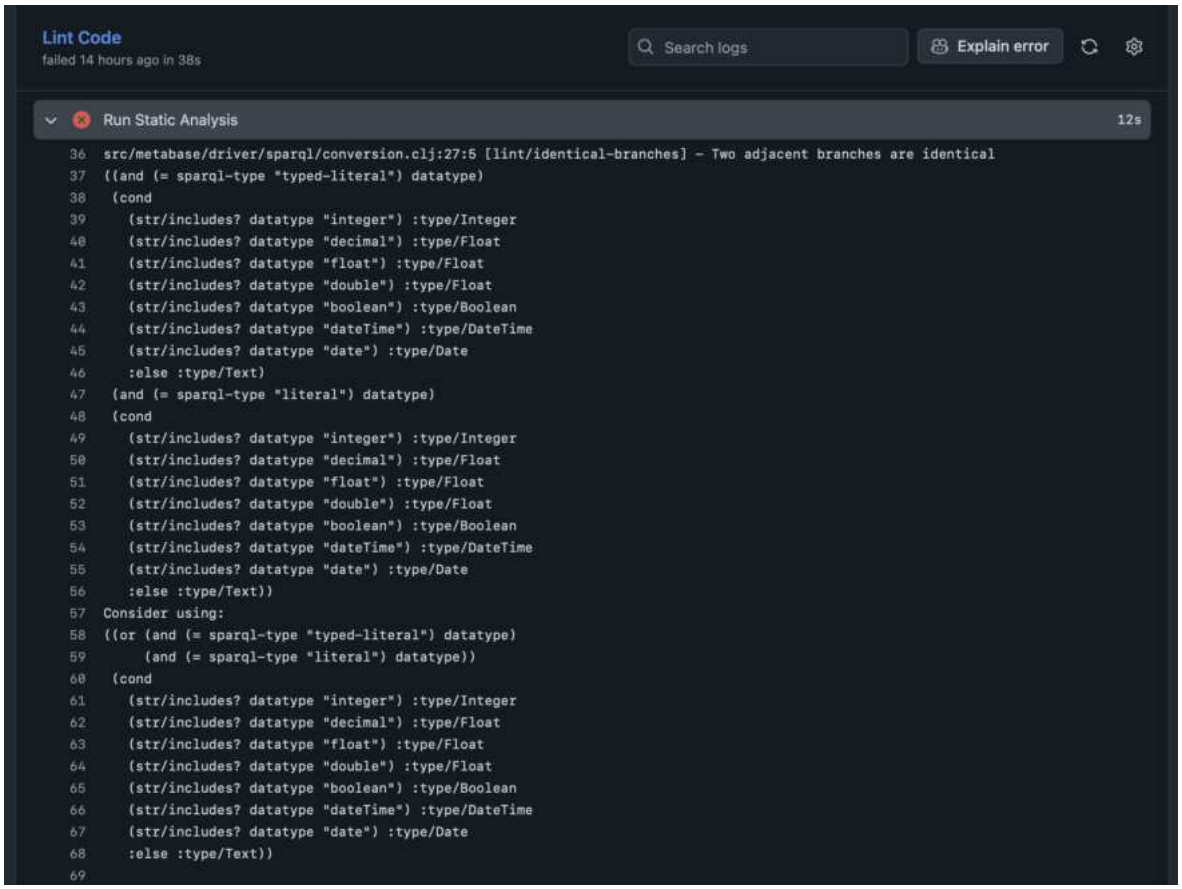
Os testes feitos reforçam que o fluxo está configurado para atuar como um filtro, garantindo que apenas contribuições válidas sejam distribuídas ao público. Com isso, o critério `C10` está plenamente atendido, garantindo a entrega contínua do *driver*.

6.6 DISCUSSÃO DOS RESULTADOS E CONSIDERAÇÕES FINAIS

Esta seção apresenta os resultados obtidos de acordo com os objetivos propostos e os critérios de aceitação definidos na seção 5.2. Em linhas gerais, as evidências apresentadas ao longo deste capítulo indicam que o *driver* atendeu aos requisitos mínimos para uso prático no Metabase, permitindo a conexão a *endpoints* SPARQL, a execução de consultas nativas dos tipos `SELECT` e `ASK`, o mapeamento grafo-tabela para exibição no *query builder*, a conversão de tipos de dados para formatos nativos do Metabase e a construção de visualizações diversas a partir dos resultados.

