Integração entre SPARQL e Business Intelligence: Um Driver para Visualização de Dados RDF no Metabase

José Vitor Hisse Cabral[1], **Giseli Rabello Lopes**[1], **João Pedro Pinheiro**[2]

[1] Instituto de Computação - Universidade Federal do Rio de Janeiro (UFRJ)

[2]Departamento de Informática - Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio)

{josehisse,giseli}@ic.ufrj.br, jpinheiro@inf.puc-rio.br

Abstract. *Interactive visualization of RDF data via SPARQL remains a challenge for Business Intelligence (BI) tools. This study presents the development of a Metabase driver, implemented in Clojure, that enables direct querying of SPARQL endpoints and data exploration within tabular models. Unlike existing web-based tools that require technical expertise, this approach integrates RDF data into a consolidated BI platform. The solution employs a class-partitioned property table strategy, performing automatic metadata discovery and XSD type conversion. Tests conducted on public endpoints, such as DBpedia and Wikidata, demonstrate the feasibility of creating bar, line, scatter, and map visualizations through both native queries and visual builders. Results indicate the technical feasibility of visualizing and exploring RDF data within a BI platform.*

Resumo. *A visualização interativa de dados RDF via SPARQL permanece um desafio para ferramentas de Business Intelligence (BI). Este trabalho apresenta o desenvolvimento de um driver para o Metabase, implementado em Clojure, que permite a consulta direta a endpoints SPARQL e a exploração desses dados em modelos tabulares. Diferentemente das ferramentas existentes, baseadas em bibliotecas web e que exigem conhecimento técnico, a proposta integra dados RDF a uma plataforma de BI consolidada. A solução utiliza a estratégia de tabelas de propriedades particionadas por classes, realizando a descoberta automática de metadados e conversão de tipos XSD. Testes realizados em endpoints públicos, como DBpedia e Wikidata, demonstram a viabilidade de criar visualizações de barras, linhas, dispersão e mapas, tanto via consultas nativas quanto de forma visual. Os resultados indicam a viabilidade técnica de visualizar e explorar dados RDF em uma plataforma de BI.*

1. Introdução

O universo de dados conectados (*Linked Data*) cresceu consideravelmente nos últimos anos. O número de conjuntos de dados publicados no LOD Cloud[1] cresceu de 95, em 2009, para 1358, em 2026. Nesse modelo, a informação é representada como triplas *sujeito-predicado-objeto* que formam grafos de conhecimento, consultáveis por meio da linguagem SPARQL, protocolo padronizado pelo W3C que opera sobre HTTP [W3C SPARQL Working Group 2013]. Esse ecossistema de dados conectados abrange diversos domínios, como governamentais [Rajabi et al. 2023], científicos [Stear et al. 2024] e empresariais [Walker et al. 2022].

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

Ao mesmo tempo, plataformas de *Business Intelligence* (BI) de autosserviço ganharam popularidade. Apache Superset, Metabase e Redash acumulam, respectivamente, cerca de 65 mil, 42 mil e 27 mil estrelas no GitHub, refletindo o interesse crescente da comunidade *open-source*. Permitem tanto a exploração dos dados quanto a criação de gráficos e visualizações interativas, sem a dependência de equipes de programação especializadas [Alpar and Schulz 2016]. Apesar disso, são focadas em fontes relacionais acessadas via JDBC, sem suporte a dados RDF.

Essa falta de suporte pelas ferramentas de BI para dados RDF mantém isolados o universo dos dados conectados e o das plataformas de BI, pois não é possível utilizá-las para consultas a *endpoints* SPARQL. A ausência de uma ponte que conecte esses dois mundos é constatada pela própria comunidade, como na discussão `8063` presente no repositório no GitHub do Metabase intitulada *"Database Support: Virtuoso or generic SPARQL"*[2] ou na discussão `4510` do repositório do Apache Superset, *"Use Superset with SPARQL endpoint"*[3].

Ferramentas existentes para visualização de dados SPARQL [Katayama 2014, Skjæveland 2015, Menin et al. 2023] exigem conhecimento técnico (programação web ou escrita de consultas SPARQL) e não oferecem recursos típicos de plataformas de BI. Essa lacuna motiva este trabalho: aproveitar uma plataforma de BI consolidada em vez de desenvolver uma nova ferramenta.

Entre as plataformas de BI *open-source*, o Metabase foi escolhido por sua arquitetura de *drivers* extensível, que permite integrar fontes não relacionais sem modificar o núcleo da plataforma, como evidenciam seus *drivers* para MongoDB e Neo4j (Seção 2.4). Este trabalho propõe uma arquitetura para integrar grafos RDF a plataformas de BI, baseada no mapeamento de classes e propriedades RDF para o modelo tabular da plataforma (tabelas de propriedades particionadas por classes) e na tradução das consultas da interface visual para SPARQL. Apresentamos sua implementação na forma de um *driver* SPARQL *open-source* para o Metabase, que conecta a plataforma diretamente a *endpoints* SPARQL via HTTP/HTTPS. Por não depender de recursos específicos do Metabase, a abordagem pode ser aplicada a outras ferramentas de BI. A contribuição inclui: (i) suporte a consultas dos tipos `SELECT` e `ASK` com resultados integrados à interface visual; (ii) mapeamento automático de classes e propriedades RDF para o modelo tabular da plataforma, representando classes como "tabelas" e propriedades como "colunas"; (iii) conversão de tipos de dados XSD (*XML Schema Definition*) para os tipos nativos do Metabase; (iv) parametrização de consultas na interface; e (v) validação com oito *endpoints* públicos, incluindo DBpedia e Wikidata, demonstrando a criação de oito tipos de visualizações distintas.

