Integrando Bases Autônomas Externas no Processamento de Consultas em Bases RDF Distribuídas

Hugo P. B. Takiuchi[1], **Raqueline R. de M. Penteado**[2], **Carmem S. Hara**[1]

[1] Departamento de Informática – Universidade Federal do Paraná
Caixa Postal 19.081 – 81.531-990 – Curitiba, PR – Brasil

[2]Departamento de Informática – Universidade Estadual de Maringá
Avenida Colombo 5.790 – 87.020-900 – Maringá, PR – Brasil
{hpbtakiuchi,carmem}@inf.ufpr.br, raque@din.uem.br

Abstract. *In RDF, a query can involve both data stored in third party autonomous databases and accessed through SPARQL endpoints, as well as data stored on a proprietary base. Federated systems process this type of query by accessing the external and proprietary databases as black-boxes. A moderator is responsible for sending sub-queries to the databases and combine their results. On the other hand, a traditional RDF system does not support accesses to external bases, but uses its proprietary base as a white-box, which allows optimizations in its internal processing strategies. This article proposes an alternative to the federated architecture, called FeSHyD, which explores a distributed proprietary base, allowing its servers to communicate with third-party databases during the query processing. The proposal promotes the parallel processing of queries on the proprietary base, decentralizing the tasks of the moderator in a federated system. Initial experiments show that FeSHyD can reduce query response time by up to 45% when compared to federated systems.*

Resumo. *No modelo RDF, uma consulta pode envolver tanto dados armazenados em bases autônomas de terceiros e acessados através de endpoints SPARQL, bem como dados armazenados em uma base proprietária. Sistemas federados processam esse tipo de consulta acessando as bases externa e proprietária como caixas-pretas. Um moderador envia subconsultas para as bases, que retornam os resultados das requisições. Por outro lado, um sistema RDF tradicional não acessa bases externas, mas utiliza sua base proprietária como caixa-branca, o que permite otimizações nas estratégias de processamento interno do sistema. Este artigo propõe uma alternativa à arquitetura federada, chamada de FeSHyD, que explora uma base proprietária distribuída, permitindo a comunicação dos seus servidores com bases de terceiros durante o processamento de consultas. A proposta promove o processamento paralelo de consultas na base proprietária, descentralizando a tarefa do moderador dos sistemas federados. Experimentos iniciais mostram que o FeSHyD pode reduzir o tempo de resposta de consultas em até 45% quando comparado com sistemas federados.*

1. Introdução

A *Web* Semântica promove a publicação de dados que podem ser interpretados tanto por humanos quanto por máquinas. O consórcio *W3C* definiu como seu modelo padrão o RDF (*Resource Description Framework*) e como linguagem de consulta o SPARQL *(SPARQL Protocol and RDF Query Language)*. Para integrar dados de diferentes fontes, consultas SPARQL podem ser enriquecidas combinando diversas bases RDF em uma única

requisição. Consultas desse tipo podem auxiliar usuários de sistemas RDF em suas pesquisas cotidianas até decisões em ambientes empresariais.

Sistemas federados dão suporte a consultas que envolvem múltiplas bases de dados [Rakhmawati et al. 2013]. Dada uma consulta, o moderador do sistema elabora um plano de execução envolvendo as etapas de análise e de seleção das fontes de dados relevantes para a consulta. No planejamento, o moderador divide a consulta em subconsultas que são enviadas às bases de dados selecionadas. Sistemas federados assumem que as bases pertencem a organizações de terceiros, acessando as bases como caixas-pretas, ou seja, elas simplesmente retornam os resultados das subconsultas recebidas. Bases deste tipo são acessadas por meio de *endpoints* SPARQL. Na sequência, o moderador processa as respostas obtidas a fim de gerar o resultado final da consulta.

Em um cenário no qual uma consulta envolve dados armazenados tanto em uma base de propriedade do próprio usuário que submete a consulta como em bases de terceiros, existem duas alternativas que podem ser adotadas para o seu processamento: *(i)* utilizar um sistema de consultas federado que considera todas as bases como caixas-pretas; *(ii)* ter uma intervenção do usuário para obter resultados intermediários do sistema RDF proprietário para encaminhá-los aos *endpoints*. A primeira alternativa é preferível, comparada à segunda, porque não passa para o usuário a responsabilidade de fazer as ligações entre as bases proprietárias e de terceiros. No entanto, tratar uma base proprietária como uma caixa-preta pode deixar de explorar possíveis otimizações, uma vez que o proprietário pode ter acesso irrestrito aos seus dados e à estratégia de processamento interno de consultas. Ou seja, a base proprietária pode ser tratada como uma caixa-branca[1]. Esse artigo explora esta alternativa, considerando uma base proprietária RDF distribuída. É proposto o sistema FeSHyD *(**Fe**derated **S**earch on **Hy**brid **D**atabases)*, que viabiliza o processamento de consultas SPARQL sobre uma base de dados distribuída e híbrida. O termo híbrido refere-se à composição da base do sistema, com uma base proprietária distribuída, tratada como caixa-branca, e bases de terceiros, acessadas como caixas-pretas por meio de *endpoints* SPARQL. A abordagem de processamento proposta permite a comunicação direta dos servidores da base proprietária com *endpoints* SPARQL durante o processamento de consultas, explorando o processamento paralelo entre os servidores durante o execução de consultas. Vale destacar que a abordagem proposta oculta dos usuários do sistema os detalhes envolvidos na execução de consultas.

Um estudo experimental comparou a abordagem de processamento de consultas do FeSHyD com uma abordagem *baseline* que adota a base proprietária como caixa-preta, assim como os sistemas federados. Os resultados experimentais mostraram que o FeSHyD pode reduzir o tempo de processamento em até 45%. O restante do artigo está organizado da seguinte maneira. A Seção 2 apresenta os trabalhos relacionados. O sistema proposto é descrito na Seção 3. Os experimentos são descritos na Seção 4 e a conclusão na Seção 5.

