RAQUELINE RITTER DE MOURA PENTEADO

OTIMIZAÇÃO DE CONSULTAS SPARQL EM BASES RDF DISTRIBUÍDAS

Tese apresentada ao Programa de Pós-Graduação em Informática do Setor de Ciências Exatas da Universidade Federal do Paraná, como requisito parcial à obtenção do título de Doutor em Ciência da Computação no Programa de Pós-Graduação em Informática do Setor de Ciências Exatas da Universidade Federal do Paraná.
Orientadora: Prof[a]. Dr[a]. Carmem Satie Hara.

CURITIBA

2017

P419o Penteado, Raqueline Ritter de Moura
Otimização ao de consultas SPARQL em bases RDF distribuídas / Raqueline Ritter de Moura Penteado. – Curitiba, 2017.
85 f. : il. color. ; 30 cm.

Tese - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática, 2017.

Orientador: Carmem Satie Hara.
Bibliografia: p. 83-85.

1. Bases de dados. 2. RDF. 3. SPARQL. I. Universidade Federal do Paraná. II. Hara, Carmem Satie III. Título.

CDD: 005.133

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DO PARANÁ
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
Setor CIÊNCIAS EXATAS
Programa de Pós-Graduação INFORMÁTICA

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da tese de Doutorado de **RAQUELINE RITTER DE MOURA PENTEADO** intitulada: **Otimização de Consultas SPARQL em Bases RDF Distribuídas**, após terem inquirido a aluna e realizado a avaliação do trabalho, são de parecer pela sua aprovação.

Curitiba, 07 de Abril de 2017.

Carmem Satie Hara
CARMEM SATIE HARA
Presidente da Banca Examinadora (UFPR)

REBECA SCHROEDER FREITAS
Avaliador Externo (UDESC)

Cristina Dutra de Aguiar Ciferri
CRISTINA DUTRA DE AGUIAR CIFERRI
Avaliador Externo (USP)

LUIS CARLOS ERPEN DE BONA
Avaliador Interno (UFPR)

THIAGO HENRIQUE SILVA
Avaliador Externo (UTFPR)

FABIO ANDRE MACHADO PORTO
Avaliador Externo (LNCC)

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DO PARANÁ
Setor de Ciências Exatas
Coordenação de Pós-Graduação em Informática

Rua Cel. Francisco H. dos Santos, 100 - Centro Politécnico da UFPR - Curitiba - Paraná - Brasil
CEP 81531-980 - Tel: (41) 3361-3101 - E-mail: ppgint@inf.ufpr.br

À Luiz Cesar, Miguel eTiago

AGRADECIMENTOS

Agradeço a Deus por me guiar, me iluminar e me dar sabedoria para seguir em frente com os meus objetivos mesmo diante de adversidades. Deus transformou medo em coragem, dor em força e sonho em realidade.

Também agradeço aos meus pais, João e Iraci, pelo apoio incondicional e incentivo durante toda a minha vida e, especialmente, durante a longa jornada do doutorado.

Ao meu esposo e amigo Luiz Cesar que, com muito carinho, amor e compreensão, não mediu esforços para que eu chegasse até esta etapa de minha vida. Estendo este agradecimento aos meus filhos, Miguel e Tiago, que esforçaram-se para serem independentes em suas tarefas diárias.

Agradeço também de maneira especial a minha professora orientadora Carmem, pela disponibilidade, pelo convívio, pelos ensinamentos e pela paciência na orientação do meu doutorado.

Agradeço aos professores membros da banca examinadora, que revisaram a minha tese contribuindo com o conteúdo e a finalização do meu trabalho. Também agradeço a todos os professores que direta ou indiretamento contribuíram com o desenvolvimento deste trabalho.

Registro também o meu agradecimento a Capes pela concessão da bolsa de doutorado e à Amazon pelo *grant* recebido para a realização de experimentos.

Não posso deixar de agradecer também aos diversos colegas e amigos que compartilharam comigo muitas histórias durante este período. Em especial aos meus amigos de estudo que sempre me acompanharam e torceram por mim: Diego, Flávio, Marco Aurélio, Rebeca, Patrick, Wendel e Walmir.

Por fim, agradeço imensamente a minha família e amigos por compreenderem minha ausência em muitos momentos. Obrigada pela confiança e palavras de incentivo carinhosas que recebi durante todos estes anos.

Direi do Senhor: Ele é o meu Deus, o meu refúgio, a minha fortaleza, e Nele confiarei.

Salmos 91:2

RESUMO

O modelo de dados RDF vem sendo usado em diversas aplicações devido a sua simplicidade e flexibilidade na modelagem de dados quando comparado aos modelos de dados tradicionais. Dado o grande volume de dados RDF existente atualmente, diversas abordagens de processamento de consultas têm sido propostas visando garantir a escalabilidade destas aplicações. De uma forma geral, estas abordagens propõem métodos de distribuição de dados a fim de promover o processamento distribuído e paralelo de consultas SPARQL em sistemas RDF. Embora a distribuição forneça escalabilidade de armazenamento, o custo de comunicação no processamento de consultas pode ser alto.

Este trabalho propõe uma abordagem de processamento de consultas SPARQL que tem o objetivo de minimizar o custo de comunicação para o processamento de consultas em sistemas RDF distribuídos. A abordagem explora a existência de padrões de alocação (PAs) na distribuição de dados, fornecida por um método de distribuição *controlada* de dados, que determina como triplas RDF são agrupadas e armazenadas em um mesmo servidor. Sendo assim, durante a distribuição, fragmentos de bases RDF seguem a composição de um determinado PA. Logo, a abordagem de processamento proposta gera planos de execução de consultas baseando-se nestes padrões viabilizando a escolha de duas estratégias de comunicação durante o processamento de consultas: *get-frag* e *send-result*. Na primeira estratégia, dada uma consulta, um servidor requisita para servidores remotos fragmentos de dados para a resolução de consultas. Na segunda, o servidor envia resultados intermediários da consulta para outros servidores continuarem a sua execução. Essas estratégias são combinadas em um método, denominado de *2ways*, que escolhe a estratégia de comunicação adequada sempre que a execução de consultas transitar entre fragmentos de dados. A escolha da estratégia depende do número de mensagens e do volume de dados a ser transmitido entre servidores. Resultados experimentais mostram que *2ways* reduz o custo de comunicação de maneira efetiva e melhora o tempo de resposta do processamento de consultas SPARQL em sistemas RDF distribuídos.

Por fim, considerando que bases RDF podem ser alteradas por meio de operações de exclusão/inserção de triplas, este trabalho estende a abordagem de processamento proposta considerando que nem sempre novos dados inseridos estarão de acordo com os PAs pré-definidos. A abordagem de atualização define um tipo especial de PA, denominado de *PaOverflow*, para o armazenamento de dados que não podem ser categorizados pelos PAs existentes. Logo, o *PaOverflow* também deve ser considerado no planejamento e no processamento de consultas. Um estudo experimental inicial mostra que, como esperado, a adoção do *PaOverflow* pode aumentar o tempo de resposta de consultas na abordagem de processamento proposta.

Palavras-chave: RDF, SPARQL, Processamento Distribuído de Consultas, Otimização de Consultas.

ABSTRACT

RDF has been used by many applications due to its simplicity and flexibility in data modeling. Due to the huge volume of RDF data that exists nowadays, many distributed query processing approaches have been proposed aiming to ensure scalability for these applications. In general, these approaches propose data distribution methods promoting distributed and parallel SPARQL query processing. However, while distribution may provide storage scalability, it may also incur high communication costs for processing queries.

This work presents a parallel and distributed query processing approach that aims to minimize the communication cost. The approach explores the existence of data allocation patterns (PAs) for data distribution, provided by a *controlled data distribution method*, that determine how RDF triples should be grouped and stored on the same server. Fragments of the RDF datastore follow a given allocation pattern. The approach generates execution plans based on this distribution model making possible the choice of two communication strategies for query processing: *get-frag* and *send-result*. With the *get-frag* approach, a server requests remote servers to send fragments that contain data required by a query. The *send-result* approach, on the other hand, forwards intermediate results to other servers to continue the query processing. These strategies are combined on a method, called *2ways*, that chooses the adequate communication strategy whenever queries traverse fragment boundaries. The choice of the communication strategy is based on the number of requisitions and the volume of the data to be transmitted. Experimental results show that our proposed technique effectively reduces the communication cost and improves the response time for processing SPARQL queries on a distributed RDF datastore.

Finally, considering that RDF datasets are dynamic, and may be updated by delete/insert operations, this work extends the query processing approach considering that not all newly inserted data may conform to the predefined allocation patterns. We define a special purpose type of PA, called *PaOverflow*, for storing data that can not be categorized by existing PAs. Consequentelly, the *PaOverflow* must be considered in query planning and processing. An initial experimental study shows that, as expected, the *PaOverflow* adoption can increase the response time for processing queries on the proposed processing approach.

Keywords: RDF, SPARQL, Distributed Query Processing, Query Optimization.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

CAPÍTULO 1

INTRODUÇÃO

Nas últimas décadas, a *Web* tem se tornado a maior fonte de aquisição de conhecimento da sociedade contemporânea. Em vista disso, a *Web Semântica* foi proposta como uma nova forma de publicação de dados na *Web* a fim de atribuir significado ao seu conteúdo, tornando-o interpretável tanto por humanos quanto por computadores. O RDF (*Resource Description Framework*) é o modelo de dados padrão da *Web Semântica*[1]. Uma base de dados RDF é um conjunto de triplas (s, p, o), representando que um sujeito s tem uma propriedade p com um objeto o. Considerando que o pode ser o sujeito de uma outra tripla, uma base RDF pode ser vista com um grafo direcionado onde sujeitos e objetos são vértices conectados por suas propriedades. A Figura 1.1 ilustra uma base de dados RDF onde, por exemplo, o recurso *Product1* é o objeto da tripla (*Offer1*, *offers*, *Product1*) e o sujeito da tripla (*Product1*, *name*, "Linea"). A flexibilidade e a simplicidade do modelo motivou a proliferação de bases de dados RDF tais como a *DBPedia*[2], uma base de conhecimento extraída da *Wikepedia*[3]. De acordo com o consórcio *W3C*[4], algumas bases de dados comerciais têm alcançado o tamanho de 1 trilhão de triplas[5]. Sendo assim, o processamento eficiente de consultas em tais bases tem se tornado um grande desafio para os sistemas gerenciadores de base de dados RDF existentes.

Sistemas tais como os propostos por Abadi et al. (2009) e Neumann e Weikum (2010) apresentam abordagens centralizadas para o armazenamento de dados RDF. Porém, este tipo de abordagem penaliza a escalabilidade horizontal em sistemas que processam grandes volumes de dados. Sendo assim, sistemas com armazenamento distribuído têm sido propostos, tais como o de (HUANG; ABADI; REN, 2011), (ZENG et al., 2013) e (GURAJADA et al., 2014). Nestes sistemas, tanto os dados quanto as consultas são distribuídas entre servidores a fim de promover o processamento distribuído e paralelo de consultas. Porém, o processamento distribuído e paralelo implica em custo de comunicação, uma vez que os dados envolvidos no resultado de uma consulta podem estar distribuídos entre múltiplos servidores. Ozsu e Valduriez (2011) destacam que o custo de comunicação neste contexto representa um fator dominante no desempenho da execução distribuída de consultas.

Diferentes métodos de distribuição de dados baseados, por exemplo, em algoritmos de particionamento de grafos (HUANG; ABADI; REN, 2011), em padrões de consultas (GAI; CHEN; WANG, 2015) e em carga de trabalho (SCHROEDER; HARA, 2015), têm sido propostos na literatura visando minimizar o

[1] https://www.w3.org/TR/rdf-primer/
[2] wiki.dbpedia.org
[3] http://www.wikipedia.org
[4] http://www.w3.org
[5] http://www.w3.org/wiki / LargeTripleStores

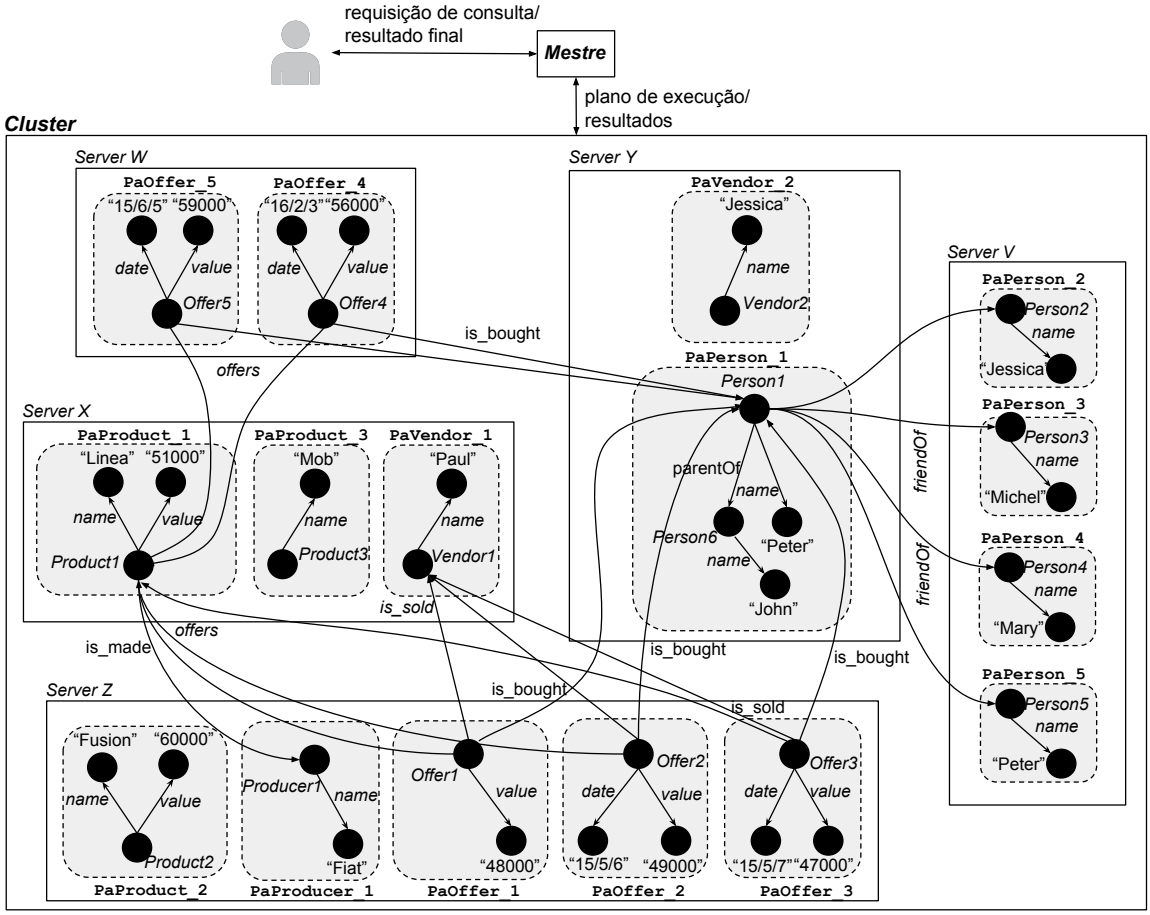


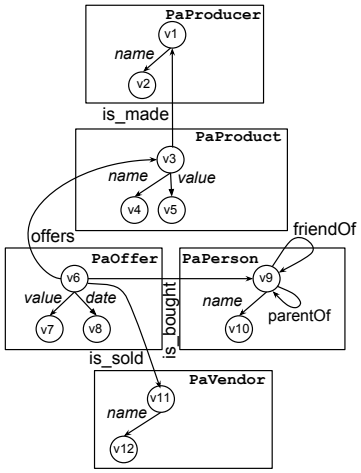
Figura 1.1: Base RDF exemplo

custo inerente à comunicação. Estes métodos procuram alocar dados relacionados em consultas em um menor número de servidores, minimizando a comunicação durante o processamento de consultas.