O restante deste artigo está organizado da seguinte forma. A Seção 2 apresenta a fundamentação teórica sobre RDF, SPARQL, mapeamento grafo-tabela e a arquitetura do Metabase. A Seção 3 discute os trabalhos relacionados. A Seção 4 detalha a arquitetura e a implementação do *driver*. A Seção 5 apresenta a avaliação e os resultados obtidos. A Seção 6 traz as conclusões e direções para trabalhos futuros.

[2]https://github.com/metabase/metabase/issues/8063
[3]https://github.com/apache/superset/issues/4510

2. Fundamentação Teórica

Os conceitos apresentados a seguir são necessários para compreender o *driver* proposto: o modelo de dados RDF, a linguagem de consulta SPARQL (com foco nos formatos de resposta em JSON e nos tipos de dados retornados) e a estratégia de mapeamento grafo $\rightarrow$ tabela que serve de base para representar classes e propriedades na interface visual da plataforma de BI.

2.1. 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 et al. 2014]. Uma tripla RDF é formada por sujeito, predicado e objeto, e pode ser representada como $t = (s, p, o)$. Os termos de uma tripla podem ser IRIs (*Internationalized Resource Identifiers*), nós em branco ou literais. O sujeito é uma IRI ou nó em branco, o predicado é obrigatoriamente uma IRI, e o objeto pode ser qualquer um dos três tipos.

Ontologias descrevem formalmente os conceitos e relações de um domínio [Berners-Lee et al. 2001]. A distinção entre esquema (classes, propriedades, hierarquias) e instâncias (entidades concretas) [Uschold 2018] é relevante para o mapeamento proposto neste trabalho.

2.2. Linguagem de Consulta SPARQL

O SPARQL pode ser definido como a linguagem de consulta e um protocolo da Web Semântica. A versão 1.1 foi padronizada pelo W3C em 2013 [W3C SPARQL Working Group 2013]. Uma consulta SPARQL se assemelha a uma consulta SQL, porém seu foco é obter resultados sobre um grafo RDF e não sobre um banco de dados relacional.

A linguagem define quatro tipos de consulta. Este trabalho utiliza `SELECT`, que retorna dados tabulares com suporte a agregações, filtros e ordenações, e `ASK`, que retorna um valor booleano. Os tipos `CONSTRUCT` e `DESCRIBE` retornam grafos RDF e não são abordados.

Embora o resultado de uma consulta `SELECT` seja logicamente tabular, ele é serializado para transporte em um dos formatos definidos pelo SPARQL. O foco deste trabalho é o formato JSON [Seaborne et al. 2013], dada sua ampla adoção no ecossistema web. No JSON de retorno de uma consulta `SELECT`, no caminho `head` $\rightarrow$ `vars`, estão os nomes das colunas da consulta SPARQL. No caminho `results` $\rightarrow$ `bindings`, estão representados os dados que satisfazem a consulta. A Tabela 1 mostra as possibilidades de representação de acordo com a especificação do SPARQL 1.1 [Seaborne et al. 2013].

Há uma diferença entre as versões do protocolo: na versão 1.0, valores com tipos específicos aparecem no JSON como objetos `typed-literal` [Clark et al. 2007], enquanto na versão 1.1 utiliza-se `literal` com o campo `datatype` associado. O *driver* proposto aceita ambas as variantes para garantir compatibilidade.

Uma das maneiras mais comuns de representar tipos de dados no RDF é a utilização do XML Schema Definition (XSD). Por definição, um literal é uma *string* sem tipo associado, mas frequentemente é necessário representar tipos como inteiro, ponto flutuante, data ou booleano. Para efeitos deste trabalho, são mapeados apenas os tipos do XSD para os formatos internos do Metabase.

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"}`

Tabela 1. Termos RDF representados em JSON de acordo com SPARQL 1.1

2.3. Mapeamento Grafo-Tabela

O mapeamento grafo-tabela é a ponte entre o modelo de dados RDF e o modelo tabular utilizado na interface da plataforma de BI. 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. Para tal, a literatura sobre armazenamento de dados RDF em bancos de dados relacionais é utilizada [Yuan et al. 2023].

Entre as abordagens existentes, a tabela de triplas e a tabela de propriedades estendida não expõem simultaneamente classes e propriedades de forma adequada para uma interface de BI. Proposta para o Apache Jena 2 [Wilkinson et al. 2003], a tabela de propriedades particionada por classes (*Property Class Table*) resolve esse problema: cada classe gera uma tabela cujas colunas são o sujeito e os predicados associados às suas instâncias, conforme mostra a Figura 1.

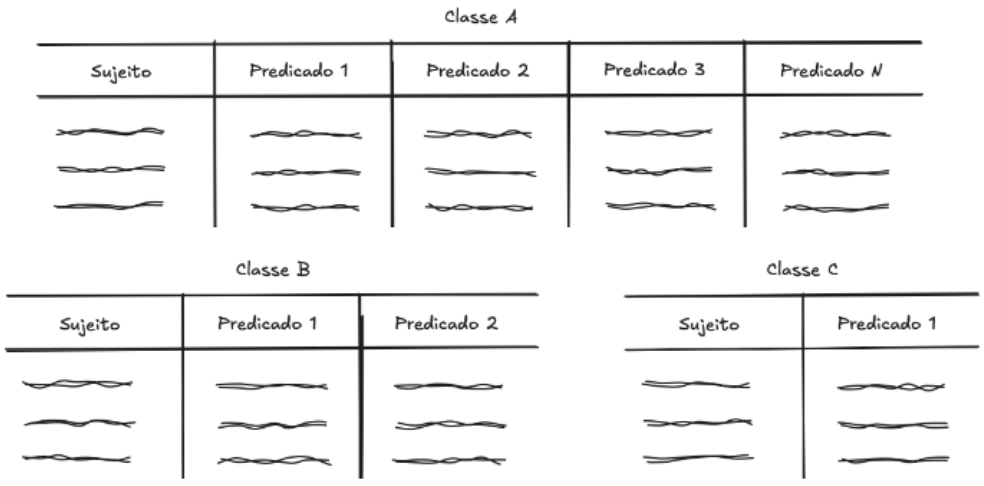

Figura 1. Tabela de propriedades particionada por classes (*Property Class Table*)