2. Trabalhos Relacionados

Existem diversos sistemas propostos na literatura que executam consultas SPARQL que envolvem múltiplas bases de dados. Elas variam na forma de acesso da

[1]Os termos caixa-branca e caixa-preta são utilizados com significado similar às técnicas de geração de casos de teste de software: que exploram a estrutura interna do programa e que utilizam apenas sua entrada e saída, respectivamente.

base proprietária e/ou de terceiros, como caixa-preta ou caixa-branca. A Tabela 1 compara cinco sistemas com a proposta do FeSHyD, que é o único sistema que processa consultas SPARQL acessando uma base proprietária distribuída como caixa-branca e bases de terceiros como caixa-preta. Os sistemas *SPLENDID* [Görlitz and Staab 2011], *oLinDa* [da Cunha and Lóscio 2014], *Lusail* [Abdelaziz et al. 2017] e *FedX* [Schwarte et al. 2011] são sistemas federados que executam consultas em bases de terceiros acessadas como caixas-pretas, mas que não consideram a existência de bases proprietárias. Dentre os sistemas que consideram uma composição híbrida estão o *Ephedra* [Nikolov et al. 2017] e o *SIHJoin* [Ladwig and Tran 2011]. O *Ephedra* acessa bases proprietárias distribuídas e de terceiros como caixa-preta. Além disso, o *Ephedra* requer auxílio dos usuários para a seleção das bases envolvidas nas consultas, enquanto o FeSHyD automatiza esse processo usando metadados da base distribuída e híbrida. Por fim, o *SIHJoin* acessa uma base proprietária centralizada como caixa-branca, além do acesso como caixa-preta a bases de terceiros. Usando a ideia de caixa-branca, o *SIHJoin* explora a indexação de dados na base centralizada para a otimização de consultas. No mesmo sentido, o FeSHyD explora a estratégia de processamento interno de consultas de uma base proprietária distribuída.

	Base Proprietária		**Base de Terceiros**	
	Armazenamento	**Acesso**	**Armazenamento**	**Acesso**
SPLENDID	X	X	Distribuída	Caixa-Preta (SPARQL)
oLinDa	X	X	Distribuída	Caixa-Preta (SPARQL)
Lusail	X	X	Distribuída	Caixa-Preta (SPARQL)
FedX	X	X	Centralizada e Distribuída	Caixa-Preta (SPARQL)
Ephedra	Distribuída	Caixa-Preta (SPARQL)	Distribuída	Caixa-Preta (SPARQL)
SIHJoin	Centralizada	Caixa-Branca	Distribuída	Caixa-Preta (SPARQL)
FeSHyD	Distribuída	Caixa-Branca	Distribuída	Caixa-Preta (SPARQL)

Tabela 1. Sistemas que executam consultas SPARQL em bases RDF distribuídas

3. FeSHyD

Esta seção apresenta o sistema FeSHyD, que viabiliza o processamento de consultas SPARQL sobre uma base de dados RDF distribuída e híbrida. O objetivo da composição híbrida é explorar otimizações que não seriam possíveis em uma arquitetura federada tradicional, na qual todas as fontes de dados são tratadas como caixas-pretas.

A arquitetura do FeSHyD é mostrada na Figura 1. Seus principais componentes são o moderador, os servidores que compõem a base proprietária distribuída e as bases externas. Para o detalhamento de como uma consulta é processada pelo sistema, considere os fluxos numerados da figura. Uma requisição SPARQL é recebida pelo moderador (1) e enviada para o *Módulo de Planejamento*, responsável pela geração de um plano de consulta, utilizando as informações armazenadas na base de metadados (2). Esta base contém, dentre outras informações, o esquema das bases proprietária e externas. A geração do plano inicia com a análise sintática e semântica (módulo *Análise da consulta*), seguida pela detecção das fontes de dados envolvidas na consulta e geração do plano de consulta (módulo *Geração do plano de consulta*). O plano define um percurso no grafo RDF. Ele é enviado para o *Módulo de execução* da consulta (3), que o encaminha para todos os servidores que compõem a base proprietária distribuída (4). Todos os servidores executam o mesmo plano de consulta em paralelo, com os dados armazenados localmente (G_{D_i}). Durante a execução de consultas, quando o percurso no grafo envolve dados armazenados externamente, as trocas de mensagens são realizadas sem a intervenção do

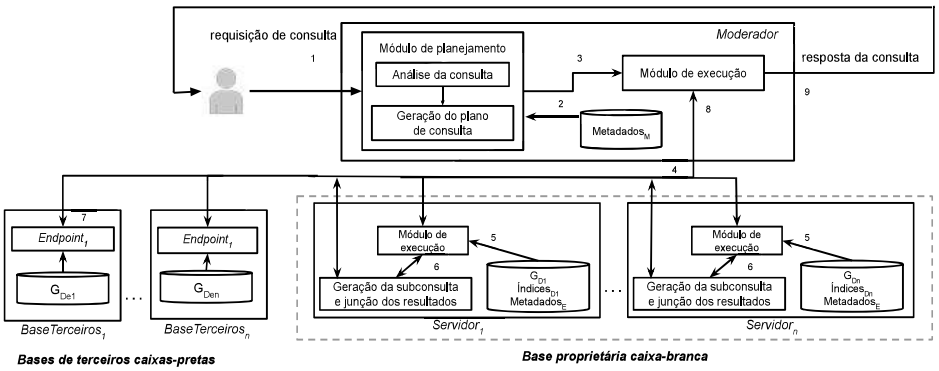


Figura 1. Arquitetura do sistema FeSHyD

moderador. Caso o dado necessário esteja em um servidor da base proprietária, o acesso direto é possível devido a existência de índices ($\acute{I}ndices_{D_i}$), que permitem que um servidor obtenha o endereço físico dos demais (5). Caso o dado necessário esteja localizado em uma base de terceiros, o módulo de *Geração da subconsulta e junção dos resultados*, gera a consulta SPARQL (6), que é enviada ao endpoint (7). Assim, que os resultados são recebidos, é feita então a junção. Ao final da execução do plano, cada servidor envia seus resultados para o moderador (8). O moderador é responsável por unir todos estes resultados e encaminhá-los ao usuário (9). Nas próximas seções, o modelo de dados e os principais módulos do FeSHyD são detalhados.