Conforme apresentado nas seções anteriores, os testes de conectividade (`C2` e `C3`) demonstraram que o *driver* estabelece comunicação com oito *endpoints* públicos, incluindo um servidor com certificado TLS inválido, atendendo aos cenários previstos. Em seguida, a descoberta de metadados (`C4`) possibilitou expor de forma automática as classes mais

Figura 54 – Sugestão de melhoria feita pela ferramenta Splint



Fonte: https://github.com/jhisse/metabase-sparql-driver/actions/runs/16611937588/job/46996548523

frequentes como “tabelas” e seus predicados mais frequentes como “colunas”, respeitando a estratégia de mapeamento discutida na seção 2.5. A execução de consultas `ASK` e `SELECT` (`C5` e `C6`) ocorreu conforme o esperado, com correta interpretação do retorno em JSON nas variantes das versões 1.0 e 1.1 da especificação SPARQL, e com conversão de tipos (`C7`) compatível com os tipos básicos definidos por XSD, conforme detalhado em seção 5.6.5, e os demais tipos não suportados são tratados como texto literal. Também foi validada a parametrização nativa (`C8`), permitindo reutilizar consultas com diferentes filtros.

Avaliando a entrega e a reprodutibilidade, foram satisfeitos os critérios no que diz respeito à licença e à disponibilização pública (`C1`), à instalação via artefato JAR (`C11`) e à disponibilização de imagem *Docker* (`C12`), o que facilita a adoção por diferentes perfis de usuários. Por fim, o fluxo de integração contínua (`C10`) contribuiu para a garantia de qualidade por meio de verificações estáticas do código, testes unitários e empacotamento automatizado, reduzindo a probabilidade de propagação de erros e facilitando contribuições futuras por parte de outros pesquisadores.

Apesar dos resultados positivos, algumas limitações foram identificadas ao longo da avaliação:

- **Custo de agregações para metadados (classe e predicados):** a obtenção das classes e predicados mais frequentes depende de consultas de agregação muitas vezes custosas computacionalmente em grafos volumosos. Como discutido, isso pode impactar o tempo de sincronização de metadados, especialmente em *endpoints* com recursos computacionais limitados. Para mitigar, foi disponibilizada a opção de desativar a sincronização de metadados (figura 26), porém essa escolha traz como consequência a incapacidade de utilizar o *query builder*, limitando o usuário apenas às consultas SPARQL nativas, ou seja, aquelas efetuadas diretamente no editor de consultas, exigindo que os analistas já tenham conhecimento do grafo.

- **Suporte parcial no *query builder*:** a exibição é intencionalmente limitada às classes e propriedades mais frequentes de acordo com a estratégia de obter classes e predicados mais frequentes. Embora essa escolha favoreça a usabilidade e o desempenho, ela impõe uma visão parcial das informações do grafo na interface; consultas nativas continuam sem restrições.

- **Variações de *endpoints*:** as diferenças de suporte a funcionalidades do SPARQL entre servidores na Web são amplamente reportadas (BUIL-ARANDA et al., 2013; LEHMANN et al., 2017). Essa variabilidade pode impactar recursos específicos, como, por exemplo, os recursos do *query builder*, o que reforça a decisão deste trabalho de focar em consultas nativas, funções básicas na conversão MBQL $\rightarrow$ SPARQL e em verificações pontuais de suporte, como foi feita com a função `NOW()`.

- **Tipos de dados:** o mapeamento prioriza tipos XSD. Tipos especializados, como os presentes na ontologia QUDT, não foram contemplados e são tratados como texto quando necessário, o que pode limitar certos cenários de visualização.

- **Rótulos e legibilidade:** por limitações fora do escopo do *driver*, a interface do Metabase exibe IRIs em lugar de rótulos (e.g., `rdfs:label`). Embora tecnicamente correta, essa apresentação pode aumentar a carga cognitiva para usuários não familiarizados com as IRIs do grafo em questão.

- **Segurança e autenticação:** conexões HTTPS/TLS com certificado inválido foram suportadas por necessidade de compatibilidade, mas devem ser evitadas em produção. Para autenticação, mecanismos de autenticação (básica ou por *token*) não foram implementados, permanecendo como trabalho futuro.

- **Métricas de desempenho:** *endpoints* públicos podem variar quanto à disponibilidade e à carga, dessa forma, a latência de rede não foi utilizada para medição neste trabalho. Assim, estudos futuros devem considerar a utilização de métricas de desempenho para avaliar o impacto da latência na experiência do usuário.

Os resultados apresentados e discutidos ao longo deste trabalho indicam que é viável integrar dados em RDF, consultados via SPARQL, a uma plataforma de BI de amplo uso como o Metabase. Em termos práticos, essa integração reduz a barreira de entrada para analistas que desejam explorar grafos de conhecimento com ferramentas familiares de visualização. Além disso, a entrega sob licença livre e com imagem *Docker* facilita a replicação por outros pesquisadores e equipes de dados, promovendo o reuso e a evolução da solução pela comunidade.