Esta é a abordagem 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 servem apenas de insumo para a representação visual dos elementos do grafo na ferramenta de BI.

2.4. Metabase e Arquitetura de Drivers

O Metabase é uma plataforma de BI *open-source* voltada para exploração e visualização de dados. A plataforma oferece um editor de consultas nativas, um construtor visual de consultas (*query builder*), 19 tipos de visualização e suporte a *dashboards* interativos [Metabase 2025].

A arquitetura do Metabase é extensível por meio de *drivers*, componentes que encapsulam a lógica de comunicação com fontes de dados. A plataforma é desenvolvida na linguagem Clojure, que oferece o mecanismo de multimétodos, uma forma de polimorfismo que seleciona a implementação de uma função em tempo de execução com base nos argumentos recebidos. Cada *driver* implementa um conjunto de multimétodos que definem como testar a conexão (`can-connect?`), descobrir metadados (`describe-database`, `describe-table`) e executar consultas (`execute-reducible-query`).

Embora a maioria dos *drivers* oficiais utilize SQL via JDBC, o Metabase mantém oficialmente um *driver* para o MongoDB, banco de dados NoSQL com linguagem de consulta própria (MQL). Na comunidade, existe um *driver* para o Neo4j, banco de dados baseado em grafos de propriedades que utiliza a linguagem Cypher. Esses precedentes demonstram a viabilidade técnica de integrar fontes não-relacionais à plataforma.

O *query builder* do Metabase traduz interações visuais do usuário em uma representação intermediária denominada *Metabase Query Language* (MBQL). Essa representação declarativa é então convertida pelo *driver* na linguagem nativa da fonte de dados. Na maioria dos *drivers*, essa conversão gera SQL; no caso do *driver* proposto, gera SPARQL. Os *drivers* são empacotados como artefatos JAR e instalados no diretório `plugins/` do Metabase, sem necessidade de modificar o código-fonte da plataforma.

3. Trabalhos Relacionados

A integração entre dados RDF e representações gráficas tem sido explorada por poucas ferramentas, a maioria baseada em uma abordagem web. O D3SPARQL [Katayama 2014] é uma biblioteca JavaScript que transforma resultados JSON de consultas SPARQL em gráficos SVG via D3.js, gerando gráficos de barras, pizza, dispersão, grafos e mapas. O Sgvizler [Skjæveland 2015] segue caminho semelhante, porém abstrai parte do código, onde o usuário define a consulta e o tipo de gráfico em atributos HTML, e a biblioteca envia uma requisição HTTP/HTTPS para o *endpoint* SPARQL e converte o JSON recebido em objetos DataTable do Google Charts. Ambas exigem manipulação de HTML e JavaScript, o que limita a adoção por analistas sem experiência em desenvolvimento web.

O LDViz [Menin et al. 2023] propõe um fluxo em cinco etapas (importação, transformação, mapeamento, renderização e interação) para explorar grafos RDF. Sua variedade de gráficos é limitada e não suporta *dashboards*. Nos testes apresentados pelos autores, apenas 42% dos 419 *endpoints* testados retornaram resultados válidos, deixando evidente os desafios de compatibilidade com servidores SPARQL públicos. Por fim, o LinkDaViz [Thellmann et al. 2015] mostra uma abordagem diferente ao propor uma ontologia de visualização para recomendar o tipo de gráfico; no entanto, verificamos que seu protótipo não está mais disponível, tornando impossível a reprodução dos resultados e uma avaliação mais detalhada.

A Tabela 2 resume o comparativo entre as ferramentas. O ponto em comum entre as soluções é a ausência de recursos típicos de plataformas de BI, como *dashboards*. As ferramentas que consultam *endpoints* diretamente exigem programação web ou escrita de consultas SPARQL; o LinkDaViz dispensa programação e escrita de SPARQL, mas ainda exige familiaridade com o modelo de dados RDF, depende da ingestão prévia dos dados e não está mais disponível. Isso fundamenta a escolha de uma plataforma de BI existente para o desenvolvimento do *driver* SPARQL e não a criação de uma nova ferramenta.

Ferramenta	**Tipo**	**Acesso aos dados**	**Conhecimento exigido**	**Dashboards**	**Mantido**
D3SPARQL	Biblioteca JS	Endpoint remoto	Programação web + SPARQL	Não	Não
Sgvizler	Biblioteca JS	Endpoint remoto	Programação web + SPARQL	Não	Sim
LDViz	Aplicação web	Endpoint remoto	Escrita de SPARQL	Não	Sim
LinkDaViz	Aplicação web	Ingestão de RDF/CSV	Modelo de dados RDF	Não	Não
Este trabalho	***Driver* para BI**	**Endpoint remoto**	**Nenhum (SPARQL opcional)**	**Sim**	**Sim**

Tabela 2. Comparativo entre as ferramentas

4. Arquitetura e Implementação

Desenvolvemos o *driver* em Clojure, linguagem nativa do Metabase, e o empacotamos como artefato JAR instalável no diretório `plugins/` da plataforma. A implementação consiste em um conjunto de multimétodos (descritos na Seção 2.4) que definem como testar a conexão, descobrir metadados, executar consultas e testar funcionalidades suportadas. Nem todos os multimétodos disponíveis precisam ser implementados, já que alguns não se aplicam ao contexto do SPARQL. Por exemplo, `truncate!` e `insert-into!` não são necessários, pois o *driver* não realiza operações de escrita na fonte de dados.