3.1. Modelo de Dados e Linguagem de Consulta

Uma base RDF é composta por um conjunto de triplas (sujeito predicado objeto), que pode ser representado na forma de um grafo, no qual vértices correspondem a sujeitos e objetos e arestas direcionadas correspondem a predicados, ligando o sujeito ao objeto. Neste artigo assume-se que os sujeitos estão armazenados com os seus objetos literais, bem como com a sua lista de adjacência, que inclui as arestas que partem e chegam ao vértice; ou seja, a base está fragmentada seguindo o padrão estrela. Um exemplo de grafo RDF de uma aplicação de comércio eletrônico, baseado no benchmark Berlin[2], é apresentado na Figura 2(a). Cada fragmento no formato estrela está delimitado com um fundo cinza. Observe que na base proprietária os dados estão distribuídos em dois servidores, W e X, e que há três bases de terceiros. Embora o FeSHyD só tenha acesso às bases de terceiros através de consultas SPARQL, assume-se que a parte do esquema que é de interesse da aplicação é conhecida. O grafo de estrutura da Figura 2(b) ilustra tanto o esquema da base proprietária, como das bases de terceiros. Assume-se que as ligações conhecidas são entre a base proprietária e uma base de terceiros, mas não entre duas bases de terceiros. O FeSHyD mantém em seus metadados o endereço (url_{ep}) de cada base de terceiros ep. É definida a função *end(v)* sobre os vértices do grafo de estrutura da seguinte forma: $end(v) = url$, caso v pertença a uma base de terceiros com endereço url; e $end(v) = \texttt{prop}$, caso v pertença à base proprietária, onde `prop` é uma constante.

O FeSHyD dá suporte à porção conjuntiva da linguagem de consultas SPARQL, na qual predicados são constantes, enquanto sujeitos e objetos são variáveis, que podem ser associadas a constantes na cláusula FILTER. Um exemplo de consulta é apresentada na Figura 3(a). Ela obtém os valores de ofertas com valores menores ou iguais a 58000

[2] `http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/`

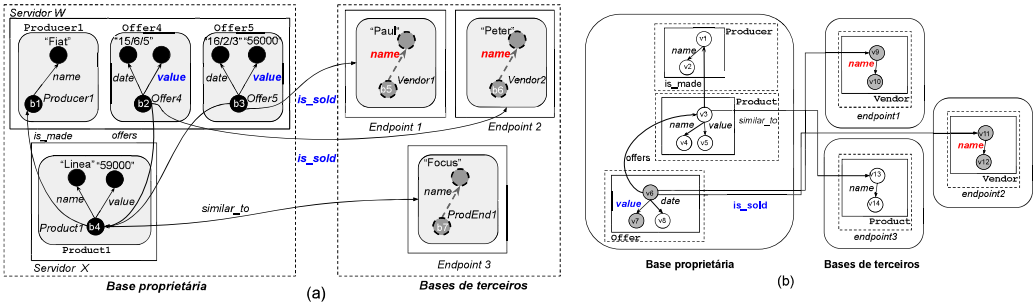


Figura 2. Base RDF distribuída (a) e seu respectivo grafo de estrutura (b)

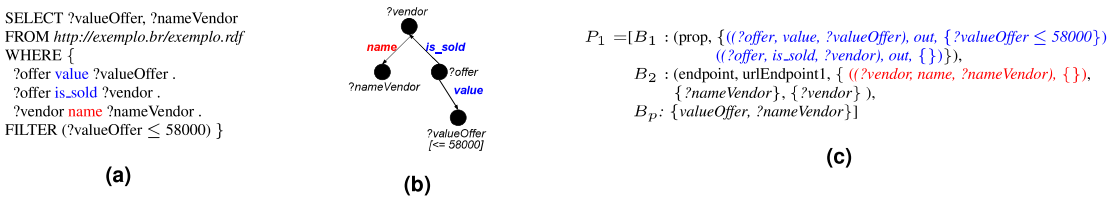


Figura 3. Consulta SPARQL (a); Grafo da consulta (b); Plano de execução (c)

com os nomes de seus respectivos vendedores. A representação de um grafo de consulta é similar à representação da base RDF, como ilustra a Figura 3(b), com filtros associados a nodos. Assim, uma consulta q é definida como um par $(G_q, result_q)$, onde G_q é um conjunto de pares $((s, p, o), F)$, nos quais (s, p, o) é uma aresta do grafo e F é um filtro aplicado sobre s ou o; e $result_q$ é o conjunto de variáveis retornados pela consulta. A consulta da Figura 3(a), é definida como: $q = (\{\ ((?offer, value, ?valueOffer), \{?valueOffer \leq 58000\}),\quad ((?offer, is\_sold, ?vendor), \{\}),\quad ((?vendor, name, ?nameVendor), \{\})\}, \{?valueOffer, ?nameVendor\})$. Neste artigo assume-se que o grafo (G_q) é conexo.