Por fim, o *driver* SPARQL para o Metabase atingiu os objetivos apontados no início deste trabalho ao permitir a conexão direta a *endpoints* SPARQL, a execução de consultas dos tipos `SELECT` e `ASK`, a conversão de tipos para o modelo tabular do Metabase e a construção de visualizações representativas a partir dos resultados. As limitações identificadas são esperadas e compatíveis com o que foi proposto no escopo e com os desafios que a natureza fluida dos grafos RDF e a heterogeneidade dos *endpoints* na Web Semântica trazem. Dessa forma, o trabalho contribui para aproximar dois ecossistemas até então distantes e oferece uma base concreta para estudos e implementações futuras que desejem explorar grafos RDF em contextos analíticos e exploratórios.

7 CONCLUSÕES

Este trabalho teve início a partir da lacuna identificada entre a crescente disponibilidade de dados em RDF, acessíveis via SPARQL, e a experiência prática identificada por analistas em plataformas de BI. Diante desse cenário, foi definida a estratégia de integrar esses dois mundos por meio do desenvolvimento de um *driver* SPARQL para o Metabase, solução que permite criar visualizações e *dashboards* diretamente a partir de grafos RDF, evitando etapas intermediárias complexas (capítulos 1 e 2).

Ao longo do estudo, foram definidos requisitos e critérios de aceitação que guiaram o desenvolvimento e a entrega do *driver* (capítulo 5). A proposta contemplou: (i) conexão a *endpoints* via HTTP/HTTPS, com opção de ignorar a validação de certificado TLS quando necessário; (ii) suporte a consultas `SELECT` e `ASK`; (iii) mapeamento de classes e propriedades RDF para o modelo tabular utilizado pela interface do Metabase; e (iv) integração dos resultados à interface visual, possibilitando a criação de gráficos e *dashboards*. De forma complementar, a proposta incluiu disponibilização pública sob licença livre, distribuição como artefato JAR e imagem Docker, além de um fluxo de integração contínua (CI) com testes automatizados.

Os resultados alcançados demonstram que a integração é tecnicamente viável, útil para fomentar análises de grafos RDF no Metabase e possui apelo prático para a criação de visualizações e *dashboards*, como evidenciado no capítulo 6:

- **Conectividade** (`C2`, `C3`): conexão com oito *endpoints*, incluindo um com TLS inválido (seção 6.2.1).
- **Metadados** (`C4`): representação de classes como tabelas e de predicados como colunas, com limites para manter a usabilidade e minimizar o impacto no desempenho do *endpoint* (com opção de desativar a sincronização de metadados; seção 6.2.2).
- **Consultas** (`C5`, `C6`): `ASK` e `SELECT` funcionando em múltiplos *endpoints*; suporte a resultados formatados em JSON de acordo com a especificação SPARQL 1.0/1.1 (seções 6.3.1 e 6.3.2).
- **Tipos** (`C7`): mapeamento de XSD para tipos nativos do Metabase (seção 6.3.3).
- **Parâmetros** (`C8`): sintaxe `{{...}}` para reuso de consultas na interface, bastando a alteração de parâmetros (seção 6.3.4).
- **Visualizações** (`C9`): diversos gráficos a partir de resultados tabulares (seção 6.4).
- **Entrega e CI** (`C1`, `C10`, `C11`, `C12`): JAR, Docker, licença aberta e CI com linters e testes unitários (seções 6.1.1 a 6.1.3 e 6.5).

Do ponto de vista científico e prático, a contribuição central é mostrar que a integração direta entre SPARQL e uma ferramenta de BI amplia o uso de análises sobre dados em RDF, mantendo o foco em ferramentas já consolidadas de *dashboards*, evitando que soluções de nicho sejam utilizadas. O trabalho também evidencia limites inerentes à natureza deste assunto: a heterogeneidade dos grafos; as diferenças de desempenho entre *endpoints*; as diferenças semânticas entre MBQL e SPARQL, que requerem decisões de mapeamento cuidadosas; e a representação de um modelo de grafo em um paradigma distinto, o tabular.

Apesar do cumprimento dos critérios de aceitação, o escopo do *driver* foi delimitado, pois o objeto deste trabalho possui a característica de estar sempre evoluindo, própria de projetos *open-source*. A conversão básica de MBQL para SPARQL implementada neste trabalho abre caminho para evoluções que ampliariam a experiência do usuário no *query builder*, necessitando de estudos mais aprofundados para funcionalidades avançadas. Entre as funcionalidades mais relevantes para trabalhos futuros relacionadas ao *query builder*, é possível destacar:

- **Agregações**: implementação de funções como `COUNT`, `SUM`, `AVG`, `MIN` e `MAX`, que permitiriam análises estatísticas diretamente na interface visual;
- **Agrupamentos**: funcionalidade essencial para análises por categoria, possibilitando criar gráficos mais elaborados através de agrupamentos;
- **Filtros avançados**: suporte a filtros de texto mais sofisticados, por exemplo, `REGEX` do SPARQL, ampliando as possibilidades de busca e filtragem de dados;
- **Junções entre classes RDF**: capacidade de relacionar diferentes classes em uma única consulta, permitindo análises mais complexas.