A Figura 2 apresenta o fluxo completo de uma consulta: do clique do usuário na interface, passando pelo *driver*, até o *endpoint* SPARQL e o retorno dos dados ao Metabase.

4.1. Conexão com *endpoints* SPARQL

A primeira decisão foi a escolha das configurações que o *driver* utiliza para se conectar a *endpoints* SPARQL. Os *endpoints* recebem consultas via protocolos HTTP e HTTPS. Portanto, dois campos básicos devem estar presentes na configuração: a URL do servidor e a opção de ignorar a validação do certificado TLS. Além dessas duas configurações, o protocolo SPARQL 1.1 permite que seja definido um grafo padrão para a consulta, ou seja, um terceiro campo do tipo *string*.

O primeiro multimétodo a ser executado quando uma nova conexão é criada é o `can-connect?`. Ele verifica se o *endpoint* está acessível. Devido à variedade de formas

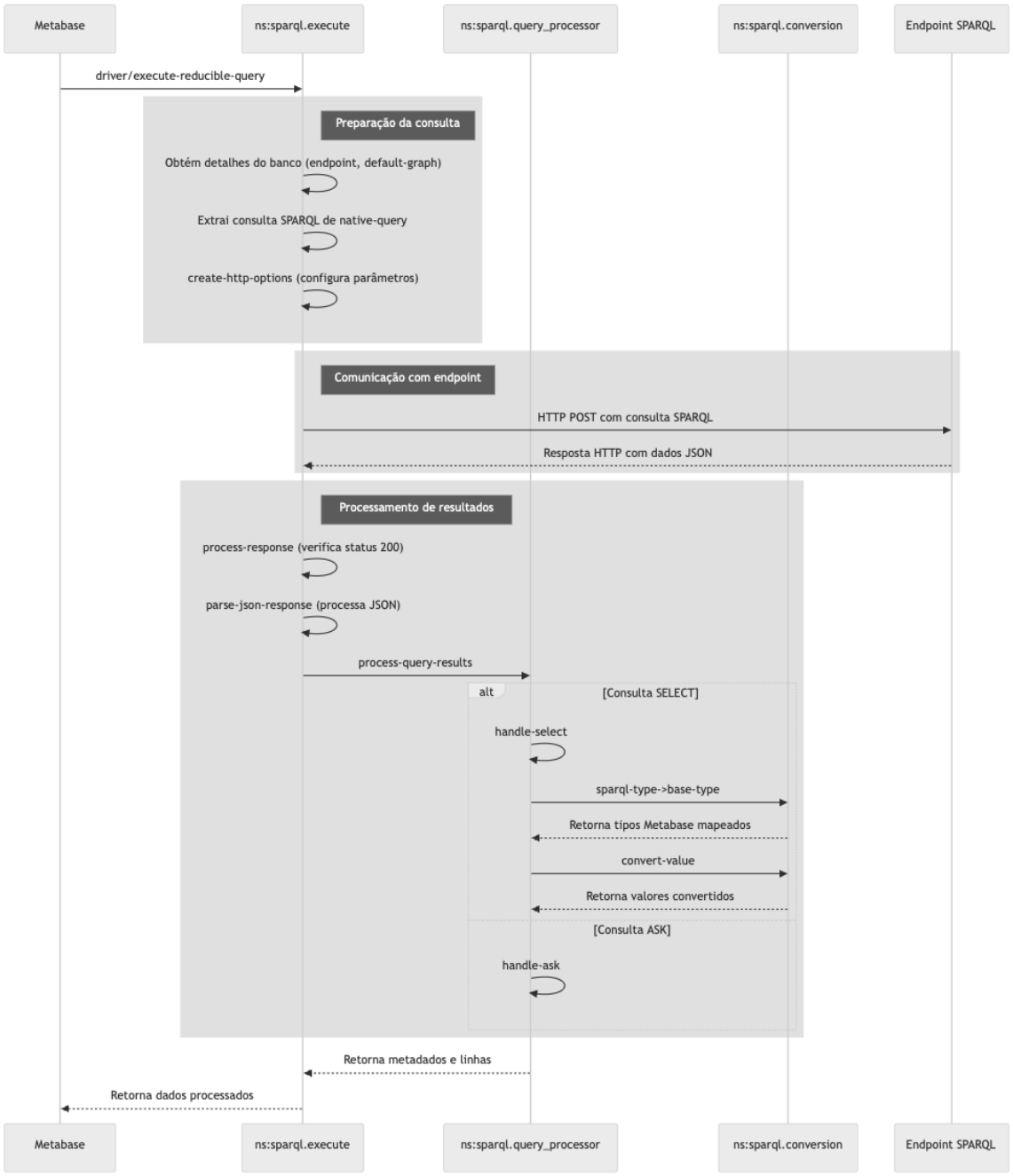

Figura 2. Diagrama de sequência da execução de uma consulta desde a origem no Metabase até o *endpoint* SPARQL

de testar uma conexão a um servidor SPARQL [Vandenbussche et al. 2017], avaliamos algumas estratégias ao longo do desenvolvimento.