Este trabalho propõe uma abordagem de processamento distribuído de consultas SPARQL que tem o objetivo de minimizar a carga de comunicação entre servidores explorando *estratégias de alocação e fragmentação controlada*, tais como as estratégias propostas em (SCHROEDER; HARA, 2015) e (MINH-DUC et al., 2015). SPARQL (*SPARQL Protocol and RDF Query Language*) é a linguagem padrão para consultas em bases RDF. Na abordagem proposta, planos de execução de consultas são gerados a partir do conhecimento prévio de como uma base de dados RDF foi particionada. O planejamento de consultas considera que uma base de dados é particionada em fragmentos e que a comunicação entre servidores deve ocorrer quando uma consulta envolve dados armazenados em fragmentos diferentes. Na execução do plano, intuitivamente, após a exploração de um fragmento de dados em um dado servidor, duas estratégias de comunicação podem ser consideradas para dar continuidade na execução: a requisição de dados necessários para outros servidores (*get-frag*) ou o envio de resultados intermediários para outros servidores (*send-result*). A escolha da estratégia depende do número de mensagens e do volume de dados a ser transmitido entre os servidores.

```
1. SELECT ?valueOffer, ?nameProduct,
2. ?namePerson, ?nameFriend
3. FROM <http://exemplo.br/exemplo.rdf>
4. WHERE {
5.    ?product name ?nameProduct .
6.    ?offer offers ?product .
7.    ?offer value ?valueOffer .
8.    ?offer is_bought ?person .
9.    ?person name ?namePerson .
10.   ?person friendOf ?friend .
11.   ?friend name ?nameFriend .
12.   FILTER (valueOffer < 60000)
13. }
```

(a)



(b)

Figura 1.2: Consulta SPARQL (a); Grafo de sumarização da base RDF da Figura 1.1 (b)

Exemplo 1.1 *Para ilustrar estas ideias, considere a base de dados RDF representada pelo grafo da Figura 1.1. Observe que a base está distribuída entre cinco servidores (V, X, Y, W e Z), e que os retângulos tracejados denotam fragmentos de dados. Os fragmentos foram gerados a partir de padrões definidos por um método de distribuição controlada de dados. Neste exemplo, foram considerados cinco padrões, sendo eles: PaProducer, PaProduct, PaOffer, PaPerson e PaVendor. Os padrões estão representados no grafo de sumarização da base RDF na Figura 1.2(b). Perceba que os nodos do grafo de sumarização representam relacionamentos entre entidades presentes no grafo RDF da Figura 1.1, onde cada entidade possui os seus atributos. Considerando o grafo de sumarização da Figura 1.2(b), a base RDF exemplo contém: um fragmento de PaProducer (PaProducer_1), três de PaProduct (PaProduct_1, PaProduct_2 e PaProduct_3), cinco de PaOffer (PaOffer_1, PaOffer_2, PaOffer_3, PaOffer_4 e PaOffer_5), cinco de PaPerson (PaPerson_1, PaPerson_2, PaPerson_3, PaPerson_4 e PaPerson_5); e, por fim, dois fragmentos de PaVendor (PaVendor_1 e PaVendor_2). Neste contexto, um fragmento de dados determina a unidade de co-alocação nos servidores, ou seja, dados de um fragmento encontram-se alocados em um mesmo servidor.*

Agora considere a consulta SPARQL da Figura 1.2(a) para a recuperação de dados relacionados a ofertas com valor < 60.000*. Para estas ofertas, a consulta retorna os seus valores e os nomes de seus produtos, compradores e amigos dos compradores. Os fragmentos de dados envolvidos na consulta são determinados pesquisando as propriedades requeridas na consulta dentro dos padrões do grafo de sumarização da base RDF. Considere um plano de consulta que segue a ordem definida na consulta, começando com o padrão PaProduct (que contém a propriedade name), passando para o padrão PaOffer que obtém o valor da oferta e que segue para o padrão PaPerson por duas vezes a fim de obter o nome do comprador e os nomes de seus amigos.*

Para a execução da consulta, o plano de execução é enviado para todos os servidores para ser processado em paralelo, começando com os fragmentos locais de PaProduct. Entretanto, nesta base de dados em

particular, somente os servidores X e Z contêm fragmentos de PaProduct. A partir destes fragmentos, os nomes "Linea" e "Mob" são extraídos de X e o nome "Fusion" de Z. Para continuar o processamento da consulta, somente Product1 continua no processamento, dado que Product2 e Product3 não possuem ofertas. Sendo assim, fragmentos de PaOffer armazenados nos servidores W e Z são requeridos por X. Neste ponto, X deve escolher a estratégia de comunicação que usará com os servidores remotos. Neste caso, existe um único resultado intermediário a ser enviado, e cinco fragmentos de PaOffer a serem recuperados. Logo, o servidor X escolhe a estratégia send-result, e envia o nome do produto para os servidores W e Z, os quais continuam com o processamento da consulta em paralelo. Ambos os servidores, depois de processarem os fragmentos de PaOffer, detectam que suas ofertas estão relacionadas à um único fragmento de PaPerson, o qual está localizado no servidor Y. Sendo assim, cada servidor deve escolher uma estratégia de comunicação para continuar a execução da consulta. Neste caso, o tamanho dos resultados intermediários é maior que o fragmento de PaPerson e então ambos os servidores escolhem a estratégia get-frag, requisitando que o servidor Y envie o fragmento de dados requerido no processamento da consulta. Finalmente, ambos os servidores processam o fragmento PaPerson_1 e escolhem a estratégia send-result para enviar os seus resultados intermediários para o servidor V. Esta estratégia é escolhida porque o tamanho dos quatro fragmentos de dados de PaPerson armazenados em V é maior do que os resultados intermediários de ambos os servidores. Quando a execução é finalizada após o processamento dos amigos de Person1, V envia os resultados finais para o servidor que requisitou a consulta. □

As duas estratégias de comunicação, *send-result* e *get-frag*, foram implementadas em um sistema de processamento de consultas SPARQL baseado em um algoritmo de exploração distribuída de grafos. Um estudo experimental mostra que a possibilidade de escolha da estratégia de comunicação apresenta melhor desempenho do que o uso de cada estratégia de maneira isolada durante o processamento distribuído de consultas. Conforme o nosso conhecimento, este é o primeiro modelo de execução *paralelo e distribuído* que combina mais do que uma estratégia de comunicação a fim de otimizar o processamento de consultas.

Além disso, este trabalho também propõe uma abordagem de atualização de dados que dá suporte à abordagem de processamento de consultas descrita, uma vez que a inserção de novas triplas pode influenciar na composição dos padrões do grafo de sumarização de uma determinada aplicação RDF. A abordagem de atualização considera as operações de inserção e de exclusão de triplas. A operação de exclusão elimina dados sem implicar alterações no grafo de sumarização. Já, a operação de inserção pode estender o grafo de sumarização a fim de categorizar novas triplas em uma dada aplicação. A operação de inserção também conta com um tipo de padrão, denominado de *PaOverflow*. Basicamente, dados que não são categorizados por meio dos padrões do grafo de sumarização da aplicação são armazenados por meio do *PaOverflow*. Consequentemente, dados armazenados por meio do *PaOverflow* também deve ser explorados no processamento de consultas. Um estudo experimental inicial mostra que, como esperado, a adoção do *PaOverflow* no processamento de consultas pode aumentar o tempo de resposta de consultas.

Em suma, as principais contribuições desta tese são:

- Uma abordagem para o processamento de consultas SPARQL em sistemas RDF que têm os seus dados distribuídos por meio de um método de distribuição *controlada* de dados;
- Um método de comunicação híbrido, denominado de *2ways*, que viabiliza a escolha entre duas estratégias de comunicação, *send-result* e *get-frag*, a fim de minimizar o custo de comunicação entre servidores durante o processamento distribuído de consultas SPAQL;
- Uma abordagem para a atualização de bases RDF para sistemas que adotam a abordagem de processamento de consultas proposta neste trabalho.

O restante deste trabalho está organizado da seguinte maneira. O Capítulo 2 introduz os conceitos básicos relacionados a este trabalho. O Capítulo 3 apresenta os trabalhos relacionados. A abordagem de processamento de consultas proposta é apresentada nos Capítulos 4. O Capítulo 5 apresenta a abordagem de atualização de dados citada anteriormente. Continuando, o Capítulo 6 mostra experimentalmente o impacto do método *2ways* e da abordagem de atualização no processamento distribuído de consultas. Por fim, o Capítulo 7 apresenta a conclusão deste trabalho.

CAPÍTULO 2

PRELIMINARES

Este capítulo apresenta os conceitos centrais relacionados às abordagens propostas neste trabalho. Eles estão organizados da seguinte forma: nas duas primeiras seções são apresentados os principais conceitos relacionados à RDF e SPARQL; na terceira seção, um algoritmo de isomorfismo de subgrafos é apresentado com o objetivo de embasar o modelo de execução adotado na abordagem de processamento; a quarta seção descreve os principais conceitos relacionados ao modelo de distribuição de dados adotado neste trabalho; e, por fim, a última seção relata sobre o dinamismo inerente aos esquemas de bases RDF.

2.1 RDF

O RDF (*Resource Description Framework*) é um modelo de dados para representar informações sobre recursos. Os recursos podem ser qualquer coisa incluindo documentos, pessoas, objetos físicos e conceitos abstratos[1]. A sintaxe do RDF é baseada no conceito de triplas da forma *(sujeito,predicado,objeto)*, onde uma tripla representa que um sujeito tem uma propriedade com um objeto.

Uma base RDF consiste de um conjunto finito de triplas que pode ser representado por meio de um grafo $G_D = (V_D, E_D)$, onde (1) V_D é o conjunto de vértices que representam os sujeitos e os objetos da base, e (2) E_D é o conjunto de arestas direcionadas e rotuladas que representam os predicados que relacionam pares de vértices do conjunto V. A Figura 1.1 mostra o exemplo de uma base RDF no contexto de ofertas de produtos de uma loja *online* considerada em (BIZER; SCHULTZ, 2009). Neste exemplo, a aresta *is_make* relaciona o vértice do sujeito *Product1* com o vértice do objeto *Producer1*. Ainda, o mesmo vértice que assume o papel de objeto, o *Producer1*, também é sujeito na tripla *(Producer, name, "Fiat")*.

Conceitualmente, os vértices podem ser do tipo: IRI (*Internationalized Resource Identifier*), literal ou *blank node*. IRIs referenciam recursos e devem ser sintaticamente absolutos[2]. Logo, um IRI referencia um único recurso. Literais são usados para a representação de números, cadeia de caracteres e datas. *Blank nodes* são vértices que não possuem referências para IRIs. Um *blank node*, por exemplo, pode ser usado para agregar valores de uma propriedade composta. IRIs podem ser usados nos três membros de uma tripla, *blank nodes* em sujeitos e objetos, e literais somente em objetos. Quanto as arestas, elas podem ser somente do tipo IRI. A fim de simplificar os exemplos apresentados neste trabalho, adotou-se uma sintaxe simplificada do RDF onde, por exemplo, a tripla *(<http://www.exemplo/Product3>*

[1] http://www.w3.org/TR/2014/NOTE-rdf11-primer-20140225/
[2] http://www.ietf.org/rfc/rfc3987.txt

<http://www.exemplo/name> "mob") é representada por *(Product3 name "mob")*. Além disso, valores literais são denotados por aspas diferenciado-os de recursos. *Blank nodes* não são considerados neste trabalho.

Extensões para o vocabulário básico do RDF foram propostas com o objetivo de enriquecer semanticamente o modelo. Um exemplo é o *RDF-Schema*[3] que pode ser usado para a definição de termos empregados na declaração de fatos. O *RDF-Schema* possibilita definição de esquemas de bases por meio de atributos de recursos e relacionamentos entre recursos. De acordo com (GUTIERREZ; HURTADO; VAISMAN, 2006) as classes *rdfs:Class* e *rdf:Property* e as propriedades *rdfs:range*, *rdfs:domain*, *rdfs:type*, *rdfs:subClassOf* e *rdfs:subPropertyOf* representam as características essenciais do RDF. Basicamente, por meio de *rdfs:Class*, *rdfs:subClassOf*, *rdf:Property* e *rdf:SubPropertyOf* são definidas classes, subclasses, propriedades e subpropriedades envolvidas em um domínio, respectivamente, e a adesão de um recurso à uma classe é feita por meio da propriedade *type*. Por fim, as propriedades *rdfs:range* e *rdfs:domain* restringem os valores que os sujeitos e objetos relacionados por uma propriedade podem assumir. O termo *estrutura RDF* usado neste trabalho refere-se a estruturas compostas somente por relacionamentos entre classes, onde cada classe possui os seus atributos. Sendo assim, propriedades tais como *SubClassOf* e *SubPropertyOf* não são consideradas em estruturas RDF.

O armazenamento de grafos RDF pode ser baseado no modelo nativo de grafos ou no modelo não-nativo. No primeiro tipo, cada recurso do grafo RDF (sujeito ou objeto) é modelado como um vértice do grafo associado aos seus vértices adjacentes de forma que estes sejam alcançados pelas arestas incidentes em ambos. Logo, mecanismos como listas de adjacências podem ser usados no armazenamento físico de dados possibilitando a exploração do grafos de forma simples e direta. Já no segundo tipo, triplas RDF *(sujeito,predicado,objeto)* são armazenadas seguindo a idéia de tuplas do modelo relacional. Um exemplo são os sistemas classificados como *triple stores* que adotam um esquema de três colunas, sendo uma para cada membro da tripla RDF.

Este trabalho baseia-se no modelo nativo de grafos no qual uma base RDF é definida como um conjunto de vértices com suas listas de adjacência. Cada vértice que representa um recurso possui um identificador único baseado em seu IRI. Para cada vértice v é definida uma tupla *(val,in,out)*, onde: *val(v)* é um valor literal, *in(v)* é o conjunto de arestas incidentes em v, e *out(v)* é o conjunto de arestas que partem de v. Por exemplo, no grafo RDF da Figura 1.1, os vértices em *PaProducer_1* são definidos como: *Producer1 = (val: undefined, in:{(is_made, Product1)}, out: {(name,* v_{Fiat}*)}*), e v_{Fiat} *= (val: "Fiat", in:{(name, Producer1)}, out:{}).*

[3]http://www.w3.org/TR/rdf-schema/

2.2 SPARQL

SPARQL (*SPARQL Protocol and RDF Query Language*) é a linguagem padrão recomendada pelo consórcio W3C para consultas em bases RDF. O núcleo da sintaxe da linguagem é baseado em um conjunto de padrões de triplas semelhantes às triplas RDF, que admitem variáveis em seus termos (ZENG et al., 2013). As variáveis são representadas por pontos de interrogação (?).

Do ponto de vista sintático, a estrutura da linguagem pode ser dividida em três blocos (ARENAS; GUTIERREZ; PÉREZ, 2009):

- Cláusula SELECT: especifica como os resultados de uma consulta serão retornados. As opções são: SELECT, que retorna os resultados da consulta em forma de tabela; ASK, que verifica se há pelo menos uma resposta e retorna um resultado lógico (verdadeiro ou falso); CONSTRUCT, que constrói um novo grafo a partir do resultado de uma consulta; e DESCRIBE, que retorna recursos associados à um determinado recurso especificado na consulta.

- Cláusula FROM: especifica uma ou mais bases de dados a serem pesquisadas na consulta.

- Cláusula WHERE: especifica, por meio de um ou mais padrões de triplas, o padrão de grafo a ser pesquisado na base RDF definida na cláusula FROM. Padrões complexos são permitidos, os quais podem ser formados pelo uso de alguns operadores como FILTER, OPTIONAL, MINUS, BIND e GROUP BY[4].

O conjunto de padrões de triplas de uma consulta constitui um padrão de grafo que é classificado como o padrão de grafo básico (PGB) da consulta. Pérez, Arenas e Gutierrez (2009) apresenta uma sintaxe algébrica para expressar padrões de grafos SPARQL, considerando padrões de grafos para bases RDF simples, desconsiderando, por exemplo, o vocabulário *RDF-Schema* entre outras extensões para RDF. A sintaxe também se limita à um fragmento da gramática da linguagem que contém os operadores considerados centrais da linguagem. Logo, uma expressão de padrões de grafos SPARQL pode ser definida recursivamente como se segue:

1. Um padrão de tripla é um padrão de grafo.

2. Se P1 e P2 são padrões de grafos, então as expressões (P1 AND P2), (P1 OPT P2) e (P1 UNION P2) são padrões de grafos. O operador AND é representado por um ponto (.) na sintaxe SPARQL.

3. Se P é um padrão de grafo e X é um URI ou uma variável, então (X GRAPH P) é um padrão de grafo.

4. Se P é um padrão de grafo e R é uma expressão de filtro SPARQL, então (P FILTER R) é um padrão de grafo. Uma condição de filtro pode ser construída, por exemplo, por meio de constantes,

[4]http://www.w3.org/TR/rdf-sparql-query/

conectivos lógicos, símbolos de desigualdade e de igualdade. Outras condições de filtro estão listadas em *http://www.w3.org/TR/2013/REC-sparql11-query-20130321/*.

Essencialmente, SPARQL é uma linguagem de consulta de casamento de grafos, onde dada uma base de dados G_D e o PGB de uma consulta, a avaliação da consulta consiste na pesquisa do PGB em G_D, e os valores obtidos desta pesquisa são processados na computação do resultado da consulta (PÉREZ; ARENAS; GUTIERREZ, 2009). Basicamente, para a avaliação de uma consulta SPARQL deve ocorrer os mapeamentos (μ_i) entre as variáveis dos padrões de triplas de seu PGB e as triplas da base a ser consultada. Desta forma, é possível definir os domínios (dom(μ_i)) de cada variável para o processamento da consulta. Porém, considerando os operadores binários da sintaxe algébrica apresentada anteriormente, a avaliação da linguagem deve considerar que:

- O operador AND representa a operação de *junção* entre P1 e P2. A avaliação da junção exige que o mapeamento μ_1 de P1 seja extensível com o mapeamento μ_2 de P2, ou seja, que a variável de P1 seja igual à variável de P2 (P1(?x) = P2(?x)). Logo, dom(μ_1) e dom(μ_2) são considerados no resultado da operação.

- O operador OPT representa a operação de *junção externa à esquerda* e sua avaliação é similar à junção, porém a extensão de μ_1 com μ_2 não é obrigatória. Sendo assim, os dois domínios ou somente o dom(μ_1) são considerados no resultado da operação.

- O operador UNION representa a operação usual de *união* de conjuntos. O resultado considera os dois domínios.

- O operador GRAPH possibilita a troca de bases de dados durante o processamento da consulta, essencial para consultas que utilizam mais que uma base.

- O operador FILTER estabelece uma condição que deve ser verdadeira para que triplas de um mapeamento sejam consideradas na consulta.

A Figura 1.2(a) ilustra o tipo de consultas SPARQL explorado neste trabalho, ou seja, consultas que adotam a cláusula SELECT, o operador AND (.), o operador FILTER e a possibilidade de variáveis somente em sujeitos e objetos de padrões de triplas. Além disso, os grafos das consultas devem ser conexos. O SELECT (linhas 1 e 2) define que o resultado da consulta será uma tabela com o valor da oferta e seus respectivos dados: nome do produto, nome do comprador e nome do amigo do comprador. A cláusula FROM (linha 3) especifica a base considerada na consulta. A cláusula WHERE (linhas 4-11) especifica os padrões de triplas da consulta que estão conectados entre si por meio de alguma variável em comum e que devem ser combinados por meio de operações de junção representadas por pontos (.). Por fim, o operador FILTER define restrições para a recuperação de triplas RDF (linha 12). A Figura 2.1

representa o PGB da consulta da Figura 1.2(a), onde os sujeitos e os objetos dos padrões de triplas são representados por vértices e os predicados por arestas. Considerando a base RDF exemplo, uma resposta da consulta seria: *("Linea", 48.000, "Peter", "Jessica").*

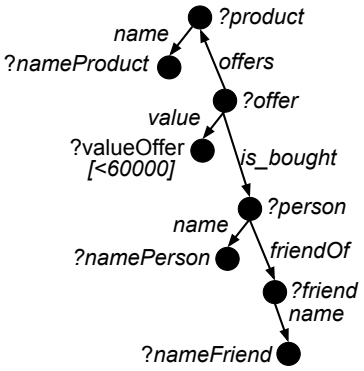


Figura 2.1: Padrão de grafo básico da consulta da Figura 1.2(a)

Em relação à diversidade estrutural dos PGBs, os padrões de tripla podem ser combinados na estrutura de estrela (Figura 2.2(a)), na estrutura de sequência ou linear (Figura 2.2(b)) ou podem compor uma estrutura híbrida combinando as duas primeiras estruturas (Figura 2.2(c)) (ALUÇ et al., 2013). Consultas com a primeira estrutura combinam somente atributos de uma único recurso em um PGB. Consultas com a segunda estrutura consideram diferentes tipos de recursos em um mesmo PGB.