Essas evoluções ampliariam o alcance e a utilidade do *driver*, tornando-o mais intuitivo para usuários que não possuem conhecimentos profundos em SPARQL. Além disso, possibilitariam que analistas de BI pudessem criar consultas complexas e visualizações utilizando apenas a interface visual.

Outra sugestão de melhoria identificada durante os testes é em relação a prefixos comuns de bancos de dados RDF. Alguns *endpoints*, como o da Wikidata, acrescentam os prefixos comuns[1] automaticamente a consultas SPARQL (exemplo: `PREFIX wdt: <http://www.wikidata.org/prop/direct/>`). O desenvolvimento desta funcionalidade facilita o trabalho do analista que escreve a consulta SPARQL.

O suporte à conversão somente de tipos XSD também é citado como uma limitação atual e segue como proposta para trabalhos futuros. O suporte a outros tipos de dados

[1] https://www.wikidata.org/wiki/EntitySchema:E49

garante a correta interpretação dos resultados na interface do Metabase e permite que as visualizações sejam feitas corretamente.

Além disso, uma abordagem semelhante ao LinkDaViz (THELLMANN et al., 2015) pode ser desenvolvida e adotada. Uma ontologia especializada em fornecer metadados para a ferramenta de BI poderia ser utilizada para descrever o banco de dados e sua estrutura sem sobrecarregar o *endpoint* SPARQL. Essa ideia necessita de lapidações, porém se mostra viável em ambientes em que se tem o controle sobre o banco de dados RDF.

São esperados trabalhos e estudos utilizando o *driver* para criar visualizações e *dashboards*. Esse tipo de uso se mostra altamente recomendado, pois evita etapas complexas de ETL, como apontado no capítulo 2. A disponibilização de um conjunto de consultas de exemplo e de uma vitrine de visualizações tende a reduzir a curva de aprendizado e a estimular contribuições da comunidade.

Outra frente relevante é em relação à segurança. O desenvolvimento de suporte à autenticação (básica, por *token* e cabeçalhos personalizados) permite ampliar a gama de *endpoints* que podem ser utilizados.

Por fim, mas não menos importante, o repositório público permite o *feedback* contínuo da comunidade, o que torna o código mais robusto e progressivamente mais abrangente, contemplando uma ampla gama de funcionalidades. A adoção em equipes de BI pode fornecer retornos sobre usabilidade, cobertura de tipos de dados e compatibilidade entre *endpoints*, retroalimentando o ciclo de evolução do projeto.

REFERÊNCIAS

ABADI, D. J. et al. Scalable semantic web data management using vertical partitioning. In: **Proceedings of the 33rd International Conference on Very Large Data Bases**. [S.l.]: VLDB Endowment, 2007. (VLDB '07), p. 411–422. ISBN 9781595936493.

ALEXAKI, S. et al. The ics-forth rdfsuite: managing voluminous rdf description bases. In: **Proceedings of the Second International Conference on Semantic Web - Volume 40**. Aachen, DEU: CEUR-WS.org, 2001. (SemWeb'01), p. 1–13.

ALPAR, P.; SCHULZ, M. Self-service business intelligence. **Business & Information Systems Engineering**, v. 58, p. 151–155, fev. 2016.

BERNERS-LEE, T. **Linked Data - Design Issues**. 2006. W3C Design Issues Note. Disponível em: https://www.w3.org/DesignIssues/LinkedData.html. Acesso em: 1 ago. 2025.

BERNERS-LEE, T.; HENDLER, J.; LASSILA, O. The semantic web: A new form of web content that is meaningful to computers will unleash a revolution of new possibilities. **ScientificAmerican.com**, mai. 2001.

BIRON, P. V.; MALHOTRA, A.; GROUP, W. X. S. W. **XML Schema Part 2: Datatypes - Second Edition**. [S.l.], 2004. Disponível em: https://www.w3.org/TR/2004/REC-xmlschema-2-20041028/.

BORGES, H.; VALENTE, M. T. What's in a github star? understanding repository starring practices in a social coding platform. **Journal of Systems and Software**, v. 146, p. 112–129, 2018. ISSN 0164-1212. Disponível em: https://www.sciencedirect.com/science/article/pii/S0164121218301961.