Após avaliar alternativas como `ASK WHERE ?s ?p ?o` (falha em grafos vazios) e `SELECT` com `BIND` (incompatível com SPARQL 1.0), adotamos `ASK { }`, que sempre retorna verdadeiro, funciona nas versões 1.0, 1.1 e no rascunho da 1.2[4], e gera *overhead* mínimo de rede.

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

4.2. Descoberta de metadados

Conforme a Seção 2.3, classes são representadas como tabelas e predicados como colunas na interface do Metabase. Para representar classes como tabelas, é necessário considerar que os dados da Web Semântica são heterogêneos. Por exemplo, a DBpedia possui 1.568 classes e o UniProt 470.306[5]. Diante disso, adotamos a estratégia de priorizar as classes mais utilizadas, ordenando-as da mais frequente para a menos frequente. A consulta "`SELECT ?class (COUNT(?s) AS ?count) WHERE { ?s a ?class . } GROUP BY ?class ORDER BY DESC(?count)`" lista as classes com mais instâncias no grafo.

Essa limitação é apenas visual; todas as classes continuam disponíveis para consultas no grafo RDF. De forma semelhante, os predicados são representados como colunas. Adotamos a estratégia de priorizar os predicados mais usados por cada instância que pertence a uma classe específica. Os menos frequentes não são exibidos na interface do Metabase. 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 e sobrecarregaria o banco de dados interno do Metabase.

Na interface, tanto as classes quanto os predicados são exibidos por suas IRIs completas. Como a consulta de agregação pode ser lenta em grafos volumosos, implementamos uma opção nas configurações da conexão para desativar a descoberta de metadados, mantendo disponíveis apenas as consultas nativas.

4.3. Execução de consultas

O editor de consultas nativas do Metabase permite aos usuários escrever consultas diretamente na linguagem que a fonte de dados compreende. No contexto deste *driver*, os usuários podem escrever consultas SPARQL completas, aproveitando todos os recursos que a linguagem oferece.

A execução segue o protocolo SPARQL 1.0 e 1.1. O *driver* apenas encaminha a consulta para o servidor; quem deve suportar as funcionalidades de cada versão do protocolo é o servidor SPARQL. O *driver* fica responsável por processar o retorno. Para isso, a função `execute-reducible-query` pega a consulta do Metabase, prepara o corpo da requisição e envia via HTTP POST para o servidor SPARQL. Durante esse processo, o *driver* configura os cabeçalhos HTTP corretos para solicitar a resposta em JSON e trata possíveis erros de rede ou do servidor.

O *driver* aceita os dois formatos JSON do protocolo (Seção 2.2) e converte os dados da mesma forma para o Metabase, independentemente da versão do servidor.

Para converter os tipos SPARQL em tipos nativos do Metabase, criamos a função `sparql-type->base-type`. Por ora, a função aceita apenas os tipos definidos na especificação XSD. Demais tipos de dados são tratados como texto literal, e IRIs são tratadas como URLs.

Outra funcionalidade 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

[5]Consulta executada em 04 de agosto de 2025

interface. Isso permite reutilizar a mesma consulta com filtros diferentes. O *driver* faz isso através da função `substitute-native-parameters`, que encontra todos os marcadores de parâmetro na consulta e os substitui pelos valores escolhidos pelo usuário antes de enviar para o servidor.

4.4. Conversão MBQL para SPARQL

O Metabase permite construir consultas por meio de uma interface visual, o *query builder*. As interações do usuário geram uma representação declarativa em MBQL, que é então convertida pelo *driver* em uma consulta SPARQL. Na implementação, adotamos a tabela de propriedades particionada por classes: `: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.

A conversão segue o seguinte fluxo: o sujeito é representado por `?s`, pois a tabela de propriedades particionada por classes define a primeira coluna como sujeito. A cláusula **`WHERE`** inclui `?s a <Classe>` e, para cada predicado referenciado em `:fields`, `:order-by` ou `:filter`, insere um padrão **`OPTIONAL`** `{ ?s <predicado> ?var . }`. 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. Filtros MBQL são mapeados para **`FILTER`** com operadores lógicos (`:and`, `:or`, `:not`), comparações (`:=`, `:!=`, `:>`, `:>=`, `:<`, `:<=`) e existência (`:is-null` → `!`**`BOUND`**, `:not-null` → **`BOUND`**).

A versão avaliada (`v0.0.5`) 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.

5. Avaliação

Para garantir que os objetivos fossem atendidos, definimos requisitos funcionais e não funcionais, cada um acompanhado de um critério de aceitação. Todos os testes foram realizados em agosto de 2025 com o *driver* na versão `v0.0.5`, Metabase versão `0.55.5.1`, Clojure `1.12.1.1550` e OpenJDK `21`.

Disponibilizamos o código-fonte em repositório público no GitHub sob a licença AGPL v3.0. O *driver* é distribuído como artefato JAR e como imagem Docker. Em ambos os casos, o Metabase inicializou sem exceções e confirmou o carregamento do *driver* via log.

Para os testes de conectividade, utilizamos oito *endpoints* SPARQL públicos (Wikidata, DBpedia, UniProt, AgroVoc, Datos España, LinkedGeoData, CoyPu e OSM-Sophox), escolhidos para abranger diferentes domínios, implementações de servidor SPARQL e escalas de grafo. Sete usam HTTPS/TLS válido; o *endpoint* `datos.gob.es` tinha certificado TLS inválido na data dos testes, mas o *driver* conectou-se a ele habilitando a opção de ignorar a validação. O teste de conexão utiliza a consulta `ASK { }`, descrita na Seção 4.