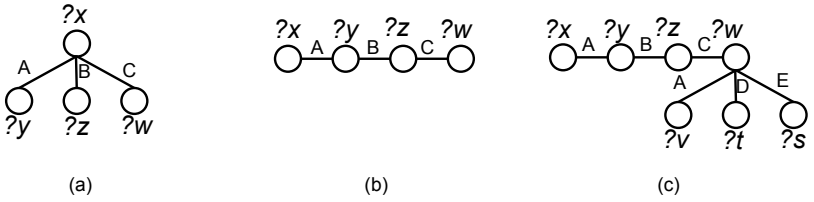


Figura 2.2: Estrutura estrela (a); Estrutura linear (b); Estrutura híbrida (c)

2.2.1 SPARUL

Considerando a atualização de grafos RDF, (SEABORNE et al., 2008) propõe a linguagem *SPARQL/Update* apelidada de *SPARUL* que permite a inserção e a exclusão de triplas em grafos RDF. Basicamente, considerando um repositório RDF, a linguagem de atualização fornece as seguintes operações:

- INSERT DATA: usada para adicionar triplas em um grafo;
- DELETE DATA: usada para remover triplas de um grafo;
- LOAD: usada para ler o conteúdo de um documento que representa um grafo para outro grafo;
- CLEAR: usada para remover todas as triplas (uma ou mais) de um grafo.

```
INSERT DATA
{
        <http://exemplo/Offer1> <http://exemplo/date> "15/4/8"
}
```

Figura 2.3: Sintaxe da linguagem SPARUL para inserir triplas em grafos RDF

A Figura 2.3 ilustra a sintaxe da linguagem para a inserção da tripla (<http://exemplo/offer1>, date, "15/4/8") em uma determinada base padrão definida em uma aplicação. Esta sintaxe é usada nos exemplos de inserção e exclusão de triplas na abordagem de atualização de dados do Capítulo 5.

2.3 Casamento de subgrafos

Casamento de padrões de grafos é uma das classes de problemas de grafos mais importante e amplamente estudada (NISAR; FARD; MILLER, 2013). Conceitualmente, algoritmos de casamento de grafos procuram encontrar subgrafos em um grafo que são similares à um grafo de uma dada consulta. O processamento de consultas SPARQL pode ser transformado para um problema de casamento de grafos onde subgrafos de uma base RDF que são homomórficos ao PGB de uma dada consulta G_Q são recuperados. Logo, dado que pares de vértices (u, v) de G_Q são conectados por propriedades $p - (u, v, p) \in E_Q -$ e E_D conecta vértices em G_D, o homomorfismo é definido com uma função f tal que $(p, f(v))$ está no conjunto de arestas que partem de $f(u)$ sempre que $(u, v, p) \in E_Q$. Sendo assim, o casamento entre dois grafos $G_1 = (V_1, E_1)$ e $G_2 = (V_2, E_2)$ consiste em um mapeamento M que associa vértices de G_1 com vértices de G_2.

A abordagem de processamento de consultas apresentada no Capítulo 4 baseia-se no algoritmo de isomorfismo de subgrafos proposto por (ULLMANN, 1976), como (ALUÇ et al., 2013) e (ZENG et al., 2013). No caso de isomorfismo de subgrafos, um mapeamento $M \subset V_1 \times V_2$ é dito isomórfico se os vértices de G_1 forem mapeados para diferentes vértices em G_2. Em contrapartida, o homomorfismo de subgrafos relaxa esta restrição injetora do casamento isomórfico entre grafos (KIM et al., 2015).

A proposta de Ullmann (1976) é implementada como um algoritmo que incrementa soluções parciais ou abandona soluções ao detectar soluções parciais que não podem ser completadas. O seu algoritmo filtra os vértices do grafo G_2 por meio de seus rótulos e os vértices de G_1 são escolhidos aleatoriamente. A escolha desordenada influencia negativamente no desempenho do algoritmo, pois a escolha de vértices que não possuem adjacência com vértices já considerados no mapeamento ocasiona um processamento desnecessário. Visando melhorar o desempenho do algoritmo proposto em (ULLMANN, 1976), vários algoritmos, tais como VF2 (CORDELLA et al., 2001), QuickSI (SHANG et al., 2008) e SPath (ZHAO; HAN, 2010), têm explorado, por exemplo, diferentes ordens de junção e informações auxiliares para a eliminação de candidatos falsos-positivo o quanto antes durante o processamento de casamento de subgrafos.

Lee et al. (2012) apresenta um pseudo-código geral baseado em (ULLMANN, 1976) (Algoritmo 1) para representar as principais etapas de um algoritmo de casamento de subgrafos. As entradas do algoritmo são um grafo de consulta G_Q e um grafo de dados G_D, e a saída é um conjunto de subgrafos isomórficos de G_Q em G_D. Um subgrafo isomórfico é representado por meio de uma lista M de pares onde cada par possui um vértice de G_Q e um vértice de dado correspondente de G_D.

Para dar início ao processo, primeiramente, os conjuntos de vértices de dados candidatos (*C(u)*) para cada vértice u ($u \in G_Q$) são selecionados (linha 2). Se *C(u)* é vazio, então a pesquisa pode ser finalizada, pois não há dados da base relacionados à u (linhas 3 e 4). Após a seleção de candidatos, o algoritmo invoca uma subrotina recursiva, *PesquisarSubgrafos*, para encontrar os pares de mapeamento de cada vértice u para vértices de dados de G_D (linha 7). *PesquisarSubgrafos* tem como parâmetros de entrada G_Q, G_D, e um mapeamento parcial M. A subrotina reporta todos os mapeamentos de G_Q em G_D.

A recursão da subrotina *PesquisarSubgrafos* é interrompida quando é encontrada a solução completa para o mapeamento (quando $|M| = |V(G_Q)|$) (linha 9). Enquanto isso não acontece, a subrotina requisita o próximo vértice u que ainda não foi combinado no mapeamento (linha 12) e refina os vértices candidatos de u (*C(u)*) por meio de regras de corte gerando o conjunto C_R (linha 13). Para cada vértice candidato ($v \in C_R$) que ainda não foi combinado no resultado da consulta (linha 14), a subrotina *ÉCompatível* verifica se as arestas entre u e vértices de G_Q já combinados no mapeamento parcial têm arestas correspondentes entre v e vértices de dados de G_D, também já combinados (linha 15). Se v é qualificado, ele forma um par com u, e o mapeamento M é atualizado (linha 16) com a adição do novo par de vértices.

O algoritmo continua com o casamento dos vértices restantes da consulta G_Q chamando recursivamente *PesquisarSubgrafos* (linha 17). Por fim, todas as alterações feitas em M são reorganizadas em *ReestruturarMapeamento* (linha 18), onde as soluções parciais que não foram completadas são excluídas do mapeamento final.

Algoritmo 1: Isomorfismo de subgrafos

```
   Entrada: grafo G_Q; grafo G_D
   Saída: Mapeamentos referentes aos subgrafos isomórficos de G_Q em G_D
 1 M:= { }; for u ∈ V(G_Q) do
 2 |   C(u) := FiltrarCandidatos(G_Q, G_D, u, ...);
 3 |   if C(u) = { } then
 4 |   |   finaliza;
 5 |   end
 6 end
 7 PesquisarSubgrafos(G_Q, G_D, M, ...);
 8 Subrotina PesquisarSubgrafos(G_Q, G_D, M, ...)
 9 |   if |M| = |V(G_Q)| then
10 |   |   reportar M;
11 |   else
12 |   |   u := PróximoVérticeConsulta(...);
13 |   |   C_R := RefinarCandidatos(M, u, C(u), ...);
14 |   |   for v ∈ C_R tal que v ainda não foi combinado do
15 |   |   |   if ÉCompatível(G_Q, G_D, M, u, v, ...) then
16 |   |   |   |   AtualizarMapeamento(M, u, v, ...);
17 |   |   |   |   PesquisarSubgrafos(G_Q, G_D, M, ...);
18 |   |   |   |   ReestruturarMapeamento(M, u, v, ...);
19 |   |   |   end
20 |   |   end
21 |   end
```

Voltando à questão do homomorfismo, a restrição imposta na linha 14 do Algoritmo 1 não é considerada e um vértice de G_D pode participar de um mapeamento de M quantas vezes for necessário. Além disso, note que a presença de variáveis em PGBs exige adaptações no algoritmo para que rótulos de arestas sejam utilizados na seleção de vértices a serem pesquisados durante a avaliação de uma consulta SPARQL. Sendo assim, por exemplo, AluÇ et al. (2013) considera o conceito de a *compatibilidade* entre uma aresta do grafo RDF e uma aresta do PGB da consulta no algoritmo de casamento de grafos. A compatibilidade existe quando as duas arestas possuem o mesmo rótulo ou quando a aresta do PGB é uma variável.

2.4 Distribuição de dados

O custo de uma estratégia de processamento distribuído pode ser expresso por meio do tempo total ou do tempo de resposta de uma dada consulta. O tempo total da consulta é a soma do tempo de todos os componentes dos servidores envolvidos no processamento da consulta, ou seja, o custo de processamento (CPU), de E/S e de comunicação de cada servidor. O tempo de resposta é o tempo decorrido do início até a conclusão da consulta. Considerando o paralelismo no processamento, a sequência de instruções mais custosa determina o custo da consulta (OZSU; VALDURIEZ, 2011).

A troca de dados entre servidores durante o processamento distribuído de consultas implica diretamente no custo de comunicação do processamento de consultas (HUANG; ABADI; REN, 2011). Segundo Ozsu e Valduriez (2011) o custo de comunicação neste contexto representa um fator dominante no desempenho da execução distribuída de consultas. Logo, métodos de distribuição de dados têm sido usados para a redução de comunicação entre servidores. Em geral, o particionamento de banco de dados é definido por duas etapas complementares que compreendem a fragmentação e a alocação de dados (ZILIO, 1998). A fragmentação divide o banco de dados em fragmentos de acordo com um modelo específico. A fragmentação adequada de dados em sistemas distribuídos pode melhorar o desempenho de consultas considerando que dados correlacionados em uma dada consulta estarão alocados no mesmo servidor. Logo, o custo de comunicação entre servidores pode ser reduzido durante o processamento de consultas. Em seguida, a alocação de dados envolve encontrar uma distribuição adequada para os fragmentos de dados no *cluster* podendo satisfazer restrições que envolvam o tempo de resposta, de armazenamento e de processamento. Com uma alocação apropriada de fragmentos, a vazão do sistema pode ser melhorada em termos do processamento paralelo de consultas sobre fragmentos alocados em servidores distintos (OZSU; VALDURIEZ, 2011).

Segundo Schroeder (2014), os modelos de fragmentação geralmente dão suporte à fragmentação horizontal e vertical. Fragmentos horizontais correspondem a um agrupamento baseado em valores de sujeitos ou objetos de triplas. Os métodos *hash partitioning*, *range partitioning* e *round-robin* são os mais elemen-

tares dentre os métodos de fragmentação horizontal e que tem em comum a escolha por uma chave de particionamento. No método baseado em *hash* uma função de espalhamento é aplicada sobre a chave de particionamento para agrupar os dados em fragmentos. Quando o método baseado em *range* é aplicado, o conjunto de dados é agrupado por intervalos de valores assumidos pela chave de particionamento. A abordagem *round-robin* é um método aleatório para distribuir os dados em fragmentos gerados. Já, a fragmentação vertical considera predicados de triplas em seus agrupamentos. Este conceito foi introduzidos pelo projeto Jena (WILKINSON et al., 2003) através da criação de tabelas de propriedades que ficou conhecido como *property tables*. Por fim, os dois modelos de fragmentação também pode ser combinados compondo um modelo híbrido.

Continuando, a fragmentação também pode ser baseada em heurísticas sobre a composição estrutural de grafos RDF ou baseada em carga de trabalho. O particionador de grafos METIS (Karypis Lab, 2014) é um exemplo de solução do primeiro tipo. Metis é um algoritmo que particiona grafos visando minimizar a quantidade de cortes entre as partições. Em contrapartida, soluções do segundo tipo consideram um grupo de consultas frequente à uma determinada aplicação e define a fragmentação do grafo de forma a favorecer o desempenho deste grupo de consultas.

Quanto a alocação, segundo Hauglid, Ryeng e Nørvåg (2010), soluções podem ser classificadas em redundantes ou não-redundantes, balanceadas ou não-balanceadas, e estáticas ou dinâmicas. A replicação de dados (redundância) é usada para a melhoria do desempenho de consultas e no balanceamento de cargas de trabalho não-uniformes. Além disso, a alocação pode ser estática ou dinâmica com relação as alterações da carga de trabalho ou da composição do ambiente distribuído.

2.4.1 Distribuição *controlada* de dados

No contexto deste trabalho, métodos de fragmentação que usam padrões pré-definidos no particionamento de bases são classificados como *métodos controlados*. Trabalhos tais como (SCHROEDER, 2014) e (MINH-DUC et al., 2015) consideram métodos deste tipo.

Schroeder (2014) apresenta uma abordagem para a fragmentação de dados baseada em carga de trabalho que define heurísticas para identificar afinidades entre dados de uma base RDF e estabelecer o agrupamento de itens fortemente relacionados em um mesmo servidor. As relações de afinidades detectadas são usadas para a definição de subgrafos disjuntos na estrutura RDF explícita de uma base RDF, onde uma estrutura é representada por suas classes (com seus respectivos atributos) e as associações existentes entre as classes. Estes subgrafos são usados como padrões para a fragmentação da base RDF em um ambiente distribuído e cada fragmento de dados é categorizado por meio do subgrafo da estrutura RDF que o gerou. Os conceitos *padrão de alocação (PA)* e *grafo de sumarização* adotados na abordagem de processamento do Capítulo 4 estão diretamente relacionados aos conceitos definidos em (SCHROEDER, 2014). PAs referem-se aos subgrafos usados na fragmentação de uma base RDF. O conceito de grafo

de sumarização baseia-se no conceito de estrutura RDF adotado em (SCHROEDER, 2014). Porém, o método usado para a definição de padrões é indiferente para a abordagem de processamento proposta.

Continuando, Minh-Duc et al. (2015) baseia-se no conceito de *characteristic sets* (CS) na fragmentação de bases. Como métodos baseados em *hash*, este conceito assegura que triplas que compartilham um mesmo sujeito em uma base são alocadas em um mesmo servidor. A composição de CSs baseia-se na propriedade *type* do *RDF-Schema*, onde cada sujeito de um fragmento de dado representa um recurso da base RDF. Desta forma, os fragmentos de dados gerados por meio de CSs podem ser categorizados por meio da estrutura RDF implícita da base.

2.5 Dinamismo de estruturas RDF

Bases RDF são classificadas como semi-estruturadas por oferecerem estruturas flexíveis que podem ser atualizadas conforme a dinâmica de suas aplicações (ABITEBOUL; HULL; VIANU, 1995). Este tipo de base pode ter sua estrutura alterada conforme seus dados são atualizados. Logo, sistemas RDF com funcionalidades baseadas em estruturas RDF devem prover mecanismos que dão suporte a este dinamismo.

Trabalhos tais como (GUTIERREZ; HURTADO; VAISMAN, 2006), (CHIRKOVA; FLETCHER, 2009), (GUTIERREZ; HURTADO; VAISMAN, 2011) e (FLOURIS et al., 2013) tratam da evolução de esquemas RDF usando o vocabulário do *RDF-Schema*. Considerando que triplas podem ser inseridas e excluídas de bases RDF, toda requisição de alteração em uma base deve ser implementada implicando em alterações válidas e mínimas na base (GUTIERREZ; HURTADO; VAISMAN, 2011) (CHIRKOVA; FLETCHER, 2009). Os três primeiros trabalhos focam na operação de exclusão de dados. Relacionamentos tais como *subClassOf* e *subPropertyOf* do *RDF-Schema* exigem uma análise cuidadosa da base, dado que uma abordagem de evolução deve assegurar que, triplas excluídas não podem ser deriváveis na base depois de uma operação de exclusão (GUTIERREZ; HURTADO; VAISMAN, 2011). Em contrapartida, o processo da operação de adição de triplas é considerado bem comportado e trivial. A operação de adição simplesmente considera a união de uma nova tripla com a base RDF (CHIRKOVA; FLETCHER, 2009) (GUTIERREZ; HURTADO; VAISMAN, 2011).

Gutierrez, Hurtado e Vaisman (2006) e Chirkova e Fletcher (2009) consideram que a evolução de esquemas RDF está intimamente relacionada com a atualização de dados, uma vez que metadados e dados estão intrinsecamente alocados em bases RDF. Gutierrez, Hurtado e Vaisman (2006) propõe um algoritmo capaz de *deduzir* as consequências de uma operação de exclusão em uma base a fim de garantir alterações mínimas na base. Chirkova e Fletcher (2009) define uma noção de determinismo detectando operações de exclusão que podem ser resolvidas de maneira única e bem definida. Em contrapartida, (GUTIERREZ; HURTADO; VAISMAN, 2011) e (FLOURIS et al., 2013) tratam de alterações de instâncias da base e de esquemas RDF de maneira isolada, onde alterações em instâncias não interferem em esquemas.

Gutierrez, Hurtado e Vaisman (2011) propõem um algoritmo determinístico para a exclusão de instâncias e uma redução para o problema não-determinístico de atualização de esquemas. Por fim, Flouris et al. (2013) apresenta um *framework* que foca no uso de regras de restrições de integridade na atualização de bases RDF. O trabalho baseia-se nos princípios de *sucesso, validade* e *mudança mínima* relacionados ao conceito de *revisão de crenças*. O mesmo princípio é usado para alterações em esquemas.

O objetivo da abordagem de atualização proposta neste trabalho é o de viabilizar o processamento de consultas em bases RDF que possam ser atualizadas. A operação de inserção de dados associa um método de extensão limitada de estruturas RDF com um método de armazenamento secundário de dados. Aqui a estrutura RDF é estendida por meio de anotações, onde a estrutura inicial não é alterada. A operação de exclusão de dados da abordagem simplesmente remove triplas de uma base RDF independentemente da sua estrutura RDF. A evolução da estrutura RDF que descreve a distribuição controlada dos dados não é o foco desta tese. No entanto, são desenvolvidos mecanismos que permitem a inserção de dados que não estão de acordo com a estrutura RDF inicial para dar suporte à natureza semi-estruturada do modelo.

A operação de inserção apresentada no Capítulo 5 baseia-se na abordagem de armazenamento de dados proposta em STORED (DEUTSCH; FERNANDEZ; SUCIU, 1999). O objetivo de STORED é o armazenamento de bases semi-estruturadas XML em bases relacionais. Logo, STORED propõe uma abordagem para a definição de esquemas relacionais a partir de bases XML. Porém, nem sempre o mapeamento XML-relacional é completo e, consequentemente, o esquema relacional não é capaz de armazenar os dados XML integralmente. Sendo assim, os dados XML que não se adequam ao esquema relacional definido por STORED são armazenados em um repositório secundário denominado de *Overflow*. Durante a fase de planejamento e de execução de consultas tanto a base relacional quanto o *Overflow* devem ser considerados.