3.2. Planejamento de consultas

Dada uma consulta, o módulo de análise faz sua análise sintática e o resultado é a geração do grafo da consulta, como ilustrado na Figura 3(b). O grafo da consulta é utilizado na etapa seguinte, executada pelo módulo de geração do plano. Esta etapa inicia com a seleção de fontes, que tem como objetivo determinar quais são as bases de dados relevantes para o processamento da consulta. Como no FeSHyD a parte da estrutura das bases de terceiros que é de interesse da aplicação é conhecida, a seleção de fontes é baseada em um algoritmo para identificar os subgrafos homomórficos ao grafo da consulta no grafo de estrutura da base de dados. Por exemplo, considere o grafo de estrutura ilustrado na Figura 2(b). Nele, há dois subgrafos homomórficos ao grafo da consulta da Figura 3(b). Um dos subgrafos envolve o valor da oferta na base proprietária com o nome do vendedor no *Endpoint1* e o outro envolve o valor da oferta da base proprietária com o nome do vendedor no *Endpoint2*. Ou seja, para o processamento desta consulta, são selecionadas as fontes *Endpoint1* e *Endpoint2*, mas não *Endpoint3*.

O plano de consulta é gerado a partir dos subgrafos identificados no processo de seleção de fontes. Um exemplo de plano de consulta é ilustrado na Figura 3(c). Ele é composto por uma sequência de blocos, onde cada bloco representa o percurso dentro de um padrão estrela na base proprietária ou um acesso a um *endpoint*. Cada bloco tem um identificador, B_i, e o valor do primeiro elemento da sua tupla determina o tipo do bloco: *prop* para o processamento na base proprietária e *endpoint* para o processamento

em uma base de terceiros. Blocos do tipo *prop* definem um percurso no grafo; já blocos do tipo *endpoint* contém todas as informações necessárias para a geração de uma consulta SPARQL para ser submetida a uma bases externa. O plano da Figura 3(c) é composto por 2 blocos, um de cada tipo. O formato da tupla que define o bloco do tipo é *prop* tem a forma (prop, *percursoProp*), onde *percursoProp* é o conjunto de padrões de triplas a serem processados no bloco. Cada padrão tem o formato *(tripla, direcao, filtro)*, onde a *tripla* e *filtro* correspondem ao padrão de tripla e predicados definidos na consulta, e *direcao* pode ter o valor *in* ou *out*, indicando se o percurso é do objeto para o sujeito ou do sujeito para o objeto, respectivamente.

A tupla que define o bloco do tipo *endpoint* tem a forma (endpoint, *url, padraoTriplas, varRes, varJuncao*), onde: (1) url é o endereço do *endpoint*; (2) *padraoTriplas* contém os padrões de triplas a serem processados pela base externa, juntamente com os filtros aplicados sobre eles; (3) *varRes* são as variáveis no resultado da consulta obtidas da base externa; e (4) *varJuncao* são as variáveis que dão sequência ao percurso da base proprietária para a base externa. O plano de consulta é finalizado com um bloco de projeção B_p, que contém as variáveis no resultado da consulta (*result*$_q$).

O plano de execução da consulta define uma ordem para o percurso sobre a grafo RDF. Para gerar esta ordem, o FeSHyD utiliza uma heurística simples: o percurso sobre a base proprietária deve ser o maior possível antes que as bases de terceiros sejam acessadas. A intuição é que a latência nas conexões externas pode afetar o desempenho da consulta. Além disso, o percurso sobre a base proprietária pode filtrar os valores de interesse a serem obtidos da base externa, minimizando o volume de dados a ser transferido. A escolha do ponto inicial de exploração do grafo na base proprietária também pode influenciar no desempenho da consulta. Porém, este não é o foco deste artigo.

A geração do plano de execução da consulta é realizada pelo Algoritmo 1. A entrada do Algoritmo 1 é o grafo de estrutura (G_s) e uma consulta $q = (G_q, result_q)$. O resultado do algoritmo é um conjunto de planos de consulta $planSet$. O algoritmo inicia com a chamada à função *findSubgraphsInGs* (Linha 2), que retorna um conjunto de mapeamentos map que definem homomorfismos de G_q para subgrafos em G_S. Em seguida, para cada subgrafo, o algoritmo gera um plano de consulta (Linhas 3-34), que é inserido em *planSet* (Linha 35). Como exemplo, considere o grafo de estrutura da Figura 2(b) e o grafo de consulta na Figura 3(b). A função *findSubgraphsInGs* retorna dois mapeamentos: $map_1 = \{?offer \mapsto v_6, ?valueOffer \mapsto v_7, ?vendor \mapsto v_9, ?nameVendor \mapsto v_{10}\}$ e $map_2 = \{?offer \mapsto v_6, ?valueOffer \mapsto v_7, ?vendor \mapsto v_{11}, ?nameVendor \mapsto v_{12}\}$.

A geração de um plano a partir de cada mapeamento segue da seguinte forma. As triplas da consulta são divididas entre triplas na base proprietária G_{prop} (Linha 4) e de terceiros G_{ep} (Linha 9). Para isso é utilizada a função end, definida sobre vértices em G_S. Assim, considerando o mapeamento map_1, $end(map_1(?offer)) = end(v_6) =$ `prop` e $end(map_1(?vendor)) = end(v_9) = url_{Endpoint1}$. Portanto, para este mapeamento, $G_{prop} = \{(?offer, value, ?valueOffer), \{?valueOffer \leq 58000\}), (?offer, is\_sold, ?vendor), \{\})\}$ e $G_{ep} = \{((?vendor, name, ?nameVendor), \{\})\}$. Para facilitar a discussão, todos os exemplos a partir deste ponto são baseados em map_1.