Após cada conexão, o Metabase requisita ao *driver* a descoberta de metadados para obter tabelas e colunas, no nosso caso, classes e propriedades RDF. Durante a sincronização, o *driver* consulta as classes mais frequentes no grafo e, para cada classe, os predicados mais utilizados por suas instâncias. Classes são exibidas como "tabelas", vide Figura 3, e predicados como "colunas" na interface.

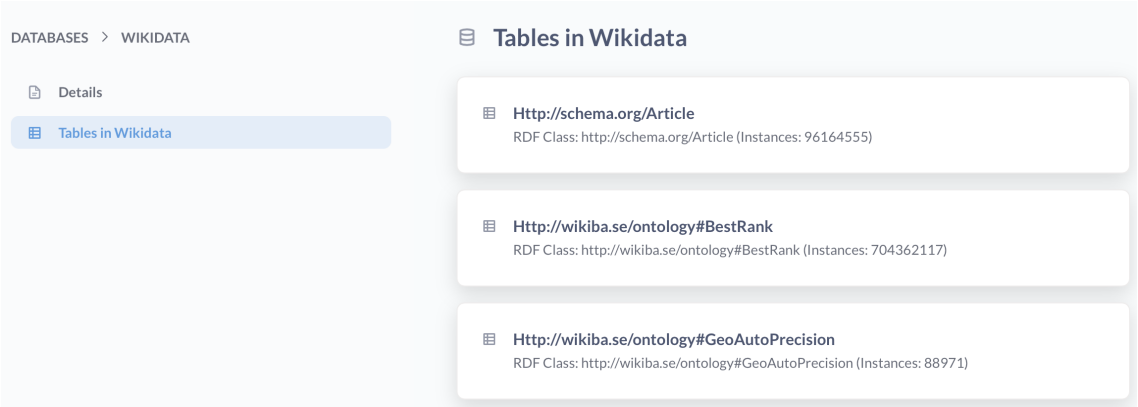

Figura 3. Mapeamento de classes como tabelas na interface do Metabase

Para as consultas, executamos cinco consultas `ASK` (em DBpedia, UniProt e AgroVoc) e cinco `SELECT` (em Wikidata, OSM-Sophox, CoyPu, UniProt e AgroVoc), escolhidas para testar os principais recursos da linguagem tratados pelo *driver*. As consultas nativas (não montadas no *query builder*) cobriram cenários de uso real, com `FILTER`, `ORDER BY`, `LIMIT`, agregações (`COUNT` e `GROUP BY`), `GROUP_CONCAT` e `REPLACE`, e todas foram processadas corretamente. A compatibilidade com as versões 1.0 e 1.1 do protocolo, descrita na Seção 2.2, foi confirmada nos testes.

Verificamos a conversão de tipos XSD para tipos nativos do Metabase por inspeção visual na interface (inteiros, decimais, booleanos, datas e IRIs exibidos corretamente), análise de *logs* de depuração e testes unitários da função de conversão; tipos não reconhecidos são tratados como texto e as IRIs como URLs. A parametrização com a sintaxe {{`nome`}} foi validada na interface e por testes unitários que verificam a consulta gerada após a interpolação. A partir dos resultados tabulares, construímos oito tipos de visualização sem adaptações adicionais, conforme a Tabela 3 e Figura 4.

Tipo	**Endpoint**	**Descrição**
Tabela	Wikidata	Eventos recentes com colunas de tipos distintos
Mapa de pontos	Wikidata	Distribuição geográfica de museus em Barcelona
Número único	UniProt	Total de isoformas de um proteoma
Barras horizontais	Wikidata	Rios mais longos por sub-região do planeta
Barras verticais	CoyPu	Total de membros por organização internacional
Linha	Wikidata	Evolução da população do Suriname ao longo de décadas
Rosca	Datos España	Distribuição de publicadores de conjuntos de dados
Bolhas	Wikidata	Relação entre massa e ponto de ebulição de elementos químicos