BUIL-ARANDA, C. et al. Sparql web-querying infrastructure: Ready for action? In: ALANI, H. et al. (Ed.). **The Semantic Web - ISWC 2013**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 277–293. ISBN 978-3-642-41338-4.

CHAUDHURI, S.; DAYAL, U.; NARASAYYA, V. An overview of business intelligence technology. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 54, n. 8, p. 88–98, ago. 2011. ISSN 0001-0782. Disponível em: https://doi.org/10.1145/1978542.1978562.

CLARK, K. G.; FEIGENBAUM, L.; TORRES, E. **Serializing SPARQL Query Results in JSON**. [S.l.], 2007. Disponível em: https://www.w3.org/TR/rdf-sparql-json-res/. Acesso em: 1 jul. 2025.

DUCHARME, B. **Learning SPARQL**. 2. ed. Sebastopol, CA: O'Reilly Media, 2013.

HAWKE, S. et al. **SPARQL Query Results XML Format (Second Edition)**. [S.l.], 2013. Disponível em: https://www.w3.org/TR/2013/REC-rdf-sparql-XMLres-20130321/.

HAWKE, S. et al. **SPARQL 1.1 Query Results JSON Format**. [S.l.], 2013. Disponível em: https://www.w3.org/TR/2013/REC-sparql11-results-json-20130321/.

HAWKE, S. et al. **SPARQL 1.1 Query Results CSV and TSV Formats**. [S.l.], 2013. Disponível em: https://www.w3.org/TR/2013/REC-sparql11-results-csv-tsv-20130321/.

KATAYAMA, T. D3sparql: Javascript library for visualization of sparql results. **CEUR Workshop Proceedings**, v. 1320, jan. 2014.

LABORIE, S. et al. Combining business intelligence with semantic web: Overview and challenges. In: **Informatique des Organisations et Systèmes d'Information et de Décision**. Biarritz, France: [s.n.], 2015.

LEHMANN, J. et al. Sparqles: Monitoring public sparql endpoints. **Semant. Web**, IOS Press, NLD, v. 8, n. 6, p. 1049–1065, jan. 2017. ISSN 1570-0844. Disponível em: https://doi.org/10.3233/SW-170254.

LEIDA, M. et al. Toward automatic generation of sparql result set visualizations: A use case in service monitoring. In: **Proceedings of the International Conference on e-Business**. [S.l.: s.n.], 2011. p. 1–6.

MENIN, A. et al. LDViz: a tool to assist the multidimensional exploration of SPARQL endpoints. In: **Web Information Systems and Technologies : 16th International Conference, WEBIST 2020, November 3-5, 2020, and 17th International Conference, WEBIST 2021, October 26-28, 2021, Virtual Events, Revised Selected Papers**. Springer, 2023, (LNBIP - Lecture Notes in Business Information Processing, LNBIP - 469). p. 149–173. Disponível em: https://hal.science/hal-03929913.

METABASE. **Metabase Documentation**. 2025. Disponível em: https://www.metabase.com/docs/latest/. Acesso em: 8 jul. 2025.

PRESSMAN, R. S.; MAXIM, B. R. **Engenharia de software - 9.ed**. [S.l.]: McGraw Hill Brasil, 2021.

RAJABI, E.; MIDHA, R.; SOUZA, J. F. de. Constructing a knowledge graph for open government data: the case of nova scotia disease datasets. **Journal of Biomedical Semantics**, Springer Science and Business Media LLC, v. 14, n. 1, abr. 2023. ISSN 2041-1480. Disponível em: http://dx.doi.org/10.1186/s13326-023-00284-w.

SKJÆVELAND, M. G. Sgvizler: A javascript wrapper for easy visualization of sparql result sets. In: SIMPERL, E. et al. (Ed.). **The Semantic Web: ESWC 2012 Satellite Events**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015. p. 361–365. ISBN 978-3-662-46641-4.

STEAR, B. J. et al. Petagraph: A large-scale unifying knowledge graph framework for integrating biomolecular and biomedical data. **Scientific Data**, Springer Science and Business Media LLC, v. 11, n. 1, dez. 2024. ISSN 2052-4463. Disponível em: http://dx.doi.org/10.1038/s41597-024-04070-w.

THELLMANN, K. et al. Linkdaviz - automatic binding of linked data to visualizations. In: ARENAS, M. et al. (Ed.). **The Semantic Web - ISWC 2015**. Cham: Springer International Publishing, 2015. p. 147–162. ISBN 978-3-319-25007-6.

USCHOLD, M. **Demystifying OWL for the enterprise**. San Rafael, CA: Morgan & Claypool, 2018. (Synthesis Lectures on Semantic Web: Theory and Technology).

W3C SPARQL Working Group. **SPARQL 1.1 Protocol**. [S.l.], 2013. Disponível em: https://www.w3.org/TR/2013/REC-sparql11-protocol-20130321/. Acesso em: 1 jul. 2025.