Considerações

Em geral, sistemas RDF de grande porte adotam o processamento paralelo e distribuído a fim de melhorar o desempenho de suas tarefas. Porém, o custo de comunicação inerente ao processamento distribuído pode implicar na redução da velocidade de execução de consultas nesses sistemas. Em geral, abordagens de distribuição de dados têm sido usadas por estes sistemas a fim de minimizar o custo de comunicação. Além disso, alguns sistemas minimizam este custo reduzindo a geração de resultados intermediários inválidos durante o processamento de consultas. Entende-se por resultados intermediários inválidos os resultados intermediários que não irão compor o resultado final da consulta. O capítulo seguinte trata do estado da arte com relação a abordagens aplicadas ao contexto de processamento de consultas SPARQL em bases RDF distribuídas.

CAPÍTULO 3

ABORDAGENS PARA O PROCESSAMENTO DISTRIBUÍDO DE CONSULTAS SPARQL

Este capítulo apresenta trabalhos relacionados ao tema central desta tese que é a otimização de consultas distribuídas. Diferentes métodos têm sido usados em sistemas RDF distribuídos a fim de minimizar o custo de comunicação no processamento distribuído. Este trabalho propõe um modelo *híbrido* de comunicação como um método de otimização para consultas SPARQL em bases RDF distribuídas. De acordo com o nosso conhecimento, nenhum outro trabalho na literatura adota um modelo *híbrido* de comunicação em um modelo de execução de consultas *paralelo* e *distribuído*.

Os trabalhos relacionados a seguir adotam uma arquitetura Mestre-Escravo *shared-nothing*, onde cada servidor é independente e auto-suficiente. Os trabalhos estão divididos de acordo com o modelo de comunicação usado no processamento distribuído de consultas, sendo eles: a estratégia *send-result*, onde servidores encaminham resultados intermediários à servidores remotos; e, a estratégia *get-frag*, onde servidores requisitam dados de outros servidores durante o processamento *distribuído e paralelo* de consultas.

3.1 Estratégia de comunicação *send-result*

Os trabalhos desta seção estão divididos de acordo com a estratégia de processamento de consultas, sendo: *i)* os que adotam uma estratégia de armazenamento e processamento de consultas baseada no estilo relacional; e, *ii)* os que usam técnicas de exploração de grafos baseadas na estrutura de grafo dos dados.

3.1.1 Abordagens relacionais

A seguir, as abordagens relacionais estão divididas em dois grupos, as que se baseiam e as que não se baseiam no modelo de computação *MapReduce* (DEAN; GHEMAWAT, 2008) no processamento de consultas. O *framework MapReduce* é um modelo de processamento distribuído adotado amplamente por sistemas RDF distribuídos baseados no estilo relacional. O seu principal objetivo é fornecer um processamento paralelo sobre um grande número de servidores de um ambiente distribuído.

3.1.1.1 Abordagens baseadas no *MapReduce*

O processamento do *framework MapReduce* é organizado por meio de *jobs*, onde cada *job* processa itens de dados na forma de pares chave-valor e consiste de duas fases, *Map* e *Reduce*, separadas por uma fase de transferência de dados (*shuffle*). A tarefa *Map* lê pares de entrada e gera uma lista de novos pares incluindo todos os diferentes valores recuperados para uma dada chave. Estas listas são enviadas para a tarefa *Reduce* que agrupa estas listas gerando os valores finais do *job*. Esses resultados podem ser usados como entrada em um outro *job*, e assim sucessivamente. Sendo assim, uma consulta SPARQL pode ser mapeada para um ou mais *jobs*, onde os resultados intermediários gerados por um *job* são enviados e armazenados em servidores para que outro *job* seja executado. Um *cluster MapReduce* tem uma arquitetura Mestre-Escravo, onde o Mestre inicializa o processo, escalona tarefas e mantém todas as informações necessárias para o seu gerenciamento. Todos os outros servidores são Escravos e executam as tarefas *Map* e *Reduce*.

SHARD (ROHLOFF; SCHANTZ, 2010) foi o pioneiro na adoção do *framework* por meio do *Apache Hadoop*[1] para a avaliação de consultas SPARQL (KAOUDI; MANOLESCU, 2015). *Hadoop* utiliza para o armazenamento o sistema de arquivos distribuído *HDFS* que é projetado para fornecer escalabilidade, tolerância a falhas e alto desempenho no processamento de sistemas distribuídos. Uma característica importante do *HDFS* é replicação de blocos, onde cada arquivo é dividido em blocos e replicados entre servidores em um valor padrão de 3. Outra característica é o uso do disco tanto para o armazenamento de bases de dados quanto para o armazenamento de dados gerados entre as fases *Map* e *Reduce* (GIMéNEZ-GARCIA; FERNáNDEZ; MARTíNEZ-PRIETO, 2014).

SHARD armazena todos os dados de uma aplicação em um único arquivo. Ele adota fragmentação horizontal, na qual cada linha do arquivo armazena todos os predicados e objetos relacionados a um dado sujeito. Por meio de um particionamento baseado em *hash*, cada servidor fica responsável por um conjunto distinto de triplas. Para o processamento de consultas, SHARD considera um *job MapReduce* para cada padrão de tripla. Para o processamento de junções é usada uma abordagem iterativa onde, para o primeiro padrão de tripla um *job* recupera triplas RDF na fase *Map* e elimina redundâncias, se necessário, na fase *Reduce*. Os *jobs* dos padrões seguintes diferenciam-se do primeiro *job*, uma vez que a fase *Reduce* destes *jobs* realiza junções parciais com resultados de *jobs* anteriores. Estas junções visam reduzir a geração de resultados intermediários inválidos durante a execução de consultas minimizando tanto o custo de processamento quanto o de comunicação. Para finalizar o processamento, um *job* extra remove redundâncias, quando necessário, e projeta o resultado final da consulta. Por fim, SHARD armazena resultados intermediários para usá-los posteriormente no processamento de consultas similares.

Segundo Kaoudi e Manolescu (2015), estratégias de processamento tais como a de SHARD exigem uma sequência de *jobs MapReduce* que potencialmente demandam um longo tempo na avaliação de consultas.

[1] http://hadoop.apache.org

Sendo assim, trabalhos como *HadoopRDF* (HUSAIN et al., 2011) e *CliqueSquare* (GOASDOUÉ et al., 2013) adotam a heurística de redução de *jobs* visando agilizar o processamento de consultas SPARQL. Ambos os trabalhos dividem consultas em subconsultas independentes tomando como base variáveis de junção comuns entre padrões de triplas. A independência entre subconsultas caracteriza-se quando suas variáveis de junção diferem entre si. Consequentemente, subconsultas independentes podem ser tratadas em paralelo em um único *job*. Além disso, Husain et al. (2011) e Goasdoué et al. (2013) propõem métodos de fragmentação de arquivos HDFS a fim de minimizar o custo de E/S existente em SHARD. Quanto ao custo de comunicação, Husain et al. (2011) propõem uma heurística para a ordenação de *jobs* no planejamento de consultas a fim de minimizar o número de resultados intermediários inválidos gerados por eles. Já Goasdoué et al. (2013) propõem um método de alocação de arquivos que é explorado no planejamento de consultas a fim de promover o processamento local de junções nos servidores.

Trabalhos tais como (HUANG; ABADI; REN, 2011), (WU; JIN; YUAN, 2012), (LEE; LIU, 2013) e (YANG et al., 2013) combinam o *framework MapReduce* com repositórios centralizados. O objetivo destes trabalhos é centralizar o processamento de junções em servidores Escravos, minimizando o número de *jobs* e a comunicação durante o processamento de consultas. (HUANG; ABADI; REN, 2011), (LEE; LIU, 2013) e (YANG et al., 2013) adotam o sistema RDF-3X (NEUMANN; WEIKUM, 2010) para o processamento local em cada servidor. RDF-3X armazena dados por meio do modelo de triplas e usa uma estratégia de indexação que combina todos os membros das triplas tornando o acesso aos dados locais eficiente. Além disso, o planejamento de RDF-3X baseia-se em (MOERKOTTE; NEUMANN, 2006) adotando programação dinâmica para enumerar diversos planos de execução e definir a melhor ordem de junção para cada consulta. Já (WU; JIN; YUAN, 2012) baseiam-se no sistema *HadoopDB* (ABOUZEID et al., 2009). *HadoopDB* adota uma arquitetura híbrida que propõe o uso de um sistema gerenciador de banco de dados relacional em cada servidor para armazenar triplas RDF em tabelas e o uso do *MapReduce* para o processamento paralelo de junções.

Quanto ao planejamento de processamento distribuído, (WU; JIN; YUAN, 2012) e (YANG et al., 2013) ordenam planos de consultas considerando o fator de seletividade de subconsultas. O fator de seletividade diz respeito à quantidade de fatos estimados que uma base RDF contém que "casam" com um determinado padrão de tripla ou com um conjunto de padrões de tripla, onde quanto menor o número de fatos relacionados aos padrões, maior é a seletividade e a prioridade deles na ordenação (STOCKER et al., 2008). Sendo assim, o uso deste fator no planejamento viabiliza a redução de resultados intermediários inválidos durante a execução de consultas, diminuindo o custo de processamento e de comunicação em ambientes distribuídos.

Os trabalhos (HUANG; ABADI; REN, 2011), (LEE; LIU, 2013) e (YANG et al., 2013) usam o *MapReduce* somente quando consultas não são totalmente paralelizáveis entre servidores. Huang, Abadi e Ren (2011) particiona grafos RDF por meio do algoritmo METIS (Karypis Lab, 2014) e replica vértices

para sobrepor dados entre fragmentos de dados de acordo com uma garantia de até *n-saltos*. Durante o planejamento de consultas, o processador de consultas verifica se uma dada consulta é completamente paralelizável ou não, de acordo com as suas partições e replicação. Se a consulta é paralelizável significa que cada servidor é capaz de executar a consulta somente com a sua base de dados local e que não é necessário o uso do *MapReduce*. Caso contrário, se a consulta não é totalmente paralelizável, o otimizador decompõe a consulta em subconsultas totalmente paralelizáveis para que seus resultados intermediários sejam processados via operação de junção por meio do *MapReduce*. Neste último caso, o objetivo do processador é criar o menor número de subconsultas possível com o objetivo de realizar o menor número de junções via *MapReduce*.

Em SHAPE, Lee e Liu (2013) adotam a mesma abordagem de processamento de (HUANG; ABADI; REN, 2011), porém eles propõem um método de particionamento alternativo para viabilizar o carregamento de cargas RDF de forma mais eficiente e intensificar o processamento local de junções. O método proposto é uma extensão do particionamento *hash* clássico que considera relações entre sujeitos e objetos para agrupar triplas RDF em partições. Por fim, Yang et al. (2013) adotam uma fragmentação baseada em carga de trabalho favorecendo o processamento local de consultas que estejam de acordo com o padrão da carga de consultas usada na fragmentação. Já para as consultas que não estiverem no padrão da carga de trabalho, a estratégia de junção adotada baseia-se no *MapReduce*.

Wu, Jin e Yuan (2012) propõem uma estratégia de fragmentação guiada pelas relações existentes entre classes de estruturas RDF com o objetivo de minimizar o número de subconsultas envolvidas no processamento de consultas e, consequentemente, o número de *jobs*. Os autores consideram que padrões de triplas relacionados em uma estrutura RDF têm alta probabilidade de serem requisitados em conjunto em consultas SPARQL. No planejamento de consultas, subconsultas são detectadas por meio do método de distribuição de dados, como, por exemplo, junções *sujeito-sujeito* ou *sujeito-objeto* com caminhos de até 2 saltos. Em adição, os autores propõem um mecanismo para a passagem de dados entre servidores durante o processamento de consultas. Desta forma, dada uma sequência de *jobs* para a execução de uma dada consulta, um *job* pode considerar resultados intermediários gerados por *jobs* anteriores a fim de reduzir a geração de resultados intermediários inválidos.

3.1.1.2 Abordagens alternativas ao *MapReduce*

As abordagens desta categoria usam diferentes repositórios e implementam diferentes estratégias de execução de consultas. TriAD-SG *(Triple-Asynchronous-Distributed)* (GURAJADA et al., 2014), por exemplo, propõem a adoção de uma etapa de pré-processamento na execução de consultas a fim de definir previamente as partições de dados dos servidores que participarão da execução de cada consulta. Para isso, dada uma base RDF de uma aplicação, TriAD-SG sumariza a base definindo super-nós por meio do algoritmo METIS. Consequentemente, a alocação de triplas RDF nos servidores preserva as informações

de localidade definidas no grafo de sumarização, e as triplas RDF são indexadas nos servidores por meio de seus super-nós. Os dados em TriAD-SG são armazenados em estruturas de dados em memória principal.

Continuando, Triad-SG explora o grafo de sumarização na fase de planejamento detectando as partições de dados que participarão da execução de cada consulta. Além disso, o planejamento de consultas baseia-se em programação dinâmica e informações estatísticas, a fim de potencializar a minimização de troca de resultados intermediários inválidos entre servidores. Diferentemente dos outros trabalhos, o planejamento de TriAD-SG também tem o objetivo de intensificar o processamento paralelo *intra-servidor* de subconsultas. Apesar de TriAD-SG adotar uma arquitetura Mestre-Escravo, seus Escravos têm autonomia total para executar consultas e trocar mensagens entre si. Logo, após receber o plano de execução do Mestre, cada servidor inicia a execução do plano usando a sua base local e resultados intermediários são trocados entre eles de maneira assíncrona, quando necessário. Este modelo de comunicação asssemelha-se ao modelo de computação BSP (*Bulk Synchronous Parallel*) assíncrono usado em (GONZALEZ et al., 2012). Uma computação BSP síncrona ocorre em uma série de *superpassos*, onde cada um consiste de três componentes: computação local, comunicação e barreira de sincronização (PENG et al., 2016). Sendo assim, cada processador executa alguma computação do primeiro *superpasso* baseado-se em dados armazenados localmente. Em seguida, o segundo *superpasso* é iniciado somente depois que todos os processadores trocam dados entre si e uma barreira de sincronização for estabelecida e, assim sucessivamente No modelo assíncrono, dados podem ser trocados entre processadores assim que forem computados.

Baseando-se em (HUANG; ABADI; REN, 2011), Hose e Schenkel (2013) propõem em *Warp* um método de replicação baseado em carga de trabalho com o objetivo de minimizar o número de subconsultas e, consequentemente, o número de junções distribuídas para o processamento de consultas SPARQL. O propósito do trabalho é atender consultas mais complexas que as consideradas em (HUANG; ABADI; REN, 2011). Sendo assim, durante o seu planejamento, cada consulta é analisada para verificar se ela pode ser executada localmente em cada servidor de maneira independente, ou se o processamento da consulta precisa ser distribuído e coordenado por um servidor dedicado. O armazenamento e o processamento local são feitos por meio do RDF-3X, e o processamento global é feito por meio de algoritmos de junção adequados à ordenação dos resultados intermediários gerados pelos padrões de triplas envolvidos nas consultas.

Partout (GALÁRRAGA; HOSE; SCHENKEL, 2012) considera um método de fragmentação baseado em carga de trabalho e o explora tanto na alocação de dados quanto no planejamento de consultas. Na fragmentação de dados, Partout executa um particionamento horizontal guiado por expressões de filtro e frequências de padrões de triplas presentes em uma dada carga de trabalho. Na alocação de dados, Partout procura agrupar fragmentos relacionados na carga de consultas em um mesmo servidor para que consultas sejam executadas localmente. Por fim, no planejamento de consultas, Partout usa informações sobre a composição dos fragmentos de dados, suas cardinalidades e os servidores que os armazenam. Sendo assim,

o processador de consultas explora informações de alocação para definir exatamente os servidores que participarão do processamento de cada consulta. A estratégia de otimização na etapa de planejamento de Partout é bem variada, pois são consideradas diversas heurísticas pertinentes à abordagem relacional como a prioridade do uso de *merge join* no lugar de *hash join*, a prioridade para operações de seleção e de projeção tomando como base a álgebra relacional e a adoção do paralelismo entre servidores sempre que possível. Na sua arquitetura Mestre-Escravo, Partout adota um sistema baseado em RDF-3X nos servidores Escravos.

Por fim, Harbi et al. (2016) propõem um método dinâmico para a distribuição de dados a fim de acompanhar a carga de trabalho de consultas. Dados são distribuídos por meio de um método baseado em *hash* nos sujeitos das triplas que é usado para dividir consultas em subconsultas na fase de planejamento. Dada uma consulta, cada servidor inicia o seu processamento com sua base local. Caso o processamento de uma subconsulta dependa de dados alocados remotamente, o servidor local recupera os resultados intermediários da subconsulta requisitando a sua execução no servidor remoto adequado. Assim como em Triad-SG, (HARBI et al., 2016) também baseia-se em programação dinâmica e informações estatísticas no planejamento de consultas distribuídas.

3.1.2 Abordagem baseada em grafos

Trinity.RDF (ZENG et al., 2013) adota uma abordagem baseada em grafos no armazenamento de dados e no processamento de consultas. Segundo Zeng et al. (2013), a exploração de grafos que considera a abordagem nativa pode ser implementada mais eficientemente que estratégias relacionais. Entretanto, a exploração de grafos implementada por meio da indexação de tabelas relacionais, *triple stores* ou chave-valor baseado em disco exige operações de junções que comprometem o desempenho da exploração nativa.

Trinity.RDF armazena dados por meio do Trinity (SHAO; WANG; LI, 2013), um sistema distribuído construído sobre um repositório chave-valor em memória. Os dados são armazenados como um conjunto de vértices onde cada par chave-valor mantém o identificador de um vértice RDF e um conjunto de identificadores para os vértices em sua lista de adjacência. A adoção de listas de adjacências possibilita o uso da técnica de exploração de grafos durante a execução de consultas. Para a distribuição de dados, Trinity.RDF adota uma fragmentação horizontal aplicando uma função *hash* nos rótulos dos vértices (sujeitos e objetos). No processamento de consultas, Trinity.RDF combina duas estratégias de junção, sendo: *i)* um algoritmo de exploração de grafos com o propósito de minimizar o número de resultados intermediários inválidos gerados entre junções, e; *ii)* uma estratégia de junção relacional para a computação dos resultados finais de consultas em um servidor central. (GURAJADA et al., 2014) destacam a alta carga de processamento no servidor Mestre para a computação final de consultas em Trinity.RDF, afirmando que a habilidade de avaliar junções em paralelo ainda permanece um fator crucial para a escalabilidade de sistemas RDF. (ZENG et al., 2013) também adotam a programação dinâmica no seu

planejamento de consultas propondo o conceito de *"ponto de exploração"* para ordenar padrões de triplas visando eliminar computação desnecessária entre operações de junções.