Tabela 3. Tipos de visualização e exemplos por *endpoint*

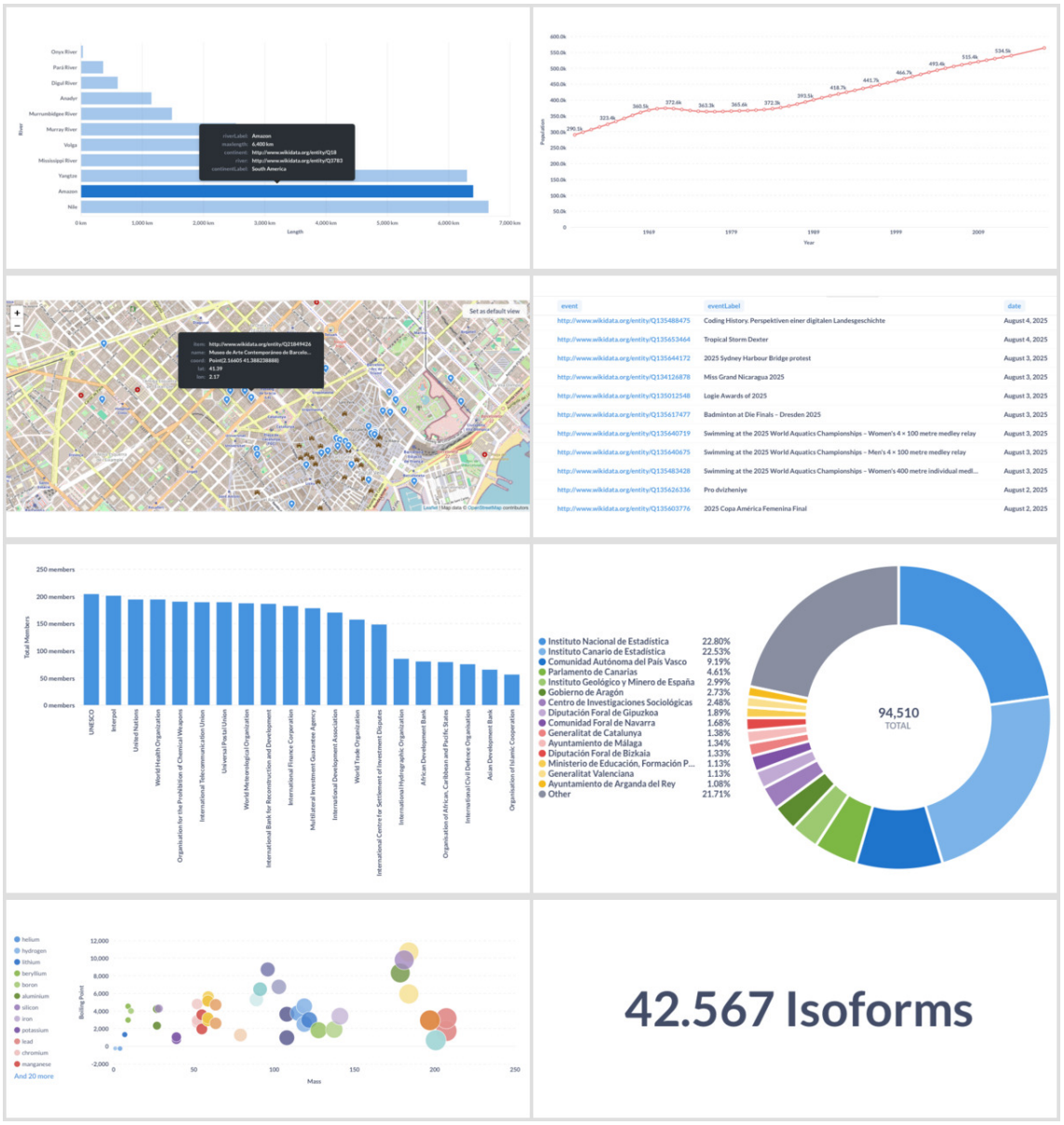

Figura 4. Visualizações geradas no Metabase a partir de consultas a *endpoints* SPARQL

Em todos os casos, o Metabase reconheceu automaticamente os tipos de dados XSD e os converteu para os tipos nativos, permitindo a exibição adequada dos eixos numéricos, escalas temporais e coordenadas geográficas.

Para garantir a qualidade do código, implementamos um fluxo de integração contínua no GitHub Actions com análise estática (clj-kondo e Splint), testes, compilação do JAR e, ao criar uma *tag*, a publicação automática da *release* e da imagem Docker. As etapas rodam em sequência, então falhas de análise ou de testes interrompem o fluxo e evitam a distribuição de artefatos defeituosos.

A conversão MBQL para SPARQL, descrita na Seção 4, foi verificada para seleção de instâncias com filtros, ordenação e limite. Consultas geradas pelo *query builder* produziram tabelas corretas, porém sem suporte a agregações ou junções.

Apesar dos resultados positivos, identificamos algumas limitações. A descoberta das classes e predicados mais frequentes depende de agregações custosas, que podem aumentar o tempo de sincronização em grafos volumosos, especialmente em *endpoints* com recursos limitados. O *query builder* exibe apenas as classes e propriedades mais frequentes, uma visão parcial do grafo; as consultas nativas continuam sem restrições. As diferenças de suporte a SPARQL entre servidores [Buil-Aranda et al. 2013] também podem afetá-lo. O mapeamento de tipos prioriza XSD, e tipos especializados como os da ontologia QUDT são tratados como texto. A interface exibe as IRIs em vez de rótulos `rdfs:label`, o que pode aumentar a carga cognitiva para usuários não familiarizados com o grafo. Por fim, não implementamos autenticação e as métricas de latência ficaram fora do escopo, dada a variabilidade dos *endpoints* públicos.

6. Conclusão e Trabalhos Futuros

Apresentamos a implementação de uma arquitetura para integrar grafos RDF a plataformas de BI, na forma de um *driver* SPARQL *open-source* para o Metabase [Cabral 2025]. O *driver* conecta a plataforma diretamente a *endpoints* SPARQL via HTTP/HTTPS, executa consultas `SELECT` e `ASK`, mapeia classes e propriedades RDF para o modelo tabular e representa visualmente os resultados na interface da ferramenta. Avaliamos a solução com oito *endpoints* públicos, entre eles DBpedia, Wikidata e UniProt, e desenvolvemos oito tipos de visualizações a partir dos dados RDF consultados.

Com o *driver*, a consulta a *endpoints* SPARQL é feita direto pelo Metabase, sem *scripts* intermediários de extração ou transformação de dados para o formato relacional. A tabela de propriedades particionada por classes como estratégia de mapeamento se mostrou adequada, pois expõe a estrutura do grafo de forma tabular para quem está acostumado a utilizar ferramentas de BI, sem exigir conhecimento prévio do *schema*.

Conforme detalhado na Seção 5, quatro limitações impactam a experiência do usuário: a visão parcial do grafo no *query builder*, restrita às classes e propriedades mais frequentes; a latência da descoberta de metadados em grafos volumosos; a conversão de tipos restrita ao XSD; e a exibição de IRIs brutas.