WALKER, J. et al. **How NXP performs event-driven RDF imports to Amazon Neptune using AWS Lambda and SPARQL UPDATE LOAD**. 2022. Disponível em: https://aws.amazon.com/pt/blogs/database/how-nxp-performs-event-driven-rdf-imports-to-amazon-neptune-using-aws-lambda-and-sparql-update-load/. Acesso em: 8 jul. 2025.

WILKINSON, K. et al. Efficient rdf storage and retrieval in jena2. In: **Proceedings of the First International Conference on Semantic Web and Databases**. Aachen, DEU: CEUR-WS.org, 2003. (SWDB'03), p. 120–139.

WOOD, D.; LANTHALER, M.; CYGANIAK, R. **RDF 1.1 Concepts and Abstract Syntax**. [S.l.], 2014. Disponível em: https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/. Acesso em: 1 jul. 2025.

YUAN, G. et al. A survey on mapping semi-structured data and graph data to relational data. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, v. 55, n. 10, fev. 2023. ISSN 0360-0300. Disponível em: https://doi.org/10.1145/3567444.

APÊNDICE A – CONSULTAS SPARQL UTILIZADAS PARA VALIDAR O CRITÉRIO DE ACEITAÇÃO *C5*

Código 23 – Consulta ASK número 1 utilizada para C5 - Verifica se alguma proteína foi adicionada à base na data de 09/01/2013 - Executada no *endpoint* UniProt

```
PREFIX up: <http://purl.uniprot.org/core/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

ASK
WHERE
{
  ?protein a up:Protein .
  ?protein up:created '2013-01-09'^^xsd:date
}
```

Fonte: https://sparql.uniprot.org/.well-known/sparql-examples#example13

Código 24 – Consulta ASK número 2 utilizada para C5 - Executada no *endpoint* AgroVoc

```
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX skosxl: <http://www.w3.org/2008/05/skos-xl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

ASK
WHERE
{
  ?subject skosxl:prefLabel/skosxl:literalForm ?label .
  FILTER (LCASE(STR(?label)) = "water")
}
```

Fonte: Elaborado pelo autor

Código 25 – Consulta ASK número 3 utilizada para C5 - Executada no *endpoint* AgroVoc

```
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX skosxl: <http://www.w3.org/2008/05/skos-xl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

ASK
WHERE
{
  ?subject skosxl:prefLabel/skosxl:literalForm ?label .
  FILTER (LCASE(STR(?label)) = "sky")
}
```

Fonte: Elaborado pelo autor

Código 26 – Consulta ASK número 4 utilizada para C5 - Executada no *endpoint* DBpedia

```
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

ASK
WHERE
{
  dbr:Barack_Obama dbo:birthDate "1961-08-04"^^xsd:date .
}
```

Fonte: Elaborado pelo autor

Código 27 – Consulta ASK número 5 utilizada para C5 - Executada no *endpoint* DBpedia

```
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX dbp: <http://dbpedia.org/property/>

ASK
WHERE
{
  dbr:Federal_University_of_Rio_de_Janeiro dbp:undergrad ?undergrad .
  FILTER (?undergrad > 10000)
}
```

Fonte: Elaborado pelo autor

APÊNDICE B – CONSULTAS SPARQL UTILIZADAS PARA VALIDAR O CRITÉRIO DE ACEITAÇÃO *C6*

Código 28 – Consulta SELECT número 1 utilizada para C6 - Agrega contagem de atribuições por fonte (organismo taxon:9606) - Executada no *endpoint* UniProt

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX taxon: <http://purl.uniprot.org/taxonomy/>
PREFIX up: <http://purl.uniprot.org/core/>
SELECT
  ?source
  (COUNT(?attribution) AS ?attributions)
WHERE
{
  ?protein a up:Protein ;
    up:organism taxon:9606 ;
    up:annotation ?annotation .
  ?linkToEvidence rdf:object ?annotation ;
    up:attribution ?attribution .
  ?attribution up:source ?source .
  ?source a up:Journal_Citation .
}
GROUP BY ?source
ORDER BY DESC(COUNT(?attribution))
```

Fonte: https://sparql.uniprot.org/.well-known/sparql-examples/?offset=15#example20

Código 29 – Consulta SELECT número 2 utilizada para C6 - Lista de mares nomeados por piratas - Executada no *endpoint* OSM - Sophox

```
SELECT *
WHERE
{
  ?sea osmt:place "sea" ;
    osmt:name:en ?english .
  FILTER CONTAINS(?english, "ar")
  BIND(REPLACE(?english, "ar", "arrr") AS ?pirate)
}
```

Fonte: https://wiki.openstreetmap.org/wiki/Sophox/Example_queries#Seas_named_by_pirates

Código 30 – Consulta SELECT número 3 utilizada para C6 - Países membros de organizações internacionais - Executada no *endpoint* CoyPu