Schatzle et al. (2015) propõem a abordagem de processamento de consultas S2X para Hadoop. S2X é implementado sobre GraphX (GONZALEZ et al., 2014), um *framework* para o processamento de grafos implementado com o Apache Spark (ZAHARIA et al., 2012). Spark é um mecanismo de computação paralela e distribuída de propósito geral baseado em memória.

Quanto ao armazenamento de dados, GraphX representa internamente um grafo RDF por meio de duas coleções distintas de arestas e vértices. Quanto a distribuição de dados, GraphX usa uma função *hash* nos rótulos dos vértices (sujeitos e objetos) a fim de viabilizar o particionamento do grafo. Quanto ao processamento de consultas, S2X divide o processamento em três etapas. Na primeira etapa, dada uma consulta, cada vértice do grafo RDF verifica quais variáveis dos padrões de tripla da consulta podem ser instanciadas por ele. Um vértice torna-se candidato à instanciação quando ele possui pelo menos o predicado de um padrão de tripla. Na segunda, após a definição de todos os vértices candidatos, dados são trocados entre os vértices candidatos a fim de validar as suas instanciações. Para finalizar a execução, junções entre as instanciações geram o resultado final da consulta.

Por fim, ainda considerando *send-result*, (HAMMOUD et al., 2015) propõem o *framework* DREAM para o processamento distribuído de consultas, que é independente de estratégia de processamento. Diferentemente de todos os outros trabalhos, bases RDF são replicadas totalmente entre servidores de um ambiente distribuído. Neste caso, consultas SPARQL são divididas e distribuídas entre servidores para que suas partes sejam executadas em paralelo. Caso uma subconsulta dependa de resultados gerados por outras subconsultas, servidores trocam dados entre si a respeito de resultados intermediários gerados.

3.2 Estratégia de comunicação *get-frag*

Os dois trabalhos citados nesta seção baseiam-se na estratégia de execução relacional. O primeiro trabalho adota uma abordagem de processamento distribuída baseada no *framework MapReduce*. O segundo recupera dados distribuídos entre servidores e processa consultas em um servidor central usando uma abordagem específica.

Przyjaciel-Zablocki et al. (2012) adotam o repositório chave-valor HBase que é um sistema de armazenamento orientado a colunas, escalável, distribuído e integrado ao HDFS. O HBase pode ser visto como uma camada adicional de armazenamento que dá suporte ao acesso randômico de dados, segundo (PRZYJACIEL-ZABLOCKI et al., 2012),. No armazenamento de dados, os autores exploram a questão da orientação a colunas do repositório, armazenando os predicados e os objetos de um dado sujeito em uma única linha. Em (PRZYJACIEL-ZABLOCKI et al., 2012) é apresentada uma técnica *MAPSIN*

(Map-Side Index Nested Loop Join), onde dada uma consulta, cada servidor Escravo recupera instâncias do seu primeiro padrão de tripla (`p1`) armazenadas localmente em seu repositório. Em seguida, para computar a junção parcial entre `p1` e o segundo padrão de tripla da consulta (`p2`), uma função *Map* é invocada para cada instância de `p1` a fim de substituir variáveis no segundo padrão de tripla da consulta. Para cada combinação de variável, `p2` é reescrito e pesquisado na base RDF. Desta forma, eliminam-se resultados intermediários inválidos e reduz o número de triplas RDF a serem recuperadas durante o processamento de junções. Desde que não há garantia que as triplas RDF pesquisadas estarão no mesmo servidor, os resultados das requisições trafegam pela rede. A fim de otimizar o processamento de consultas, os padrões de triplas que compartilham uma variável de junção são recuperados e avaliados ao mesmo tempo em uma única fase do *Map*.

Punnoose, Crainiceanu e Rapp (2012) propõem em Rya uma abordagem similar à de (PRZYJACIEL-ZABLOCKI et al., 2012) a fim de eliminar resultados intermediários inválidos entre operações de junções. Para o processamento, um servidor central implementa um algoritmo de laços aninhados indexados que realiza múltiplos *lookups* no repositório *Accumulo*[2] durante a execução de consultas. *Accumulo* também é um repositório chave-valor distribuído orientado a colunas que organiza suas linhas em ordem lexicográfica ascendente de acordo com os valores de suas chaves. Esta ordenação do repositório favorece consultas que consideram intervalos de valores (*ranges*). No planejamento de consultas, Rya ordena padrões de triplas das consultas dando prioridade aos padrões mais seletivos. Rya ainda usa o *RDF-Schema* para explorar o uso do conceito de classes e subclasses para reescrever consultas e inferir fatos implícitos em bases.

3.3 Estratégia de comunicação *send-result* ou *get-frag*

O sistema H_2RDF+ (PAPAILIOU et al., 2014) armazena dados por meio do repositório chave-valor *HBase*. Para o processamento de consultas são propostos dois algoritmos de junção, denominados de *Merge join* e *Sort-Merge join*. Enquanto o primeiro algoritmo processa junções entre dados ordenados, o segundo algoritmo processa junções considerando resultados intermediários desordenados. Por exemplo, dada uma consulta, o primeiro algoritmo é usado para processar junções entre o primeiro primeiro padrão de tripla da consulta (`t1`) e todos os outros padrões da consulta que compartilham uma variável de junção com `t1`. Durante a execução da junção, dados são recuperados de servidores do *cluster*, conforme necessário, para que as junções sejam processadas. Já, o segundo algoritmo processa junções usando resultados intermediários gerados por padrões de triplas já processados e dados recuperados de outros servidores, conforme o primeiro algoritmo. Note que a recuperação de dados intermediários caracteriza a estratégia *send-result* e a recuperação de dados no *cluster*, a estratégia *get-frag*.

Visando minimizar o custo de processamento de consultas, H_2RDF+ propõe uma abordagem híbrida para o processamento de consultas na qual o processador pode optar entre dois tipos de estratégias para o

[2] https://accumulo.apache.org/

	Trabalhos	Estratégia de execução	Modelo de processamento	*MapReduce*	Comunicação
1	(ROHLOFF; SCHANTZ, 2010)	*relacional*	*distribuído*	√	*send-result*
2	(HUSAIN et al., 2011)	*relacional*	*distribuído*	√	*send-result*
3	(GOASDOUÉ et al., 2013)	*relacional*	*distribuído*	√	*send-result*
4	(HUANG; ABADI; REN, 2011)	*relacional*	*distribuído*	√	*send-result*
5	(LEE; LIU, 2013)	*relacional*	*distribuído*	√	*send-result*
6	(YANG et al., 2013)	*relacional*	*distribuído*	√	*send-result*
7	(WU; JIN; YUAN, 2012)	*relacional*	*distribuído*	√	*send-result*
8	(HOSE; SCHENKEL, 2013)	*relacional*	*distribuído*		*send-result*
9	(GALÁRRAGA; HOSE; SCHENKEL, 2012)	*relacional*	*distribuído*		*send-result*
10	(GURAJADA et al., 2014)	*relacional*	*distribuído*		*send-result*
11	(HARBI et al., 2016)	*relacional*	*distribuído*		*send-result*
12	(ZENG et al., 2013)	*grafo*	*centralizado*		*send-result*
13	(SCHATZLE et al., 2015)	*grafo*	*centralizado*	√	*send-result*
14	(HAMMOUD et al., 2015)	-	*distribuído*		*send-result*
15	(PRZYJACIEL-ZABLOCKI et al., 2012)	*relacional*	*distribuído*	√	*get-frag*
16	(PUNNOOSE; CRAINICEANU; RAPP, 2012)	*relacional*	*centralizado*		*get-frag*
17	(PAPAILIOU et al., 2014)	*relacional*	*centralizado* e/ou *distribuído*	√	*send-result* ou *get-frag*
18	(PENTEADO, 2017)	*grafo*	*distribuído*		*get-frag* e *send-result*

Tabela 3.1: Abordagens de Processamento Distribuído de Consultas

processamento dos algoritmos *Merge join* e *Sort-Merge join*, o centralizado ou o distribuído. A estratégia distribuída baseia-se no *framework MapReduce*, onde junções são executadas em paralelo entre diversos servidores. De uma forma geral, a abordagem centralizada favorece a execução de consultas seletivas. Entende-se por consultas seletivas, consultas que possuem pelo menos um padrão de tripla que restringe a quantidade de dados de entrada no processamento da consulta. Já, a abordagem distribuída favorece consultas não-seletivas. Segundo os autores, os custos inerentes aos *jobs* inviabiliza o processamento de consultas seletivas por meio do *MapReduce*.

3.4 Análise

Diante dos trabalhos apresentados neste capítulo, é possível constatar diferentes estratégias de processamento de consultas baseadas na estratégia de comunicação *send-result*. Em contrapartida, constata-se pouca atenção à estratégia *get-frag*. Apesar de Punnoose, Crainiceanu e Rapp (2012) e Papailiou et al. (2014) adotarem a estratégia *get-frag*, somente Przyjaciel-Zablocki et al. (2012) emprega a estratégia em um modelo de execução *paralelo e distribuído*.

A Tabela 3.1 apresenta de maneira sumarizada os trabalhos relacionados, considerando a estratégia de execução de consultas, o modelo de processamento, o uso do *framework MapReduce* e o modelo de comunicação usado entre servidores. Caso os dados sejam recuperados de maneira paralela, porém o processamento da consulta seja feito em um único servidor, o modelo de processamento é classificado como centralizado. Caso contrário, o modelo é definido como distribuído. Todos os trabalhos da tabela apresentam alguma proposta de otimização que influencia o custo de comunicação do processamento distribuído.

O penúltimo trabalho da Tabela 3.1 propõe um algoritmo de junção que adota a estratégia de comunicação *send-result* e a *get-frag* durante o seu processamento. Porém, estas estratégias são usadas de

maneira isolada em cada etapa do algoritmo. Lembrando que o objetivo do trabalho é a redução do uso de *jobs* no processamento de consultas seletivas. Na abordagem de processamento proposta no presente trabalho, uma única consulta pode combinar as duas estratégias de comunicação a fim de minimizar a troca de dados entre servidores durante o seu processamento. Durante a transição entre fragmentos de dados, cada servidor escolhe a melhor estratégia de comunicação com outro servidor remoto conforme o seu conhecimento dos dados envolvidos na consulta. Estudos experimentais no Capítulo 6 mostram as vantagens do uso de métodos de distribuição *controlada* de dados no processamento de consultas. O custo de comunicação pode ser reduzido com a combinação das duas estratégias de comunicação (*send-result* e *get-frag*) em um único método de execução paralelo e distribuído, quando comparado com o uso de um única estratégia de comunicação.

A abordagem de processamento apresentada no próximo capítulo adota um algoritmo de exploração de grafos baseado em Trinity.RDF (ZENG et al., 2013). Além disso, a abordagem também adota um modelo de comunicação assíncrono entre servidores, assim como em Triad-SG (GURAJADA et al., 2014). Por fim, a categorização de fragmentos de dados por meio de padrões viabiliza automaticamente uma forma de indexação de dados reduzindo o espaço inicial de pesquisa no processamento de consultas.

CAPÍTULO 4

UMA ABORDAGEM PARA O PROCESSAMENTO DISTRIBUÍDO DE CONSULTAS SPARQL

Este capítulo apresenta a abordagem para o processamento de consultas SPARQL em bases RDF distribuídas. Bases RDF são armazenadas usando o modelo nativo de grafos e a execução de consultas baseia-se em um algoritmo de exploração distribuída de grafos. Para a comunicação no *cluster*, a abordagem de processamento propõe um método de comunicação híbrido denominado de *2ways*. Neste modelo Escravos podem escolher entre duas estratégias de comunicação, *get-frag* e *send-result*, na troca de dados no *cluster*. A escolha é feita entre cada par Escravo-Escravo e depende diretamente dos dados envolvidos no processamento de cada consulta.

4.1 Arquitetura

A abordagem de processamento de consultas proposta baseia-se em uma arquitetura *Mestre-Escravo shared-nothing*. A arquitetura adota o paralelismo entre servidores, onde cada Escravo executa requisições em paralelo com outros Escravos sem a exigência de sincronismo entre os mesmos.

Para o processamento de consultas, conforme mostra a arquitetura da Figura 4.1, o servidor Mestre recebe e analisa requisições de consultas, gerando planos de execução e requisitando que todos os Escravos do *cluster* os executem. Logo, todos os Escravos iniciam paralelamente o processamento do plano com os seus dados locais. Caso consultas envolvam dados armazenados em servidores remotos, os Escravos devem trocar dados entre si. Sendo assim, considerando os fluxos numerados da Figura 4.1, dada uma base RDF G_D, o servidor Mestre a fragmenta de acordo com padrões pré-definidos (1) e distribui os fragmentos gerados entre os Escravos do *cluster* (2). Dada uma requisição de consulta (3), o Mestre elabora o seu plano de execução de acordo com os mesmos padrões pré-definidos (4) e dispara a execução da consulta (5) enviando o plano da consulta para todos os Escravos (6). A partir deste ponto, cada Escravo inicia a execução do plano usando o seu repositório local (7) e trocando mensagens no *cluster* conforme necessário (6) escolhendo uma estratégia de comunicação (8 e 9). Ao final, o Mestre recebe os resultados gerados pelos Escravos (6) retornando o resultado final da consulta ao seu requisitor (10).

O *módulo de alocação* da Figura 4.1 representa a funcionalidade de *armazenamento de dados* na arquitetura. Já os *módulos de planejamento*, *de execução* e *de estimativa de custo* representam a funcionalidade de *processamento de consultas*. Os *repositórios* (representados pelas figuras em cinza) armazenam dados, índices e metadados dando suporte às funcionalidades. As funcionalidades são descritas a seguir.

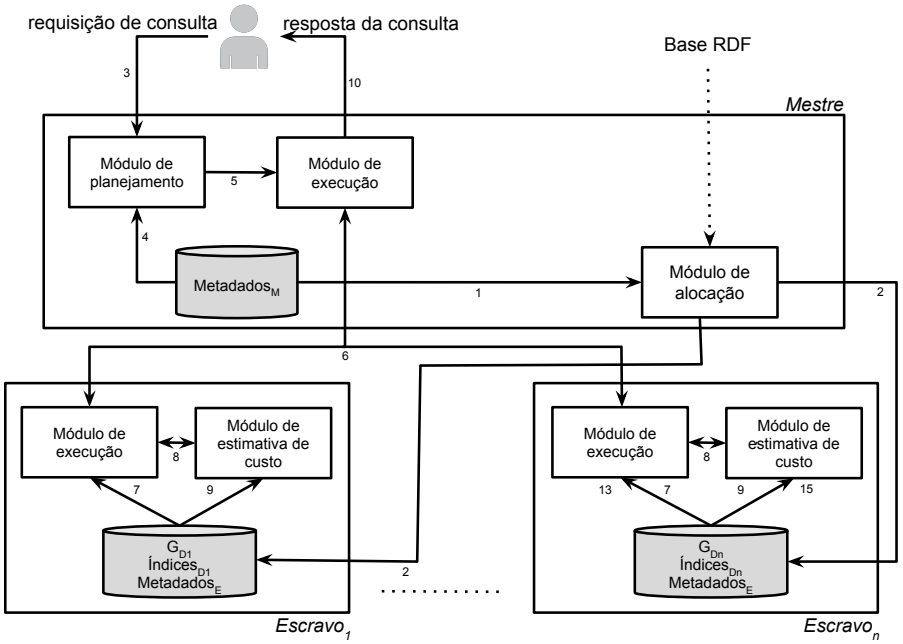


Figura 4.1: Arquitetura Mestre-Escravo da abordagem de processamento de consultas

4.2 Armazenamento de dados

O *módulo de alocação* é responsável por fragmentar e distribuir uma base G_D entre os Escravos de um *cluster*. Cada Escravo$_j$ agrupa e armazena em seu repositório os fragmentos de dados (G_{Dj}) recebidos do Mestre, conforme mostra a Figura 4.1. A distribuição de dados de uma base G_D é feita de maneira controlada tomando como base partições pré-definidas em sua estrutura RDF, conforme citado na Seção 2.4.1. Uma estrutura RDF é representada por um grafo de sumarização $G_S = (V_S, E_S)$ extraído de $G_D = (V_D, E_D)$. Considerando a base G_D da Figura 1.1, a estrutura RDF da Figura 1.2(b) representa o mapeamento *total* dos vértices de G_D para os vértices de V_S. Por exemplo, *Person1* $\mapsto$ $v9$, onde definimos que *type(Person1)* = $v9$. A fim de dar suporte ao módulo de execução, cada vértice v_D de V_D deve adicionar o valor de *type(v_D)* em sua definição.

Uma partição $\mathcal{P}$ do grafo de sumarização G_S define um conjunto disjunto de vértices (conectados), denominados de *padrões de alocação* (PAs). Estes PAs são usados para particionar uma base G_D em fragmentos de dados que seguem a composição dos mesmos. Por exemplo, a Figura 1.2(b) apresenta cinco PAs, onde o *PaProduct* é composto pelos vértices v_3, v_4 e v_5. A Figura 1.1 apresenta três fragmentos que seguem a estrutura *PaProduct*: *PaProduct_1* e *PaProduct_3*, armazenados no servidor X e *PaProduct_2* armazenado nno servidor Z. PAs definem unidades de comunicação e co-alocação no armazenamento. Ou seja, vértices em G_D que pertencem ao mesmo fragmento são alocados em um mesmo servidor e qualquer transmissão de dados entre os servidores envolve no mínimo um fragmento.

A função Pa mapeia vértices v_s de G_S para a partição em $\mathcal{P}$ que contém v_S, logo *Pa(v9)* = *PaPerson*. A Figura 1.2(a) exibe cinco PAs. Por exemplo, *PaPerson* = {$v9$, $v10$}. Para cada vértice v_S

de V_S é definida uma tupla *(in,out)*, onde: *in(v_S)* é o conjunto de arestas incidentes em v_S, e *out(v_S)* é o conjunto de arestas que partem de v_S. Observe que as arestas em G_S podem conectar vértices do mesmo PA ou de PAs distintos. O primeiro tipo de aresta é classificada como *propriedade intra-PA (I)* e o segundo como *propriedade inter-PA (E)*. Uma propriedade *intra-PA* (v_{S1}, v_{S2}, e_S) garante que pares de vértices de V_D mapeados para v_{S1} e v_{S2} e conectados por e_S estarão alocados em um mesmo fragmento. Continuando com o exemplo da Figura 1.2(b), $PaPerson = \{v9,\ v10\}$, onde $v9 = (in : \{(parentOf, v9, I), (friendOf, v9, E)\}, out : \{(name, v10, I), (parentOf, v9, I), (friendOf, v9, E)\})$ e $v10 = (in : \{(name, v10, I)\}, out : \{\})$. Intuitivamente este PA define que pessoas de uma mesma família *devem* ser agrupadas em um mesmo fragmento, enquanto que amigos *podem* ser armazenados em fragmentos distintos, conforme mostra a Figura 1.1.