Caso a parte da consulta sobre a base proprietária G_{prop} esteja vazia, a consulta deve ser completamente executada sobre uma base de terceiros. Assim, o plano de con-

Algoritmo 1: Geração do plano para uma consulta q

```
   Entrada: G_S, q = (G_q, result_q)
   Saída: planSet: conjunto de planos de consulta
1  planSet := {};
2  M := findSubgraphsInGs(G_S, G_q);
3  for each map in M do
4      G_prop := {((?s, p, ?o), F) ∈ G_q | end(map(?s)) = prop or end(map(?o)) = prop} ;
5      if G_prop = {} then
6          seja ((?s, p, ?o), F) um elemento em G_q;
7          plan := epBlock(1, end(map(?s)), G_q, result_q, {});
8      else
9          G_ep := G_q − G_prop;
10         ?s_1 := nodo em G_prop de maior grau;
11         plan := [ ]; bNum := 1; varList := [?s_1]; varInd := 1;;
12         for each endpoint ep_i do varToEp[i] := {};
13         while G_prop ≠ {} or G_ep ≠ {} do
14             if varInd ≤ |varList| then
15                 suj := varList[ind];
16                 block := createPropBlock(suj, map, G_prop, varList, varToEp);
17                 if block ≠ [] then
18                     plan := plan + propBlock(bNum, block);
19                     bNum := bNum + 1 ;
20                 varInd := varInd + 1;
21             else
22                 for each endpoint ep_i do
23                     while varToEp[i] ≠ {} do
24                         ini := uma variável em varToEp[i];
25                         block := createEpBlock(ini, map, G_ep, usedVars, outVars);
26                         if block ≠ [] then
27                             vJoinIn := varList ∩ usedVars;
28                             vJoinOut := outVars − varList;
29                             vRes := (usedVars ∩ result_q) ∪ vJoinOut;
30                             plan := plan+
31                             epBlock(bNum, url_ep_i, block, vRes, vJoinIn);
32                             insere vJoinOut em varList;
33                             bNum := bNum + 1;
34                         varToEp[i] := varToEp[i] − usedVars;
35     insert plan in planSet ;
36 return planSet;
```

sulta conterá um único bloco contendo todas as triplas da consulta, que é gerado pela função `epBlock`. Esta função recebe como entrada todos os componentes do bloco do tipo *endpoint*: o número do bloco, a url do endpoint, os padrões de triplas (G_q), as variáveis do resultado e as variáveis de junção (Linhas 5-7). Caso G_{prop} não seja vazio, o plano da consulta conterá blocos do tipo *prop* (gerados nas Linhas 14-20) e possivelmente blocos do tipo *endpoint* (Linhas 22-34). O plano sempre inicia com um percurso sobre a base proprietária. É escolhido o ponto inicial de exploração do grafo ($?s_1$). No algoritmo, é escolhido o nodo em G_{prop} de maior grau (Linha 10), mas outras estratégias podem ser adotadas. Embora a estratégia possa afetar o desempenho da consulta, ela não interfere na geração do plano. Os vértices em G_{prop} já visitados são guardados na variável $varList$, que inicia com $?s_1$. O índice $varInd$ indica a variável em $varList$ para a qual um novo bloco do plano de consulta será gerado. Ou seja, todas as variáveis em $varList$ anteriores à $varInd$ já foram consideradas na geração do plano. No algoritmo, todas as variáveis em $varList$ são processadas antes de gerar blocos do tipo *endpoint* (Linha 14). Isso garante que o plano gerado percorre o maior grafo conexo em G_{prop} que contém $?s_1$. Para cada variável, o bloco é gerado pela função *createPropBlock*. Além da geração do bloco, a função remove de G_{prop} as triplas processadas, insere em $varList$ as variáveis visitadas em G_{prop}, e insere em $varToEp[i]$ as variáveis que fazem ligações com a base

externa $endpoint_i$. Retornando ao exemplo, a variável *?offer* é escolhida como ponto inicial de exploração. Após a execução da função *createPropBlock*, o bloco B_1 da Figura 3(c) é gerado, G_{prop} estará vazio, $varList = [?offer, ?valueOffer]$, $varInd = 2$ e $varToEp[1] = \{?vendor\}$. A função será chamada novamente para a variável *?valueOffer*, mas como *block* retorna vazio, o plano não é alterado. Como todos as variáveis em $varList$ foram processadas, o algoritmo passa a processar variáveis localizadas em G_{ep}.

Para cada endpoint ep_i, os pontos de entrada na base estão armazenadas na variável *varToEp[i]*. Como um bloco do tipo *endpoint* não define um percurso na base, mas apenas identifica quais os padrões de triplas que devem ser processadas externamente, o algoritmo inicia obtendo aleatoriamente uma variável em *varToEp[i]*, chamada de ini (Linha 24). A partir desta variável a função *createEpBlock* gera um bloco para ser inserido no plano. Ao contrário da função *createPropBlock*, que limita-se a visitar os vizinhos de um nodo, esta função obtém o maior grafo conexo em G_{ep} que contém ini. Assim, o percurso neste grafo na base externa será requisitado em uma única consulta SPARQL durante o processamento da consulta. Além de gerar o bloco, a função *createEpBlock* remove de G_{ep} os padrões de triplas processados e retorna dois conjuntos de variáveis: as variáveis visitadas em ep_i (*usedVars*) e as variáveis na base proprietária ligadas à ep_i (*outVars*). A variável *usedVars* é utilizada para determinar as variáveis de junção do resultado da consulta SPARQL com os resultados dos blocos anteriores na consulta (Linha 27), bem como identificar quais destas variáveis compõem o resultado da consulta q (Linha 29). Além disso, é possível que nem todos os valores necessários de ep_i formem um único grafo conexo. Dessa forma, após gerar um bloco a partir de uma variável ini, as variáveis em *usedVars* são removidas de $varToEp[i]$ (Linha 34). Se ainda restarem variáveis, novos blocos podem ser gerados a partir delas (Linha 25).