```
PREFIX coy: <https://schema.coypu.org/global#>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?org ?orgLabel
  (GROUP_CONCAT(DISTINCT ?altLabel; SEPARATOR="; ") AS ?alt_labels)
  (COUNT(DISTINCT ?member) AS ?num_members)
  (GROUP_CONCAT(DISTINCT ?memberLabel; SEPARATOR="; ") AS ?members)
WHERE
{
  ?org a coy:InternationalOrganization ;
    rdfs:label ?orgLabel ;
    skos:altLabel ?altLabel .
  ?member coy:memberOf ?org ;
    rdfs:label ?memberLabel .
  FILTER (LANG(?memberLabel) = "en")
}
GROUP BY ?org ?orgLabel
ORDER BY ?orgLabel
LIMIT 200
```

Fonte: https://docs.coypu.org/SparqlSampleQueries.html#get-all-members-of-various-international-organizations

Código 31 – Consulta SELECT número 4 utilizada para C6 - Países mais populosos do mundo - Executada no *endpoint* Wikidata

```
SELECT DISTINCT ?countryLabel ?population
WHERE
{
  ?country wdt:P31 wd:Q6256 ;
    wdt:P1082 ?population .
  SERVICE wikibase:label { bd:serviceParam wikibase:language
     "[AUTO_LANGUAGE],mul,en" }
}
GROUP BY ?population ?countryLabel
ORDER BY DESC(?population)
```

Fonte: https://www.wikidata.org/wiki/Wikidata:SPARQL_query_service/queries/examples#Countries_sorted_by_population

Código 32 – Consulta SELECT número 5 utilizada para C6 - Conjuntos de dados mais populares no portal de dados abertos da Espanha - Executada no *endpoint* Datos.gob.es

```
SELECT DISTINCT ?label (COUNT(?x) AS ?num)
WHERE
{
  ?x a <http://www.w3.org/ns/dcat#Dataset> .
  ?x <http://purl.org/dc/terms/publisher> ?publicador .
  ?publicador <http://www.w3.org/2004/02/skos/core#prefLabel> ?label .
}
GROUP BY ?label
ORDER BY DESC(?num)
LIMIT 10
```

Fonte: https://datos.gob.es/en/accessible-sparql - Seção: “To obtain the names of the ten entities that have the most published data sets and view how many there are”

APÊNDICE C – CONSULTAS SPARQL UTILIZADAS PARA VALIDAR O CRITÉRIO DE ACEITAÇÃO *C9*

Código 33 – Consulta SELECT número 1 utilizada para C9 - eventos recentes (Wikidata)

```
SELECT ?event ?eventLabel ?date
WITH {
  SELECT DISTINCT ?event ?date
  WHERE {
    ?event wdt:P31/wdt:P279* wd:Q1190554 .
    OPTIONAL { ?event wdt:P585 ?date . }
    OPTIONAL { ?event wdt:P580 ?date . }
    FILTER (BOUND(?date) && DATATYPE(?date) = xsd:dateTime) .
    BIND (NOW() - ?date AS ?distance) .
    FILTER (0 <= ?distance && ?distance < 31) .
  }
  LIMIT 150
} AS %i
WHERE {
  INCLUDE %i
  SERVICE wikibase:label { bd:serviceParam wikibase:language
     "[AUTO_LANGUAGE],mul,en" . }
}
```

Fonte: https://www.wikidata.org/wiki/Wikidata:SPARQL_query_service/queries/examples#Recent_events

Código 34 – Consulta SELECT número 2 utilizada para C9 - museus em Barcelona (Wikidata)

```
SELECT DISTINCT ?item ?name ?coord ?lat ?lon
WHERE {
  hint:Query hint:optimizer "None" .
  ?item wdt:P131* wd:Q1492 .
  ?item wdt:P31/wdt:P279* wd:Q33506 .
  ?item wdt:P625 ?coord .
  ?item p:P625 ?coordinate .
  ?coordinate psv:P625 ?coordinate_node .
  ?coordinate_node wikibase:geoLatitude ?lat .
  ?coordinate_node wikibase:geoLongitude ?lon .
  SERVICE wikibase:label {
    bd:serviceParam wikibase:language "ca" .
    ?item rdfs:label ?name
  }
}
ORDER BY ASC(?name)
```

Fonte: https://www.wikidata.org/wiki/Wikidata:SPARQL_query_service/queries/examples#All_museums_in_Barcelona_with_coordinates

Código 35 – Consulta SELECT número 3 utilizada para C9 - número único (UniProt)