Os exemplos e os experimentos deste trabalho definem PAs por meio do padrão estrutural do tipo estrela assegurando que triplas que compartilham o mesmo sujeito fiquem alocadas em um mesmo servidor, conforme (MINH-DUC et al., 2015) citado na Seção 2.4.1. Além disso, a fim de considerar a alocação de dois ou mais recursos em fragmentos de dados nos exemplos usados neste trabalho, admitiram-se relacionamentos recursivos *intra-PA* em G_S. Entretanto, é essencial destacar que a abordagem de processamento proposta pode ser aplicada para qualquer tipo de modelo de distribuição que fragmenta e categoriza dados por meio de padrões pré-definidos.

4.2.1 *Índices e metadados*

Mecanismos de indexação são utilizados para a implementação das funcionalidades da arquitetura. Cada Escravo$_j$ armazena localmente em seu repositório índices referentes à sua base local (Índices$_{Dj}$). A Tabela 4.1 exibe os índices usados pelas funcionalidades da arquitetura da Figura 4.1.

O índice *IxRemoteServer* viabiliza a exploração distribuída entre os vértices alocados no *cluster*. Dada uma propriedade *inter-PA* que conecta dois vértices x e y, alocados no $Escravo_L$ e no $Escravo_R$, respectivamente, o índice *IxRemoteServer* permite que o $Escravo_L$ obtenha o endereço físico do $Escravo_R$ viabilizando a exploração distribuída de grafos. O índice *IxPaFrags* associa cada padrão de alocação (PA) a todos os fragmentos deste tipo em um $Escravo_L$. Já, o índice *IxFragNodes* associa cada fragmento aos vértices nele contidos. Ou seja, por meio de *IxFragNodes*, todos os identificadores dos vértices de um fragmento do $Escravo_L$ podem ser recuperados, dado o identificador de um fragmento. Por fim, *IxRemoteFrag* viabiliza a identificação de um fragmento remoto dado o identificador de um determinado vértice remoto.

Além dos índices, metadados também são gerados pelo *módulo de alocação*. Cada Escravo$_j$ armazena localmente os metadados necessários para o seu funcionamento (Metadados$_E$). Metadados sobre o tamanho médio dos fragmentos de dados de G_D gerados pelo *módulo de alocação* são alocados nos Escravos dando suporte ao *módulo de estimativa de custo* da abordagem de processamento de consultas (fluxo 9

	Chave de pesquisa	Valor associado
IxRemoteServer	Identificador do vértice remoto	IP do servidor remoto
IxPaFrags	Padrão de alocação	{Identificadores dos fragmentos}
IxFragNodes	Identificador do fragmento	{Identificadores dos vértices}
IxRemoteFrag	Identificador do vértice remoto	Identificador do fragmento remoto

Tabela 4.1: Índices usados pelas funcionalidades da arquitetura

da arquitetura), conforme apresentado na Seção 4.3.2. O Mestre armazena em seu repositório o grafo de sumarização de G_D dando suporte ao *módulo de alocação* e ao *módulo de planejamento* (Metadados$_M$).

No Capítulo 5 a arquitetura volta a ser discutida, dado que o capítulo propõe a funcionalidade de *atualização de dados* na arquitetura atual. Consequentemente, a nova funcionalidade também influencia os índices e os metadados discutidos aqui.

4.3 Processamento de consultas

A abordagem de processamento distribuído de consultas está dividida em duas partes, o planejamento e a execução de consultas. A primeira parte refere-se ao *módulo de planejamento* e, a segunda, ao *módulo de execução* e ao *módulo de estimativa de custo* da arquitetura da Figura 4.1.

4.3.1 Planejamento

Um plano de consulta deve determinar em qual ordem os padrões de tripla de uma consulta SPARQL serão explorados na base G_D distribuída, dado que o modelo nativo prevalece no armazenamento de dados e o modelo de execução baseia-se na exploração de grafos.

Formalmente, dado um grafo de consulta $G_Q = (V_Q, E_Q, F, P)$, uma consulta pode ser representada por meio de um conjunto de sequências *lineares* de exploração. Cada sequência S_I do conjunto é composta por passos, $S_I = [s_1, \ldots, s_n]$, onde cada passo s_i representa um padrão de tripla de G_Q a ser explorado na aplicação. A Figura 4.2(b) mostra uma possível sequência *linear* da consulta da Figura 4.2(a) onde cada passo s_i é representado como uma tupla (a, dir, f), onde: (**1**) a é um padrão de tripla $(s, p, o) \in E_Q$; (**2**) $dir \in \{in, out\}$, se $dir = out$ a exploração é do sujeito s para o objeto o, e *source(s_i)* é definido como s, e *target(s_i)* como o. Caso contrário, se $dir = in$ a exploração é do objeto o para o sujeito s, e *source(s_i)* é definido como o, e *target(s_i)* como s; e, (**3**) f é um conjunto de filtros definidos para *target(s_i)*. Uma sequência é dada como *linear* se para todo passo s_j, $2 <= j <= |S_I|$, se *source(s_j)* $= v$ existe um passo s_i, $i < j$ tal que $v =$ *source(s_i)* ou $v =$ *target(s_i)*. O *ponto inicial de exploração* de uma sequência é representado pela *origem* do seu primeiro passo.

Os padrões de alocação do grafo de sumarização G_S são usados aqui para reduzir o espaço de pesquisa dos *pontos iniciais de exploração* no grafo RDF (G_D) durante o processamento de consultas. Logo, combinando o grafo de consulta G_Q com G_S de uma dada aplicação, é possível determinar quais PAs são

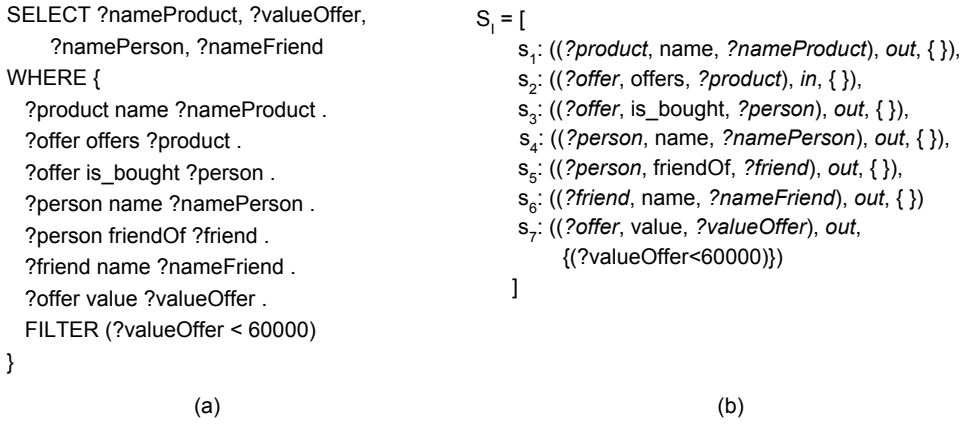


Figura 4.2: Consulta SPARQL (a); Sequência S_I da consulta (b)

requeridos para o processamento da consulta e então reduzir o espaço inicial de pesquisa para somente os fragmentos de dados destes padrões. Sendo assim, no início do planejamento, torna-se necessário o mapeamento de G_Q para G_S. O mapeamento $G_Q \mapsto G_S$ é feito por meio de um algoritmo de exploração de grafos que encontra subgrafos em G_S que são homomórficos à G_Q. O Algoritmo 2 representa este processo de mapeamento onde a sua entrada é S_I e G_S, e o seu o resultado é um conjunto de mapeamentos M_{QS}. Embora o casamento de grafos tenha uma complexidade de tempo exponencial, na prática, grafos de consultas são usualmente pequenos e grafos de sumarização são menores do que o grafo RDF da base em questão. A Figura 4.3(a) mostra o mapeamento gerado para a sequência da Figura 4.2(b).

O Algoritmo 2 inicia mapeando o *ponto inicial de exploração* de uma sequência S_I para todos os possíveis PAs de G_S (linhas 3–16). A *origem* de s_1 define o *ponto inicial de exploração* do mapeamento. O mapeamento ocorre quando um vértice v_S do grafo G_S possui uma propriedade que representa o *predicado* do padrão de tripla de s_j de acordo com a sua direção $s_j.dir$. Considerando como entrada a sequência S_I da Figura 4.2(b) e o grafo G_S da Figura 4.3(b), quatro mapeamentos são gerados: $m_{QS_0} = \{?product \mapsto v1\}$, $m_{QS_1} = \{?product \mapsto v3\}$, $m_{QS_2} = \{?product \mapsto v9\}$, $m_{QS_3} = \{?product \mapsto v11\}$; desde que $v1, v3, v9$ e $v11$ possuem a propriedade *name* em suas listas de adjacência *out*.

Continuando, a partir de cada mapeamento gerado, cada passo s_i da sequência adiciona uma nova variável de mapeamento caminhando em G_S de acordo com o seu padrão de tripla (s, p, o) e a direção *dir* (linhas 19–29). Prosseguindo com o exemplo, os quatro mapeamentos são incrementados com o mapeamento de *?name* para G_S. Porém, com a exploração do segundo passo de S_I somente m_{QS_1} continua no processo de mapeamento, dado que os outros vértices mapeados ($v1, v9$ e $v11$) não possuem a propriedade *offers* em suas listas de adjacência *in*. Por fim, a Figura 4.3(a) representa o mapeamento final de m_{QS_1}.

Considerando o grafo de consulta G_Q da Figura 4.2(a), as arestas tracejadas na Figura 4.3(b) destacam o subgrafo homomórfico à G_Q em G_S encontrado pelo Algoritmo 2. Observe que, em geral, podem existir

Algoritmo 2: Mapping G_Q to G_S

Entrada: S_I; G_S
Saída: M_{QS}

```
1  M:= {};
2  for j = 1 ... |S_I| do
3      if j=1 then
4          for each PA of the G_S do
5              for each vertex v_S of PA do
6                  if dir = in then
7                      if p is an incoming edge in v_S.in then
8                          insert {(source(s_j) ↦ v_S)} in M;
9                      end
10                 else
11                     if p is an outgoing edge in v_S.out then
12                         insert {(source(s_j) ↦ v_S) } in M;
13                     end
14                 end
15             end
16         end
17     end
18     newM:= {};
19     for each m in M do
20         Let v be a node such that source(s_j) ↦ v is in m;
21         if dir = in then
22             targetSet = {(v_new) | (p,v_new) is an incoming edge in v.in};
23         else
24             targetSet = {(v_new) | (p,v_new) is an outgoing edge in v.out};
25         end
26         for all v_new in targetSet do
27             newM:= newM ∪ {m ∪ {target(s_j) ↦ (v_new)}};
28         end
29     end
30     M_QS = M_QS ∪ newM;
31 end
32 return M_QS;
```

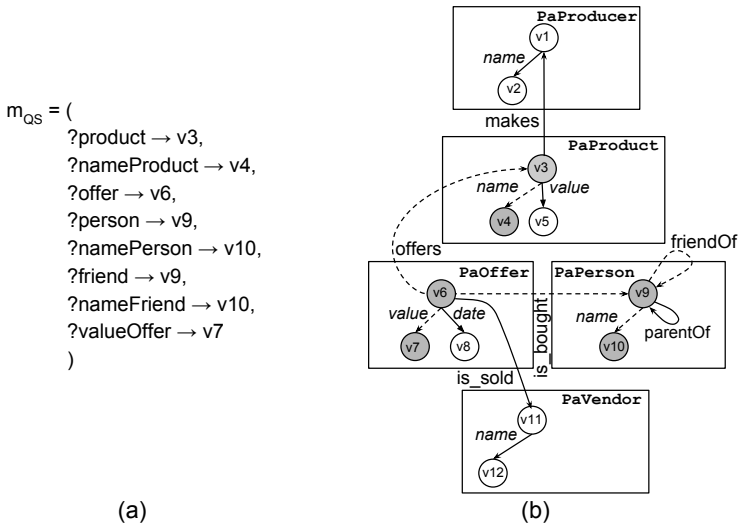


Figura 4.3: Mapeamento $G_Q \mapsto G_S$ de S_I (a); Subgrafo homomórfico à G_Q em destaque no G_S da Figura 1.1(b)

múltiplos mapeamentos resultantes. Por exemplo, a consulta *(SELECT ?nameAll WHERE {?x name ?nameAll.})* possui três subgrafos homomórficos em G_S da Figura 4.3(b).

Para cada mapeamento $m_{QS} \in M_{QS}$ é gerado um plano de execução. Sendo assim, o processamento da consulta é definido como a união dos resultados gerados por este *conjunto* de planos de execução. Nossa abordagem considera a vantagem do armazenamento de distribuição controlada a fim de minimizar a comunicação entre servidores Escravos. Sendo assim, ocorrências de PAs são usadas para gerar um

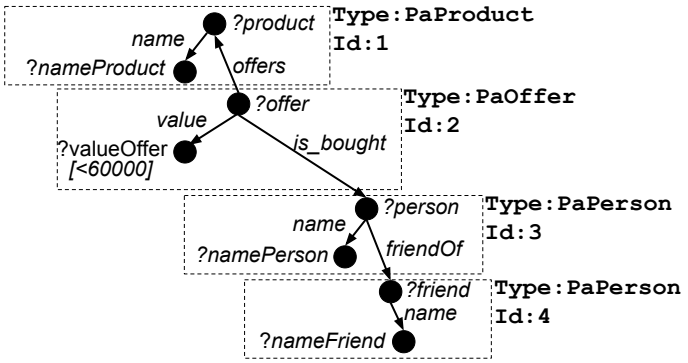


Figura 4.4: Ocorrências de PAs da consulta da Figura 4.2(a)

plano de consulta no qual os vértices de um mesmo PA são visitados em bloco, antes do processamento considerar outro padrão. Ocorrências de PAs são definidas como subgrafos conectados (V_{PA}, E_{PA}) de V_Q tal que todos os vértices em V_{PA} são mapeados para vértices em um mesmo PA de G_S. Além disso, durante o processamento de consultas, cada ocorrência de PA deve ser usada para explorar somente um fragmento de dados de G_D. Como um exemplo, a Figura 4.4 mostra a existência de quatro ocorrências de PAs na consulta da Figura 4.2(a). Cada ocorrência de PA *oc* tem um *tipo*, consistindo do PA para o qual todos os vértices de *oc* são mapeados, e *oc* é associado à um único identificador $id(oc)$.

Para a definição de ocorrências de PAs, primeiramente, os passos da sequência S_I da consulta são agrupados em um ou mais grupos g_i. Considerando que v é o vértice para o qual $source(s_i)$ foi mapeado em G_S, $Pa(v)$ define o grupo de cada passo de S_I, conforme mostra a Figura 4.5(a). Por exemplo, note que, a tripla de s_1 está representada no grupo g_1 e de s_4 em g_2. Isso acontece porque a origem de cada passo foi mapeada para PAs diferentes em G_S. Enquanto, a origem de s_1 (*?product*) foi mapeada para $v3$ em G_S e $Pa(v3) = PaProduct$, a origem de s_4 (*?offer*) foi mapeada para $v6$ e $Pa(v6) = PaOffer$.

Continuando com a detecção de ocorrências, cada grupo g_i é analisado a fim de detectar duas ou mais ocorrências de PAs no grupo. Isso é necessário dado que dois ou mais fragmentos de dados com um mesmo tipo de PA podem ser usados no processamento de um único resultado de uma consulta. Sendo assim, primeiramente, a análise procura detectar subgrafos desconexos em cada grupo g_i. Caso existam subgrafos desconexos em g_i, o grupo é dividido gerando um novo grupo para cada subgrafo desconexo. Na sequência, o processo de análise procura em cada g_i pares de vértices conectados por meio de propriedades *inter-PA*, dado que propriedades *inter-PA* em G_S relacionam diferentes fragmentos de dados em G_D. Logo, para cada g_i, cada par de vértices conectados por meio de uma propriedade *inter-PA* divide o grupo em dois novos grupos, onde cada novo grupo possui um vértice do par. Por exemplo, a aresta *friendOf* representa uma propriedade *inter-PA* que conecta os vértices *?person* e *?friend* do grupo g_3 da Figura 4.5(a). Sendo assim, g_3 gera os grupos g_4 e g_5 na Figura 4.5(b). Esta divisão é feita sucessivamente até que todos os pares de vértices conectados por meio de uma propriedade *inter-PA* do grupo sejam considerados. Por fim, cada grupo g_i gerado na análise representa uma ocorrência de PA *oc*

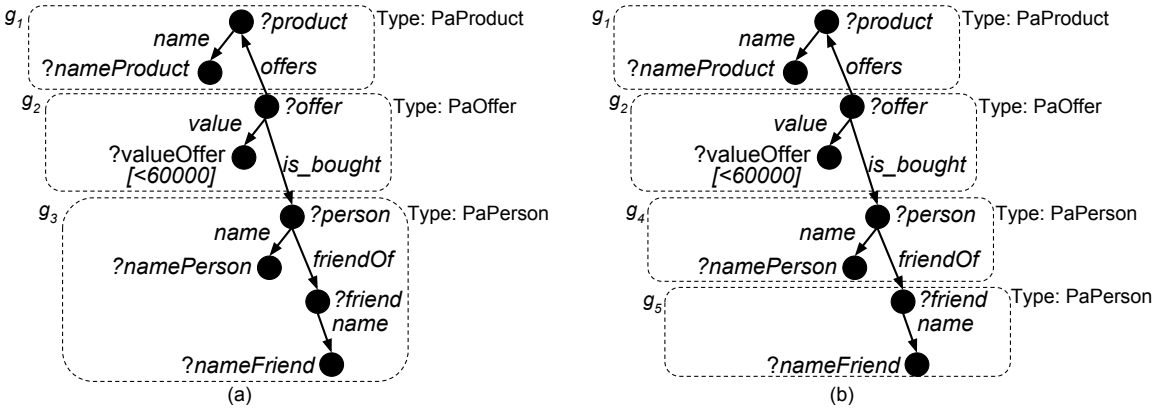


Figura 4.5: Resultados gerados durante a detecção de ocorrências de PAs da sequência S_I (a) e (b)

S = [
s_1: ((*?product*, name, *?nameProduct*), *out*, *PaProduct*, 1, { }),
s_2: ((*?offer*, offers, *?product*), *in*, *PaProduct*, 1, { }),
s_3: ((*?offer*, value, *?valueOffer*), *out*, *PaOffer*, 2,
{(?valueOffer<60000)}),
s_4: ((*?offer*, is_bought, *?person*), *out*, *PaOffer*, 2, { }),
s_5: ((*?person*, name, *?namePerson*), *out*, *PaPerson*, 3, { }),
s_6: ((*?person*, friendOf, *?friend*), *out*, *PaPerson*, 3, { }),
s_7: ((*?friend*, name, *?nameFriend*), *out*, *PaPerson*, 4, { })
]

Figura 4.6: Sequência S da consulta da Figura 4.2(a)

que recebe um identificador único, o $id(oc)$. A Figura 4.5(b) mostra as ocorrências de PAs detectadas na sequência S_I da consulta da Figura 4.2(b). Note que há duas ocorrências de PAs do tipo *PaPerson* com identificadores diferentes, dado que diferentes fragmentos de dados do tipo *PaPerson* são usados para compor um único resultado da consulta.