Já a variável *outVars* é importante quando o grafo em G_{prop} é desconexo. Ou seja, embora é assumido que o grafo da consulta G_q seja conexo, a ligação entre eles percorre arestas em G_{ep}. Assim, *outVars* contém as variáveis que fazem a "re-entrada" na base proprietária a partir de uma base de externa. Portanto, estas variáveis devem ser retornadas no resultado da consulta (Linhas 28-29) e são inseridas na variável $varList$ para que o algoritmo prossiga com a geração de novos blocos do tipo *prop* (Linha 32). O plano P_1 da Figura 3(c) é gerado a partir do grafo homomórfico representado por map_1. A partir de map_2 é gerado um plano similar, chamado de P_2, com acesso ao *Endpoint2*.

3.3. Processamento da consulta

O conjunto de planos gerados são enviados para o módulo de execução do moderador. Ele analisa cada plano e para cada um, determina se ele envolve apenas bases de terceiros. Neste caso, as requisições são feitas diretamente pelo moderador. Isso ocorre quando o primeiro bloco da consulta é do tipo *endpoint*. Caso contrário, a consulta envolve dados da base proprietária. Assim, o moderador envia o plano de consulta para todos os servidores da base proprietária, que iniciam sua execução em paralelo com os dados armazenados localmente. O diferencial do FeSHyD é que toda a comunicação entre os servidores da base proprietária, bem como com as bases de terceiros é realizada sem a interferência do moderador. Isso evita que o moderador seja um gargalo no processamento das consultas.

É importante observar que a base de dados proprietária é fragmentada seguindo o padrão estrela e que os blocos do plano de consulta são gerados seguindo esta mesma estrutura. Esta coincidência entre as estruturas permite que apenas no início do proces-

samento de cada bloco seja necessário verificar se há necessidade de comunicação entre servidores. Isso é possível porque o modelo de fragmentação garante que todos os vértices do fragmento estão alocados em um mesmo servidor.

A comunicação entre os servidores da base proprietária é viabilizada pelo armazenamento de índices e metadados em cada servidor. Assim, no início de cada bloco do tipo *prop*, o módulo de execução verifica se ele acessa dados em outro servidor. Em caso afirmativo, os resultados intermediários são enviados para este servidor, que continua o processamento do plano. Cada resultado intermediário da consulta é representado por um mapeamento das variáveis para vértices da base de dados. Para exemplificar, considere o plano de consulta P_1 da Figura 3(c) e a base de dados da Figura 2(a). No servidor W, a execução do bloco B_1, gera como resultado os mapeamentos $r_1 = \{?offer? \mapsto b_2, ?valueOffer \mapsto 57000, ?vendor \mapsto b_6\}$ e $r_2 = \{?offer? \mapsto b_3, ?valueOffer \mapsto 56000, ?vendor \mapsto b_5\}$. Os dois mapeamentos são mantidos no resultado porque ambos satisfazem o filtro $?valueOffer \leq 58000$. Neste exemplo são utilizados identificadores para representar vértices em bases de terceiros (b_5 e b_6). Eles representam a url do recurso armazenado na base externa. A execução do plano P_2 no servidor W gera o mesmo resultado ($\{r_1, r_2\}$). No servidor X, nenhum resultado é gerado, uma vez que não existe nenhum vértice que contenha os predicados *value* e *is_sold*. Assim, o processamento da consulta prossegue apenas no servidor W.

Como o bloco seguinte é do tipo *endpoint*, o módulo de execução envia os resultados intermediários e o bloco para o módulo de geração de subconsulta e junção dos resultados. A consulta SPARQL é gerada da seguinte forma. Dado o bloco $B =$ *(endpoint, url, padraoTriplas, varRes, varJuncao)*, a cláusula select contém todas as variáveis em *varRes*; a cláusula where contém as triplas em *padraoTriplas*; e e a cláusula filter contém os filtros em *padraoTriplas*, além de uma disjunção de cláusulas conjuntivas que envolvem variáveis em *varJuncao*. Mais detalhadamente, cada resultado intermediário com variáveis em *varJuncao* associados a recursos com a url do endpoint, gera uma conjunção na cláusula filter. Dando sequência à consulta utilizada como exemplo, *varJuncao* contém uma única variável $?vendor$. O resultado intermediário gerado pela bloco B_1 é composto pelos mapeamentos $\{r_1, r_2\}$. Na execução do plano P_1, r_2 é descartado porque P_1 tem $url_{Endpoint1}$ e $r_2(?vendor) = b_6$ e b_6 não tem $url_{Endpoint1}$ como seu prefixo. Assim, P_1 gera, a partir de r_1, a expressão $(?vendor = b_5)$. Similarmente, P_2 descarta o resultado r_1 e gera a expressão $(?vendor = b_6)$. Observe que se os dois recursos pertencessem ao *Endpoint1*, o plano P_1 geraria a expressão $((?vendor = b_5)or(?vendor = b_6))$. O processo de geração dos filtros envolve também a eliminação de cláusulas conjuntivas repetidas. Como resultado final, o plano P_1 gera a consulta "SELECT *?vendor*, *?nameVendor* WHERE *?vendor name ?nameVendor* FILTER $(?vendor = b_5)$)", que é submetida ao *Endpoint1*. O plano P_2 gera uma consulta similar, que é submetida ao *Endpoint2*.

Os resultados da consulta são recebidos pelo servidor que submeteu a consulta, que realiza então uma equijunção com os resultados intermediários anteriores sobre os atributos em *varJuncao*. Assim, os resultados intermediários são estendidos com todas as variáveis presentes em *res*. Na consulta do exemplo, $res = \{?nameVendor\}$. Assim, o resultado da equijunção para P_1 é a extensão de r_1 com o mapeamento $(?nameVendor \mapsto "Paul")$ e em P_2, o resultado é a extensão de r_2 com $(?nameVendor \mapsto "Peter")$. O plano de consulta sempre termina com um bloco de projeção. Assim, apenas as variáveis

presentes neste bloco são mantidos nos resultados da consulta, que são encaminhados para o moderador. O moderador apenas faz a união dos resultados recebidos por todos os servidores para gerar o resultado final. O diferencial do FeSHyD é que, ao contrário da arquitetura tradicional de sistemas federados, nos quais a junção entre dados de bases distintas é realizada de maneira centralizada no moderador, ele trata a base proprietária como uma caixa-branca. Isso permite que o processamento desta operação, que em geral é custosa, seja realizada em paralelo nos servidores que compõem a base proprietária. O estudo experimental apresentado na próxima seção analisa o efeito desta estratégia no tempo de execução das consultas.