Para trabalhos futuros, o suporte a tipos de dados além do XSD e o uso de uma ontologia auxiliar de metadados para descrever o grafo sem sobrecarregar o servidor podem ser frentes de pesquisa. Uma avaliação com usuários reais permitiria medir o impacto na produtividade de analistas que trabalham com dados RDF, algo que este trabalho não cobriu e que fortaleceria a validação da proposta. O *driver* passou a receber contribuições da comunidade *open-source* após a publicação, como o suporte a autenticação (HTTP Basic e Bearer) e a agregações e junções entre classes na conversão `MBQL->SPARQL`, que possibilitam análises mais elaboradas por usuários não especialistas em SPARQL. O código-fonte, a imagem Docker e a documentação estão disponíveis publicamente em um repositório Git[6].

[6]`https://github.com/jhisse/metabase-sparql-driver/tree/v0.0.5`

Referências

Alpar, P. and Schulz, M. (2016). Self-service business intelligence. *Business & Information Systems Engineering*, 58:151–155.

Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The semantic web. *Scientific American*, 284(5):34–43.

Buil-Aranda, C., Hogan, A., Umbrich, J., and Vandenbussche, P.-Y. (2013). Sparql web-querying infrastructure: Ready for action? In Alani, H., Kagal, L., Fokoue, A., Groth, P., Biemann, C., Parreira, J. X., Aroyo, L., Noy, N., Welty, C., and Janowicz, K., editors, *The Semantic Web - ISWC 2013*, pages 277–293, Berlin, Heidelberg. Springer Berlin Heidelberg.

Cabral, J. V. d. C. H. A. (2025). Driver para visualização de dados RDF via SPARQL no Metabase. Monografia, Universidade Federal do Rio de Janeiro, Rio de Janeiro.

Clark, K. G., Feigenbaum, L., and Torres, E. (2007). Serializing sparql query results in json. W3c working group note, World Wide Web Consortium (W3C).

Katayama, T. (2014). D3sparql: Javascript library for visualization of sparql results. In *Semantic Web Applications and Tools for Life Sciences 2014 (SWAT4LS 2014)*, volume 1320 of *CEUR Workshop Proceedings*.

Menin, A., Maillot, P., Faron, C., Corby, O., Dal Sasso Freitas, C. M., Gandon, F., and Winckler, M. (2023). 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*, volume LNBIP - 469 of *LNBIP - Lecture Notes in Business Information Processing*, pages 149–173. Springer.

Metabase (2025). Metabase documentation. Available at: `https://www.metabase.com/docs/latest/`. Accessed on: Jul. 8, 2025.

Rajabi, E., Midha, R., and de Souza, J. F. (2023). Constructing a knowledge graph for open government data: the case of nova scotia disease datasets. *Journal of Biomedical Semantics*, 14(1):4.

Seaborne, A., Clark, K. G., Feigenbaum, L., and Torres, E. (2013). Sparql 1.1 query results json format. W3c recommendation, World Wide Web Consortium (W3C).

Skjæveland, M. G. (2015). Sgvizler: A javascript wrapper for easy visualization of sparql result sets. In Simperl, E., Norton, B., Mladenic, D., Della Valle, E., Fundulaki, I., Passant, A., and Troncy, R., editors, *The Semantic Web: ESWC 2012 Satellite Events*, pages 361–365, Berlin, Heidelberg. Springer Berlin Heidelberg.

Stear, B. J., Mohseni Ahooyi, T., Simmons, J. A., Kollar, C., Hartman, L., Beigel, K., Lahiri, A., Vasisht, S., Callahan, T. J., Nemarich, C. M., Silverstein, J. C., and Taylor, D. M. (2024). Petagraph: A large-scale unifying knowledge graph framework for integrating biomolecular and biomedical data. *Scientific Data*, 11(1).

Thellmann, K., Galkin, M., Orlandi, F., and Auer, S. (2015). Linkdaviz - automatic binding of linked data to visualizations. In Arenas, M., Corcho, O., Simperl, E., Strohmaier, M., d'Aquin, M., Srinivas, K., Groth, P., Dumontier, M., Heflin, J., Thiruna-

rayan, K., Thirunarayan, K., and Staab, S., editors, *The Semantic Web - ISWC 2015*, pages 147–162, Cham. Springer International Publishing.

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

Vandenbussche, P.-Y., Umbrich, J., Matteis, L., Hogan, A., and Buil-Aranda, C. (2017). Sparqles: Monitoring public sparql endpoints. *Semant. Web*, 8(6):1049–1065.

W3C SPARQL Working Group (2013). Sparql 1.1 protocol. W3c recommendation, World Wide Web Consortium (W3C).

Walker, J., Buijs, O., Ivie, C., and de Koning, J. (2022). How NXP performs event-driven RDF imports to Amazon Neptune using AWS Lambda and SPARQL UPDATE LOAD. Available at: `https://aws.amazon.com/pt/blogs/database/how-nxp-performs-event-driven-rdf-imports-to-amazon-neptune-using-aws-lambda-and-sparql-update-load/`. Accessed on: Jul. 8, 2025.

Wilkinson, K., Sayers, C., Kuno, H. A., and Reynolds, D. (2003). Efficient rdf storage and retrieval in jena2. In *Proceedings of the First International Conference on Semantic Web and Databases*, SWDB'03, pages 131–150, Aachen, DEU. CEUR-WS.org.

Wood, D., Lanthaler, M., and Cyganiak, R. (2014). Rdf 1.1 concepts and abstract syntax. W3c recommendation, World Wide Web Consortium (W3C).

Yuan, G., Lu, J., Yan, Z., and Wu, S. (2023). A survey on mapping semi-structured data and graph data to relational data. *ACM Comput. Surv.*, 55(10).