Por fim, formalmente, dada a sequência S_I de uma consulta e as ocorrências de PAs detectadas, o resultado do gerador de planos de consultas para um mapeamento m_{QS} é uma nova sequência de exploração *linear* $S = [s_1, \ldots, s_n]$ que incrementa S_I representada por meio da tupla (a, dir, f, pat, id), onde os três primeiros elementos são os mesmos de S_I e:

- pat é o tipo de ocorrência de PA que contém a;
- id é o identificador único da ocorrência de PA que contém a.

Uma possível sequência de exploração S para o grafo da Figura 4.2(b) consiste dos passos mostrados na Figura 4.6, onde *?product* é o *ponto inicial de exploração* de S. Note que S não segue necessariamente a mesma ordem que S_I dado que as ocorrências de PA usadas no planejamento podem exigir uma reorganização nos padrões de triplas de S_I conforme descrito a seguir.

Considera-se uma sequência S de exploração *válida* em relação ao grafo da consulta $G_Q = (V_Q, E_Q, F, P)$, a estrutura do grafo G_S, ao mapeamento m_{QS} e uma sequência S_I se ela satisfaz as seguintes condições:

- (**C1**) ela processa todos os padrões de triplas de G_Q, ou seja, $\bigcup_{s \in S}\{s.a\} = E_q$ e $|S| = |E_q|$;
- (**C2**) para todo passo s_j, $2 <= j <= |S|$, se *source(*s_j*)* $= v$ então existe um passo s_i, $i < j$ tal que $v =$*source(*s_i*)* ou $v =$*target(*s_i*)*;
- (**C3**) para algum par de passos s_i, s_j, $i < j$, se $id(s_i)$=$id(s_j)$ então para todo passo s_k, $i < k < j$, $id(s_k)$=$id(s_i)$;
- (**C4**) para todo passo s_i, se $source(s_i)$=v e v é um vértice na ocorrência de PA oc então $id(s_i) = id(oc)$ e $pat(s_i)$ é igual ao tipo de PA de oc.

A condição C1 define que todos os passos de S devem ser executados. A condição C2 define que começando a exploração a partir de *pontos iniciais*, toda exploração de um passo está relacionada a algum vértice obtido em passos anteriores. A condição C3 define que a exploração dentro de cada padrão deve ser feita em bloco. A condição C4 define que todas as ocorrências de PAs devem conter o número correto de passos.

A ordem na qual os PAs são explorados impacta no desempenho do processamento da consulta. Entretanto, a ordenação de exploração de PAs está fora do escopo deste trabalho. Os resultados do Capítulo 6 mostram que *2ways* é efetivo independente da ordenação de PAs nas consultas consideradas nos experimentos.

4.3.2 Execução

Planos de execução de consultas, compostos de um conjunto de sequências de exploração $\mathcal{S} = \{S_1, \ldots S_m\}$ são gerados pelo servidor Mestre e enviados para todos os servidores Escravos do *cluster* para começarem a execução em paralelo. Conforme os Escravos finalizam a computação de todos os passos de uma sequência, os seus resultados são enviados ao Mestre. Lembrando que cada Escravo no *cluster* possui índices e metadados que dão suporte à execução de consultas.

O Algoritmo 3 mostra o processo usado para a execução de todos os planos de consulta em G_D, onde cada sequência $S_i \in \mathcal{S}$ é processada pelo servidor $Escravo_L$ como descrito a seguir. A descrição do algoritmo baseia-se na base G_D da Figura 1.1 por meio da consulta da Figura 4.2(a).

Considere $S_i = [s_1, \ldots, s_n]$ e G_{D_L} como o subconjunto da base RDF armazenado no $Escravo_L$. Primeiro, fragmentos de $s_1.pat$ são recuperados para a obtenção dos vértices do grafo RDF que correspondem ao ponto inicial de exploração *source(*s_1*)* (linhas 2-8). Logo, os pontos iniciais de exploração são recuperados por meio dos índices *IxPaFrags* e *IxFragNodes*. O primeiro recupera os fragmentos de um determinado tipo de PA, e o segundo, os vértices dos fragmentos. Continuando, os pontos iniciais de exploração são computados para uma sequência S_i em um servidor $Escravo_L$ como $s_{0L} = \{\{source(s_1) \mapsto v_D\} \mid type(v_D) = v_S, m_{QS}(source(s_1)) = v_S \text{ and } v_D \in G_{D_L}\}$. Note que neste

Algoritmo 3: PlanExecution

```
   Entrada: [m_QS, S, step j, M]
   Saída: [a set of mappings M]
 1 Let s_j be ((s, p, o), dir, f, pat, id);
 2 if j = 1 then
 3 |   for each vertex v in a fragment f of type pat from G_D do
 4 |   |   if type(v) = m_QS(source(s_1)) then
 5 |   |   |   insert {(source(s_1) ↦ v)} in M;
 6 |   |   end
 7 |   end
 8 end
 9 newM:= {};
10 for each m in M do
11 |   Let v be a node such that source(s_j) ↦ v is in m;
12 |   if dir = in then
13 |   |   targetSet = {(v_new) | (p,v_new) is an incoming edge in v.in};
14 |   else
15 |   |   targetSet = {(v_new) | (p,v_new) is an outgoing edge in v.out};
16 |   end
17 |   for all v_new in targetSet do
18 |   |   if v_new satisfies filters f then
19 |   |   |   newM:= newM ∪ {m ∪ {target(s_j) ↦ (v_new)}};
20 |   |   end
21 |   end
22 end
23 if j < |S| then
24 |   if s_{j+1}.id = s_j.id then
25 |   |   for each mapping m in newM do
26 |   |   |   execute PlanExecution(m_QS,S, j + 1, newM) on L;
27 |   |   end
28 |   else
29 |   |   Initialize partial[R] and frag[R] with empty sets for every server Slave_R;
30 |   |   for each mapping m in newM do
31 |   |   |   Let joinNode be a node such that source_{j+1} ↦ joinNode is in m;
32 |   |   |   Let R be the server that stores joinNode;
33 |   |   |   partial[R]:= partial[R] ∪ {m};
34 |   |   |   frag[R]:= frag[R] ∪ {joinNode};
35 |   |   end
36 |   |   for each server R with frag[R] ≠ { } do
37 |   |   |   if R=L then
38 |   |   |   |   execute PlanExecution(m_QS,S,j+1,partial[L]) on server Slave_L;
39 |   |   |   else
40 |   |   |   |   if Cost_SR[R] < Cost_GF[R] then
41 |   |   |   |   |   execute PlanExecution(m_QS,S,j+1,partial[R]) on server Slave_R;
42 |   |   |   |   else
43 |   |   |   |   |   request frag[R] from server R;
44 |   |   |   |   |   execute PlanExecution(m_QS,S,j+1,partial[R]) on server Slave_L;
45 |   |   |   |   end
46 |   |   |   end
47 |   |   end
48 |   end
49 else
50 |   return newM to master server; /* execution plan completed*/;
51 end
```

ponto torna-se essencial que cada vértice de G_D armazene em sua lista de adjacência o vértice de G_S para o qual ele é mapeado. Na consulta exemplo temos $s_{0Z} = \{\{?product \mapsto Product2\}\}$, desde que $typeProduct2 = v_3$, $m_{QS}(?product) = v_3$ e *Product2* é armazenado no servidor $Escravo_Z$. Consequentemente, $s_{0X} = \{\{?product \mapsto Product1\}, \{?product \mapsto Product3\}\}$. Para os outros Escravos, s_{0Y}=s_{0W}=s_{0V}=$\{\}$.

Iniciando a exploração dos vértices recuperados a partir de *source(s_1)*, cada passo s_i da sequência de exploração adiciona uma nova variável de mapeamento caminhando no grafo RDF de acordo com o padrão de tripla (s, p, o) e a direção *dir*. Se o padrão de tripla não é mapeado para G_D ou se o novo mapeamento não satisfaz os filtros, o mapeamento correspondente é removido do conjunto resultante (linhas 10–22). Filtros de s_i consideram valores constantes e instanciações de padrões de triplas anteriores em *target*. No exemplo, depois de processar o passo s_1, $s_{1Z} = \{\{?product \mapsto Product2, ?nameProduct \mapsto v_{\text{“}Fusion''}\}\}$

e $s_{1X} = \{\{?product \mapsto Product1, ?nameProduct \mapsto v_{\text{“}Linea''}\}, \{?product \mapsto Product3, ?nameProduct \mapsto v_{\text{“}Mob''}\}\}$. Observe que s_2 pode ser processado no mesmo servidor X e Z como o passo s_1 porque $s_1.id = s_2.id$ (linha 24), devido ao fato de que o vértice *source* de ambos os passos estarem no mesmo padrão de alocação e existir a *garantia* de que eles estão armazenados no mesmo servidor.

Continuando com o nosso exemplo, o resultado do processamento do passo s_2 é: $s_{2X} = \{s_{1X} \cup \{?offer \mapsto Offer1\}, s_{1X} \cup \{?offer \mapsto Offer2\}, s_{1X} \cup \{?offer \mapsto Offer3\}, s_{1X} \cup \{?offer \mapsto Offer4\}, s_{1X} \cup \{?offer \mapsto Offer5\}\}$. Note que os caminhos de exploração iniciados por *Product2* e *Product3* são descartados dado que os vértices não possuem o predicado *offers* em suas listas de adjacências *in*. Continuando, o processamento não pode continuar no mesmo servidor para o passo s_3 uma vez que $s_3.id \neq s_2.id$ (linha 28). Observe que no exemplo a variável *?offer* é mapeada para um vértice armazenado em um servidor remoto via a propriedade inter-PA *offers*. Como resultado, para continuar a consulta, ou o servidor X requisita que os servidores remotos transmitam os fragmentos de dados nos quais as suas ofertas estão armazenadas e continua o processamento da consulta em X, ou os mapeamentos gerados são enviados para o servidores remotos, e eles continuam a execução do plano (linhas 29–47). A primeira estratégia de comunicação é classificada como *get-frag*, e a segunda como *send-result*.

Estimativa de custo

A escolha da estratégia de comunicação é feita durante a execução da consulta, baseada no número de requisições e no volume de dados a ser transmitido entre cada par Escravo-Escravo. Considere s_i como o passo no qual a exploração entre diferentes padrões acontece, ou seja, $s_i.id \neq s_{i+1}.id$, $i >= 1$. Considere $varJoin$ como a variável *source* de s_{i+1} e $[\![M_{join}]\!]|_{Escravo_R}$ como o subconjunto de mapeamentos de $varJoin$ ($[\![M_{join}]\!]$) para vértices no servidor $Escravo_R$. Logo, $[\![M_{join}]\!]|_{Escravo_R} = \{m \in [\![M_{join}]\!]\} \mid varJoin \mapsto joinNode \ \ está\ em\ m$, e $joinNode \in G_{D\,Escravo_R}\}$. Intuitivamente, estes mapeamentos são os *resultados intermediários* para serem enviados para cada servidor R pela estratégia *send-result*. Lembrando, que o índice *IxRemoteServer* viabiliza o conhecimento da localização de um vértice remoto no *cluster*. Continuando, o número de mensagens a ser transmitido ao servidor remoto R (NM_R) é o tamanho do conjunto $[\![M_{join}]\!]|_{Escravo_R}$ (linha 33). O volume de dados (SM_R) a ser transmitido em cada mensagem é o tamanho (em bytes) de um resultado intermediário. Sendo assim, um servidor $Escravo_L$ pode computar para cada servidor remoto $Escravo_R$, o custo de aplicar a estratégia *send-result* (SR) por:

$$Cost_{SR}[R] = NM_R * (cc + SM_R/tr)$$

onde cc é o custo (dependente de hardware) de estabelecer uma conexão do $Escravo_L$ para o $Escravo_R$ e tr é a taxa de transmissão (dependente de rede) do $Escravo_L$ para o $Escravo_R$.

Observe que vários mapeamentos em $[\![M_{join}]\!]|_{Escravo_R}$ podem compartilhar o mesmo valor para $varJoin$, ou seja, pode existir mais de um resultado intermediário que "aponte" para um mesmo vértice armazenado em um servidor $Escravo_R$. No exemplo corrente, se a estratégia *get-frag* for usada quando i=4, somente um fragmento deve ser transmitido do servidor remoto para $Escravo_L$ para continuar o processamento dos múltiplos resultados intermediários. Por meio do índice *IxRemoteFrag*, o $Escravo_L$ identifica os fragmentos remotos a serem recuperados no $Escravo_R$. Definiu-se aqui que estes conjuntos de vértices (distintos) sejam requisitados para um servidor $Escravo_R$ aplicando a estratégia *get-frag* como $[\![M_{join}]\!]|_{Escravo_R}[varJoin] = \{joinNode \mid m \in [\![M_{join}]\!]|_{Escravo_R}, e\ varJoin \mapsto joinNode\ \text{está}\ em\ m\}$ (linha 34). Sendo assim, o número de mensagens a serem transmitidas do servidor remoto $Escravo_R$ para o $Escravo_L$ (NM_R) é o tamanho do conjunto $[\![M_{join}]\!]|_{Escravo_R}[varJoin]$. O volume de dados a ser transmitido em cada mensagem varia conforme o padrão de alocação. O tamanho médio dos fragmentos de G_D (SM_{pat}) é determinado durante a fragmentação/distribuição da base G_D no *cluster*. Entretanto, este valor pode ser recalculado em períodos de tempo caso o fluxo de inserções seja intenso na aplicação. Aqui, um servidor $Escravo_L$ pode computar para cada servidor remoto $Escravo_R$, o custo de aplicar a estratégia *get-frag* (GF) por:

$$Cost_{GF}[R] = NM_R * (cc + SM_{pat}/tr + SM_{pat} * st)$$

onde cc e tr já foram definidos anteriormente, e st é o custo (dependente de repositório) de armazenar localmente o fragmento no $Escravo_L$.

Caso o custo de *send-result* ($Cost_{SR}[R]$) seja menor que o custo de *get-frag* ($Cost_{GF}[R]$) os resultados intermediários (mapeamentos) são enviados para o servidor R, que continua com o processamento da consulta (linha 41). Caso contrário, o servidor $Escravo_L$ requisita os fragmentos necessários e continua com a execução (linhas 43 e 44). Note que o $Escravo_L$ também pode possuir fragmentos de dados que apontam para $joinNode$ em sua base local, dispensando a função de custo (linha 37). Finalmente, quando um servidor Escravo explora o último passo de uma sequência de exploração, o conjunto de mapeamentos é projetado sobre as variáveis da cláusula *select* da consulta e o resultado final é enviado para o servidor Mestre (linha 50) .

Note que o custo para cada estratégia de comunicação é computada para cada servidor remoto (linha 36). Como um resultado, é possível que para um servidor remoto R_1, $Cost_{SR}[R_1] < Cost_{GF}[R_1]$, e para outro servidor remoto R_2, $Cost_{SR}[R_2] > Cost_{GF}[R_2]$. Então, este trabalho propõe o método *2ways*, que escolhe para cada servidor, a estratégia que representa o menor custo de comunicação no processamento de consultas.

Observe que foi considerado um modelo no qual múltiplas mensagens para um mesmo servidor não são "empacotadas" a fim de minimizar o custo do estabelecimento de conexões. Para tais sistemas,

a função de custo deveria ser modificada de acordo. Além disso, os valores de *cc*, *tr* e *st* podem ser diferentes entre pares de servidores e variar conforme o tempo. A calibração (dinâmica) destes valores não foi considerada neste trabalho. Os valores das constantes usados no estudo experimental do Capítulo 6 foram determinados empiricamente conforme o ambiente computacional usado nos experimentos.

Considerações

Este capítulo apresenta uma técnica de otimização, o método *2ways*, que possibilita a escolha da estratégia de comunicação mais adequada, *send-result* ou *get-frag*, para cada par de servidores durante o processamento distribuído de consultas. Um estudo experimental no Capítulo 6 mostra que *2ways* apresenta melhor desempenho do que cada estratégia considerada de maneira isolada.

A abordagem de processamento que implementa *2ways* planeja consultas baseando-se em padrões de alocação (PAs) definidos a partir de estruturas RDF. A execução de consultas é feita por meio de um algoritmo de exploração distribuída de grafos. O planejamento garante que durante o processamento de consultas entre servidores, dados de um mesmo fragmento sejam explorados localmente em bloco antes da exploração considerar outro fragmento de dado, que pode estar alocado remotamente.

O próximo capítulo apresenta uma abordagem de atualização de dados que estende a abordagem de processamento apresentada neste capítulo a fim de tratar a questão do dinamismo de bases RDF.

CAPÍTULO 5

UMA ABORDAGEM PARA A ATUALIZAÇÃO DE DADOS

Bases de dados podem ser alteradas por meio de operações de inserção e exclusão de dados. Considerando que bases RDF são classificadas semi-estruturadas, esquemas RDF podem ser alterados durante tais alterações de bases. Consequentemente, estes esquemas podem ser atualizados para representarem de maneira adequada o estado atual de suas respectivas bases, conforme mostra a Seção 2.5 do Capítulo 2.