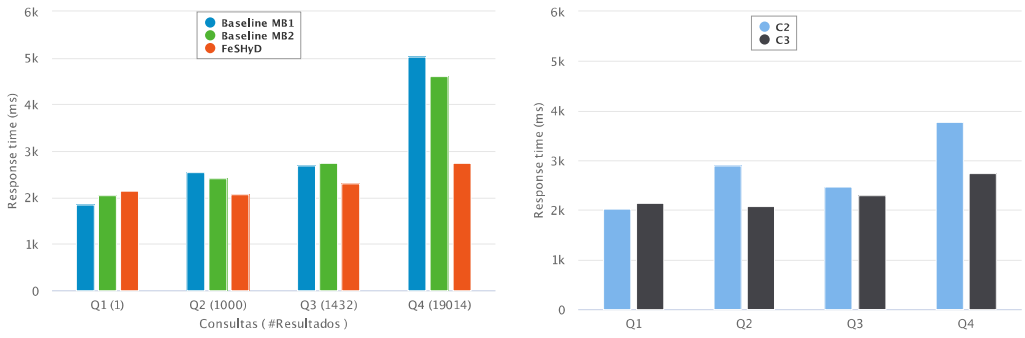
4. Estudo experimental

Foi conduzido um estudo experimental que analisou o *desempenho* da abordagem de processamento de consultas do FeSHyD, além do *efeito da variação do número de servidores da base proprietária* no sistema. O FeSHyD foi implementado como uma extensão do sistema *PAbS* [Penteado et al. 2019], que é uma base RDF distribuída. O *PAbS* armazena os dados em memória usando o repositório *Berkeley DB*. Na arquitetura do FeSHyD, o *PAbS* desempenha o papel da base proprietária. As extensões foram implementadas em C e a biblioteca *LibCurl* foi usada para a requisição de consultas SPARQL aos *endpoints*.

Para efeitos de comparação, foi implementado um modelo de processamento alternativo, no qual todos os acessos às bases de terceiros são executados no moderador. Ou seja, a partir do mesmo plano de consulta gerado pelo FeSHyD, sempre que há a necessidade de acessar uma base externa, os resultados intermediários são enviados ao moderador, que é responsável por preparar a consulta SPARQL, submetê-la à base externa, fazer a junção e dar continuidade à execução do plano. Esta estratégia é similar à adotada pelo sistema *Ephedra*, que é composta por uma base proprietária distribuída acessada como caixa-preta. Foram implementadas duas variantes desta estratégia: MB1, na qual *cada* recebimento dos resultados intermediários de cada servidor dispara requisições aos endpoints; e MB2, que aguarda o recebimento de todos os resultados intermediários dos servidores da base proprietária para fazer uma única requisição a cada endpoint. Em MB1, as requisições são feitas em paralelo no moderador usando o recurso de *threads*. Assim, a principal diferença entre as duas implementações é o número de requisições executadas durante o processamento das consultas.

Configuração dos experimentos. A base de dados usada nos experimentos contou com uma base proprietária gerada pelo *benchmark* Berlin e com uma base de dados de terceiros. O gerador de dados do Berlin usa o número de produtos como fator de escala. A base foi gerada usando o fator 1000 (107,17 MB). As instâncias de cada entidade foram distribuídas de maneira balanceada, ou seja, com cada servidor armazenando uma quantidade similar de dados. Foram consideradas duas configurações de clusters: $C2$, com dois servidores, e $C3$, com três servidores. $C2$ utilizou duas máquinas *i5* com 4 GB de RAM e $C3$ adicionou uma máquina *i5* com 8 GB de RAM em $C2$. Para o moderador foi utilizada uma máquina *i3* com 4 GB de RAM.

Quatro consultas SPARQL ($Q1$–$Q4$), baseadas nos casos de uso do Berlin, foram usadas nos experimentos. $Q1$ recupera dados de um determinado produto e do seu produtor; $Q2$ recupera dados de todos os produtos e seus produtores; $Q3$ recupera revisões com avaliação menor que 3 e seus revisores; e $Q4$ recupera ofertas com valor maior que 500,00 e seus vendedores. Em todas as consultas é gerada uma única subconsulta SPARQL que



(a) (b)

Figura 4. Tempo de resposta em $C3$ (a) e Efeito da quantidade de servidores (b)

acessa o *Wikidata* e que retorna um único resultado com 264 bytes. $Q1$, $Q2$, $Q3$ e $Q4$ retornam 1, 1000, 1432 e 19014 resultados, respectivamente. Os resultados reportados nos experimentos consideram o tempo mediano de 5 execuções de cada consulta.

Desempenho. O tempo de execução das consultas na configuração $C3$ é apresentado na Figura 4(a). É possível notar que o FeSHyD apresenta melhor desempenho, comparado a `MB1` e `MB2`, exceto para a consulta Q_1. O motivo para tal diferença é que a consulta Q_1 retorna um único resultado. Portanto, ela não explora o paralelismo na execução da consulta nos servidores da base proprietária. Assim, a diferença no tempo execução nos sistemas deve-se principalmente à latência na conexão com o endpoint, que é uma variável sobre a qual os sistemas não tem controle. De fato, para o FeSHyD, do total de 2138ms, 1877ms referem-se ao tempo de comunicação com o endpoint e junção dos resultados; para MB1 este tempo foi de 1893ms do total de 2052ms e para MB2 foi 1623ms do total de 1847. Assim, desconsiderando a diferença nos tempos de conexão, o FeSHyD tem uma pequena desvantagem com relação aos demais sistemas. Isso se deve ao fato da junção do resultado com o único resultado da consulta ser realizado no servidor. Assim, ao contrário de MB1 e MB2, que enviam para o moderador apenas o resultado intermediário, no FeSHyD a transmissão contém também o resultado da consulta retornado pelo endpoint.