```
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX taxon: <http://purl.uniprot.org/taxonomy/>
PREFIX up: <http://purl.uniprot.org/core/>
PREFIX proteome: <http://purl.uniprot.org/proteomes/>
SELECT
  (COUNT(DISTINCT ?sequence) AS ?allIsoforms)
WHERE {
  ?protein up:reviewed true .
  ?protein up:organism taxon:9606 .
  ?protein up:sequence ?sequence .
  ?protein up:proteome/^skos:narrower proteome:UP000005640 .
}
```

Fonte: https://sparql.uniprot.org/.well-known/sparql-examples/?offset=50#example59

Código 36 – Consulta SELECT número 4 utilizada para C9 - rios mais longos por continente (Wikidata)

```
SELECT ?continent ?river ?continentLabel ?riverLabel ?maxlength
WHERE {
  {
    SELECT ?continent (MAX(?length) AS ?maxlength)
    WHERE {
      ?river wdt:P31/wdt:P279* wd:Q355304 ;
             wdt:P2043 ?length ;
             wdt:P30 ?continent .
    }
    GROUP BY ?continent
  }
  ?river wdt:P31/wdt:P279* wd:Q355304 ;
         wdt:P2043 ?maxlength ;
         wdt:P30 ?continent .
  SERVICE wikibase:label { bd:serviceParam wikibase:language
     "[AUTO_LANGUAGE],mul,en" . }
}
ORDER BY ?continentLabel
```

Fonte: https://www.wikidata.org/wiki/Wikidata:SPARQL_query_service/queries/examples#Longest_river_of_each_continent

Código 37 – Consulta SELECT número 5 utilizada para C9 - membros por organização (CoyPu)

```
PREFIX coy: <https://schema.coypu.org/global#>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?org ?orgLabel
       (GROUP_CONCAT(DISTINCT ?altLabel; SEPARATOR=", ") AS ?alt_labels)
       (COUNT(DISTINCT ?member) AS ?num_members)
       (GROUP_CONCAT(DISTINCT ?memberLabel; SEPARATOR=", ") AS ?members)
WHERE {
  ?org a coy:InternationalOrganization ;
       rdfs:label ?orgLabel ;
       skos:altLabel ?altLabel .
  ?member coy:memberOf ?org ;
          rdfs:label ?memberLabel .
  FILTER (LANG(?memberLabel) = "en")
}
GROUP BY ?org ?orgLabel
ORDER BY DESC(?num_members)
LIMIT 20
```

Fonte: https://docs.coypu.org/SparqlSampleQueries.html#get-all-members-of-various-international-organizations - Limitada a 20 entidades e ordenada de forma decrescente por número de membros

Código 38 – Consulta SELECT número 6 utilizada para C9 - série temporal de população (Wikidata)

```
SELECT ?year ?population
WHERE {
  wd:Q730 p:P1082 ?p .
  ?p pq:P585 ?year ;
     ps:P1082 ?population .
  SERVICE wikibase:label { bd:serviceParam wikibase:language "en" }
}
ORDER BY ?year
```

Fonte: https://www.wikidata.org/wiki/Wikidata:SPARQL_query_service/queries/examples#Population_growth_in_Suriname_from_1960_onward

Código 39 – Consulta SELECT número 7 utilizada para C9 - distribuição por publicador (datos.gob.es)

```
SELECT DISTINCT ?label (COUNT(?x) AS ?num)
WHERE {
  ?x a <http://www.w3.org/ns/dcat#Dataset> .
  ?x <http://purl.org/dc/terms/publisher> ?publicador .
  ?publicador <http://www.w3.org/2004/02/skos/core#prefLabel> ?label .
}
GROUP BY ?label
ORDER BY DESC(?num)
```

Fonte: https://datos.gob.es/en/accessible-sparql (seção: “To obtain the names of the ten entities that have the most published data sets and view how many there are”) - Removida a cláusula `LIMIT 10` da consulta original

Código 40 – Consulta SELECT número 8 utilizada para C9 - elementos químicos: massa (X) e ponto de ebulição (Y) (Wikidata)

```
SELECT ?elementLabel ?boiling_point ?melting_point
?electronegativity ?density ?mass
WHERE {
?element wdt:P31 wd:Q11344 ;
          wdt:P2102 ?boiling_point ;
          wdt:P2101 ?melting_point ;
          wdt:P1108 ?electronegativity ;
          wdt:P2054 ?density ;
          wdt:P2067 ?mass .
  SERVICE wikibase:label { bd:serviceParam wikibase:language
     "[AUTO_LANGUAGE],en" }
}
LIMIT 100
```

Fonte: https://www.wikidata.org/wiki/Wikidata:SPARQL_query_service/queries/examples/advanced#Dimensions_of_elements