O desempenho do FeSHyD melhora com o aumento do número de resultados. O ganho para as consultas Q_2-Q_4 foi de 18%, 14% e 45% com relação a MB1 e de 14%, 16% e 40% com relação a MB2. Como os dados estão distribuídos de forma balanceada entre os 3 servidores, o ganho se deve ao paralelismo nos servidores na execução das tarefas de: (1) geração da consulta e armazenamento dos resultados intermediários; (2) acesso ao endpoint; e (3) junção dos resultados. A quantidade de resultados intermediários gerados para as consultas Q_2-Q_4 no FeSHyD em cada servidor foi de aproximadamente 323, 476 e 6373. No entanto, em MB1 e MB2, a quantidade total de resultados intermediários recebidos foi de 1000, 1432 e 19014, respectivamente. Ou seja, as 3 tarefas que são realizadas em paralelo pelos servidores são realizadas pelo moderador para um volume muito maior, impactando no tempo total de processamento. Pode-se observar também que quanto maior o número de resultados, maior o volume de dados transmitidos dos servidores para o moderador no FeSHyD, uma vez que cada resultado intermediário é estendido com os valores retornados pelo endpoint. No entanto, este custo adicional não é maior que o ganho gerado pelo paralelismo no processamento dos dados armazenados em cada servidor. Esta diferença reflete no crescimento do tempo de execução no FeSHyD com relação a quantidade de resultados. No FeSHyD, ele cresce em um ritmo menor que em MB1 e MB2, como pode ser observado na Figura 4(a). Com relação ao número de requisições para as consultas Q_2-Q_4, MB1 realizou 3 requisições ao endpoint,

enquanto MB2 realizou uma única requisição. Em Q_4 nota-se que apesar da espera para o recebimento de 19014 resultados intermediários em MB2, a execução de 3 requisições ao endpoint penalizou MB1.

Efeito da variação do número de servidores da base proprietária. O número de servidores da base proprietária influencia diretamente no paralelismo explorado pelo FeSHyD durante a execução de consultas. A Figura 4(b) mostra o tempo de execução das consultas $Q1$–$Q4$ nos *clusters* $C2$ e $C3$ pelo FeSHyD. Como esperado, $Q1$ não obteve vantagem com o aumento de servidores, uma vez que retorna um único resultado. Mas o tempo de resposta das demais consultas diminuiu de $C2$ para $C3$. Esta diminuição foi de 28%, 6% e 27% para Q_2-Q_4, respectivamente. Q_3 apresentou uma redução menor devido à latência da conexão com o endpoint, que foi em média 200ms maior que para as demais consultas. Este experimento mostra que, para uma distribuição balanceada dos dados, um número maior de servidores favorece o paralelismo e reduz o tempo de resposta das consultas, principalmente quando há um grande número de respostas.

5. Conclusão

Esse artigo apresentou o sistema FeSHyD, que dá suporte ao processamento de consultas SPARQL em bases RDF distribuídas e híbridas. Ele explora o acesso irrestrito ao modelo de processamento da base proprietária para permitir a comunicação direta dos servidores que a compõem com bases externas. Dessa forma, a carga de processamento comumente centralizada no moderador de sistemas federados passa a ser descentralizada e distribuída. O estudo experimental realizado mostrou que o FeSHyD melhora o desempenho de consultas, principalmente quando elas envolvem uma grande quantidade de resultados. Como trabalho futuro, planeja-se estender o sistema para outros operadores da linguagem SPARQL, integrá-lo a um modelo mais elaborado para escolha dos pontos iniciais de exploração do grafo e investigar a utilização de consultas do tipo ASK para a seleção de fontes quando o grafo de estrutura da base externa não é conhecido.

Referências

Abdelaziz, I., Mansour, E., Ouzzani, M., Aboulnaga, A., and Kalnis, P. (2017). Lusail: a system for querying linked data at scale. *Proc. of the VLDB Endowment*, 11(4).

da Cunha, D. R. B. and Lóscio, B. F. (2014). oLinDa: uma abordagem para decomposição de consultas em federações de dados interligados. In *Anais do XXIX do SBDD*.

Görlitz, O. and Staab, S. (2011). Splendid: SPARQL endpoint federation exploiting VoID descriptions. In *Proc. of the 2nd Int. Conference on Consuming Linked Data*.

Ladwig, G. and Tran, T. (2011). SIHJoin: Querying remote and local linked data. In *The Semantic Web: Research and Applications*. Springer Berlin Heidelberg.

Nikolov, A., Haase, P., Trame, J., and Kozlov, A. (2017). Ephedra: Efficiently combining RDF data and services using SPARQL federation. In *Int. Conference on Knowledge Engineering and the Semantic Web*. Springer.

Penteado, R. R. M., Takiuchi, H. P. B., and Hara, C. S. (2019). PAbS: Um processador de consultas SPARQL sobre bases distribuídas. In *Anais do XXXIV SBDD, Demos track*.

Rakhmawati, N. A., Umbrich, J., Karnstedt, M., Hasnain, A., and Hausenblas, M. (2013). Querying over federated SPARQL endpoints-a state of the art survey. *ArXiv:1306.1723*.

Schwarte, A., Haase, P., Hose, K., Schenkel, R., and Schmidt, M. (2011). Fedx: Optimization techniques for federated query processing on linked data. In *Int. Semantic Web Conference*.