Seguindo o mesmo raciocínio, este capítulo propõe uma abordagem de atualização de dados que dá suporte à abordagem de processamento proposta no capítulo anterior, uma vez que o grafo de sumarização G_S usado na distribuição de dados (grafo denominado de G_S *inicial* neste capítulo) pelo *módulo de armazenamento* da Figura 4.1 pode tornar-se inadequado para o planejamento de consultas da abordagem de processamento. De uma forma geral, a operação de exclusão da abordagem simplesmente elimina relacionamentos entre vértices da aplicação. Em contrapartida, a operação de inserção permite uma extensão limitada de G_S *inicial* a fim de permitir o armazenamento de novas triplas na aplicação RDF.

Intuitivamente, considerando que recursos alocados no repositório G_D são categorizados de acordo com PAs de G_S *inicial*, a ideia é que estes recursos possam viabilizar a categorização de novos dados (recursos e valores literais) durante inserções de novas triplas RDF. Em um primeiro momento, triplas RDF mapeáveis para vértices de PAs de G_S *inicial* são consideradas como categorizáveis e armazenadas diretamente em G_D. Um mapeamento ocorre quando pelo menos um dos membros da tripla é um recurso de G_D categorizado por um vértices de G_S *inicial* que possui o predicado da tripla. Aqui o termo *membro* refere-se ao sujeito e ao objeto de triplas RDF. Caso não ocorra o mapeamento, PAs podem ser estendidos a fim de categorizar novas triplas em G_S *inicial*. Considerando que a composição de um PA implica diretamente no volume de armazenamento de seus fragmentos, definiu-se aqui a restrição *one-hop* onde um vértice de G_S *inicial* pode ser estendido em no máximo *um* passo. Ou seja, durante uma extensão, cada novo vértice inserido em um PA deve se relacionar com um vértice de G_S *inicial*. Além disso, outra restrição denominada de *only-literal* define que somente objetos mapeados a partir de valores literais podem adicionar vértices em PAs de G_S *inicial*. Note que, a adição de vértices mapeados a partir de recursos poderia violar facilmente esta restrição, uma vez que os recursos em geral possuem propriedades. Para dar suporte ao armazenamento de membros de triplas que violarem as restrições, a operação de inserção adota um repositório distribuído secundário de dados, denominado aqui de *Overflow*. Além do suporte as restrições, o *Overflow* também é usado para armazenar triplas da aplicação que são desconexas de G_D. Isso é feito porque os membros de uma tripla desconexa de G_D não podem ser categorizados por meio dos PAs de G_S *inicial*.

Dados armazenados no *Overflow* são representados por meio de um novo vértice (v_O) no grafo de sumarização da aplicação. O padrão de alocação de v_O é denominado de *PaOverflow*. Perceba que agora o grafo de sumarização da aplicação (G_S) conta com os PAs de G_S *inicial* e com o *PaOverflow*. A fim de simplificar o processo de inserção de dados, o *PaOverflow* possui somente o vértice v_O e propriedades *inter-PA* que são representadas por meio de arestas recursivas em v_O. Estas propriedades representam relacionamentos existentes entre vértices de diferentes fragmentos do *Overflow*. Por fim, propriedades *inter-PA* entre vértices de PAs de G_S *inicial* e v_O representam relacionamentos entre vértices de G_D e do *Overflow*. As arestas de v_O são usadas para a validação de planos que envolvam o *PaOverflow*.

A possibilidade de extensão de G_S requer alterações em sua composição. Agora as arestas do grafo devem ser categorizadas como E_S, caso a aresta pertença à G_S *inicial*, ou como E_{Se}, caso a aresta tenha sido usada para estender G_S *inicial*. Esta informação é usada durante o armazenamento de novas triplas na Seção 5.2.1.2, dado que este trabalho adota a suposição de que somente triplas RDF com predicados representados por arestas pertencentes à E_S podem categorizar novos recursos em G_D. Além disso, os vértices do grafo também devem indicar qual o seu "tipo" de dado, ou seja, recurso ou valor literal. Esta informação é essencial para o armazenamento de triplas com propriedades *Intra-PA* na Seção 5.2.1.1. Sendo assim, diante destas alterações, os vértices de *PaPerson* da Figura 4.3(b) ficariam como: $v9$ = *(in:{(parentOf, v9, I, E_S), (friendOf, v9, E, E_S)}, out:{(name, v10, I, E_S), (parentOf, v9, I, E_S), (friendOf, v9, E, E_S)}, type:resource)* e $v10$ = *(in:{(name, v10, I, E_S)}, out:{ }, type:literal value)*.

O modelo de armazenamento de dados do *Overflow* adota o modelo nativo de grafos e os seus fragmentos baseiam-se no padrão estrutural do tipo estrela, onde cada sujeito é armazenado com os seus respectivos objetos que representam valores literais. Para isso, uma função *hash* $f(m)$ de ser usada para definir a localidade de recursos m de uma tripla (sujeito e/ou objeto) em um *cluster*. Quanto aos índices, o *Overflow* também faz uso do *IxRemoteServer* para que vértices de G_D relacionados com o *Overflow* sejam localizados. Além disso, novos índices são considerados, conforme mostra a Tabela 5.1. Aqui, *IxSubjects* e *IxObjects* possibilitam a recuperação dos *pontos iniciais de exploração* no *Overflow* indexando propriedades e referenciando sujeitos e objetos do repositório, respectivamente.

Ao longo deste capítulo serão apresentadas diversas requisições de inserção de triplas que podem ser feitas por meio do operador *INSERT DATA* da linguagem SPARUL, conforme descrito no Capítulo 2. Considere a base G_D original ilustrada na Figura 1.1.

Exemplo 5.1 *A inserção da tripla (Offer1, date, "15/4/8") é realizada no fragmento PaOffer_1 de G_D dado que a tripla é mapeável para ($v6$, date, $v8$) em PaOffer. Já, a inserção da tripla (Product1, color, "silver") estende PaProduct com a aresta color e um novo vértice, dado que o vértice $v3$ de G_S mapeado a partir do sujeito da tripla (Product1) não possui a aresta color e o objeto da tripla é um valor literal. Desta forma, a tripla é armazenada no fragmento PaProduct_1 em G_D. Em contrapartida, a inserção da tripla (Review1, reviews, Offer1) viola a restrição only-literal dado que o vértice $v6$ de G_S mapeado*

	Chave de busca	Valor associado
IxRemoteServer	Identificador do vértice remoto	IP do servidor remoto
IxPaFrags	Padrão de alocação	{Identificadores dos fragmentos}
IxFragNodes	Identificador do fragmento	{Identificadores dos vértices}
IxRemoteFrag	Identificador do vértice remoto	Identificador do fragmento remoto
IxSubjects	predicado	{Identificadores dos vértices sujeitos}
IxObjects	predicado	{Identificadores dos vértices objetos}

Tabela 5.1: Tabela de índices estendida devido ao uso do *Overflow*

a partir do objeto da tripla (Offer1) não possui a aresta reviews e o sujeito da tripla é um recurso que não existe na aplicação. Desta forma, o recurso Review1 é armazenado no Overflow e G_S *é estendido com a aresta reviews entre os vértices* $v6$ *e* v_O*. Por fim, a inserção da tripla (Review2, rating, "10") é realizada no Overflow, dado que o recurso Review2 da tripla não existe em* G_D*.* □

Perceba que com a adoção do *Overflow*, a abordagem de processamento proposta no capítulo anterior exige alguns ajustes para que ambos os repositórios sejam considerados na execução de consultas. Este assunto é tratado na Seção 5.3.

5.1 Arquitetura estendida

A abordagem de atualização proposta requer uma extensão da arquitetura da Figura 4.1, conforme mostra a Figura 5.1. Agora a arquitetura conta com o *módulo de atualização*, que é o responsável pelo gerenciamento de inserção e exclusão de triplas de uma dada aplicação. Com a nova arquitetura, dada uma requisição de inserção de uma tripla (fluxo 11), o Mestre inicia o processo de inserção consultando o grafo de sumarização corrente da aplicação em seus metadados (fluxo 12) e trocando mensagens com os Escravos do *cluster* (fluxo 13). Durante o processo de inserção, cada Escravo consulta os seus repositórios locais determinando a inserção da tripla (fluxos 14 e 15). Por fim, ainda no processo de inserção, o Mestre pode voltar a acessar os seus metadados a fim de estender o grafo de sumarização da aplicação, se necessário (fluxo 12). Quanto à operação de exclusão, a requisição é enviada para todos os Escravos do *cluster* (fluxo 13). Cada Escravo analisa o seu repositório local (fluxos 14 e 15) providenciando a exclusão da tripla, quando possível. Tanto a operação de inserção quanto a de exclusão podem ser resolvidas localmente por um único Escravo ou de maneira distribuída (fluxo 13).

5.2 Atualização de dados

Esta seção descreve as operações de inserção e de exclusão definidas para a abordagem de atualização proposta. Primeiramente, a operação de inserção é descrita e, na sequência, a de exclusão.

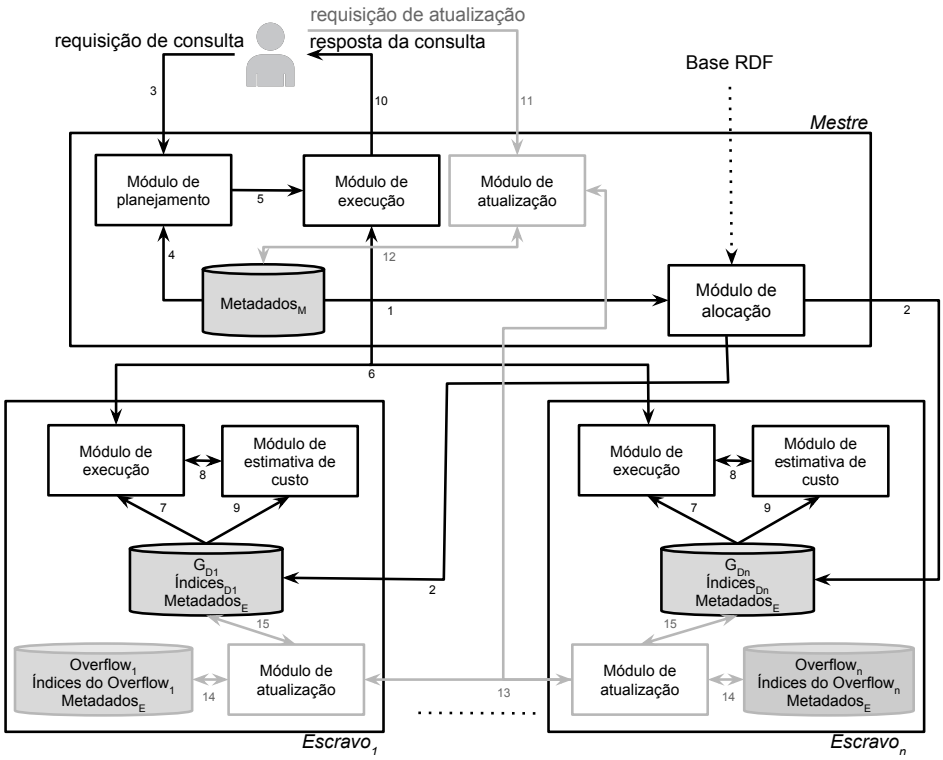


Figura 5.1: Arquitetura estendida

5.2.1 Inserção de dados

A tripla (s, p, o) de uma requisição de inserção pode ser denominada como *classificável* ou *não-classificável*. Uma tripla é *classificável* quando *pelo menos* um de seus membros pode ser categorizado em G_S *inicial*, caso contrário a tripla é denominada de *não-classificável*. Sendo assim, uma tripla (s, p, o) é *não-classificável* quando: *1)* s e o não existem na aplicação no momento da inserção; ou, *2)* s e o são recursos alocados no *Overflow*; ou, *3)* um membro da tripla representa um recurso alocado no *Overflow* e o outro membro não existe na aplicação. É essencial ressaltar que somente os membros que representam recursos nas triplas são usados para definir uma tripla como *classificável* ou *não-classificável*. Membros com valor literal são sempre considerados novos vértices não existentes na aplicação.

O processo de inserção é dividido em duas etapas, sendo elas: *1)* a extensão de G_S quando necessário; e, *2)* o armazenamento da tripla da requisição. O processamento de cada requisição baseia-se no estado corrente da aplicação, ou seja, em G_S, em G_D e no *Overflow*. Qualquer tipo de tripla RDF pode estender o grafo de sumarização corrente de uma aplicação. Durante o processo de inserção, requisições com triplas *não-classificáveis* são armazenadas diretamente no *Overflow*. Já, requisições com triplas *classificáveis* requerem uma análise mais cuidadosa na inserção dado que os seus membros podem ser categorizados de acordo com G_S *inicial*. Consequentemente, todas as inserções implicam em atualizações nos índices de G_D e/ou do *Overflow*. As situações de extensão e de armazenamento são apresentadas a seguir.

5.2.1.1 Extensões em G_S

Enquanto o predicado de uma tripla *não-classificável* pode estender G_S somente com propriedades *inter-PA*, triplas *classificáveis* estendem G_S tanto com propriedades *intra-PA* quanto com *inter-PA*.

Considerando uma tripla *não-classificável* (s, p, o), o grafo de sumarização da aplicação é estendido somente quando s e o representam recursos e p não existe como uma aresta recursiva em v_O. Neste caso, a extensão deve adicionar a aresta (v_O, p, v_O) em G_S para que a tripla possa ser armazenada na aplicação. Desta forma, caracteriza-se a primeira situação de extensão da abordagem (S:O-O).

Considerando uma tripla *classificável* (s, p, o), uma propriedade *intra-PA* é adicionada em G_S em duas situações, sendo elas:

S:G_D-Lit) Esta situação ocorre quando s é um *recurso* representado por um vértice v_D de V_D, o é um *valor literal* e p não existe como propriedade *intra-PA* entre $type(v_D)$ e um vértice de G_S que represente um valor literal. Aqui, um novo vértice v é criado no PA de $type(v_D)$ e G_S é estendido com a tripla $(type(v_D), p, v)$ onde p é *intra-PA*;

S:Intra-G_D-G_D) Esta situação caracteriza-se quando s e o são *recursos* representados por vértices de V_D, v_{D_1} e v_{D_2}, que existem em um mesmo fragmento em G_D e p não existe como uma propriedade *intra-PA* entre os vértices $type(v_{D_1})$ e $type(v_{D_2})$ em G_S. Logo, G_S é estendido com a tripla $(type(v_{D_1}), p, type(v_{D_2}))$ onde p é *intra-PA*.

Continuando com triplas *classificáveis*, uma propriedade *inter-PA* é adicionada em G_S em três situações, sendo elas:

S:Inter-G_D-G_D) Esta situação ocorre quando s e o são *recursos* representados por vértices de V_D, v_{D_1} e v_{D_2}, que existem em diferentes fragmentos em G_D e p não existe como propriedade *inter-PA* entre $type(v_{D_1})$ e $type(v_{D_2})$ em G_S. Aqui, G_S é estendido com a tripla $(type(v_{D_1}), p, type(v_{D_2}))$ onde p é *inter-PA*;

S:G_D-O) Esta situação caracteriza-se quando um membro da tripla é um *recurso* representado por um vértice v_D de V_D, o outro membro é um *recurso* alocado no *Overflow* e p não existe como uma propriedade *inter-PA* entre $type(v_D)$ e v_O em G_S. Com a extensão em G_S, o predicado p torna-se uma propriedade *inter-PA* entre o vértice $type(v_D)$ e o vértice v_O;

S:G_D-newO) Esta situação ocorre quando um membro é um *recurso* representado por um vértice v_D de V_D, o outro membro é um *recurso* que não existe na aplicação e p não existe como uma propriedade inter-PA em $type(v_D)$. Neste caso, G_S é estendido com a tripla $(type(v_D), p, v_O)$ onde p é *inter-PA*.

Perceba que em todas as análises de extensão os relacionamentos entre os membros das triplas devem obedecer a direção de s para o.

5.2.1.2 Armazenamento de triplas

Após a análise de G_S, a tripla de uma requisição pode finalmente ser armazenada. Neste momento, assume-se que G_S foi estendido pela requisição em questão, caso uma extensão seja necessária.

Triplas *não-classificáveis* (s, p, o) são armazenadas diretamente no *Overflow* envolvendo diferentes situações de armazenamento. Caso s e o não existam na aplicação, os vértices de seus respectivos membros são criados e armazenados no *Overflow* usando a função *hash* que define a localidade de vértices no *Overflow*. Caso s represente um recurso e o um valor literal, a tripla é armazenada no Escravo que alocará s (A:newO-Lit). Caso s e o representem recursos, cada vértice é criado e armazenado em seu respectivo Escravo (A:newO-newO).

Continuando com triplas *não-classificáveis*, caso um membro exista no *Overflow* e o outro membro não exista na aplicação, o membro inexistente é criado e armazenado no *Overflow*. Da mesma forma que em A:newO-Lit, caso s seja um recurso e o um valor literal, a tripla é armazenada no Escravo que aloca s (A:O-Lit). Caso, o membro inexistente represente um recurso, o vértice do membro inexistente é criado e armazenado no Escravo apropriado (A:O-newO). Por fim, caso s e o representem recursos que já existam no *Overflow*, os vértices referentes aos membros são simplesmente atualizados (A:O-O). As situações A:newO-newO, A:O-newO e A:O-O podem estender G_S por meio de S:O-O.

Já, a inserção de triplas *classificáveis* (s, p, o) depende das propriedade de G_S e das bases envolvidas na inserção. Considerando que p representa uma propriedade *intra-PA* em G_S, a operação de inserção deve garantir que s e o fiquem alocados em um mesmo fragmento. Porém, nas triplas *classificáveis*, o pode ser representado tanto por um *valor literal* quanto por um *recurso*. Tratando-se de valor literal, a tripla é armazenada localmente no repositório G_D do Escravo que aloca s (A:G_D-Lit). Aqui G_S pode ser estendido com S:G_D-Lit. Considerando o como *recurso*, enquanto um dos membros da tripla pertence à G_D, o outro membro pode pertencer à G_D, ao *Overflow* ou simplesmente não existir na aplicação. O primeiro membro é denominado aqui como $membro_{G_D}$ e o segundo como $membro_{flex}$. Neste caso, o armazenamento de triplas *classificáveis* com propriedades *intra-PA* pode ser feito das seguintes formas:

A:Intra-G_D-G_D) Esta situação ocorre quando o $membro_{flex}$ pertence ao mesmo fragmento que o $membro_{G_D}$ em G_D. Aqui os membros da tripla são simplesmente atualizados com a requisição e G_S pode ser estendido por meio de S:Intra-G_D-G_D;

A:Intra-G_D-newG_D) Esta situação caracteriza-se quando o $membro_{flex}$ representa um *recurso* que não existe na aplicação, p é uma aresta de E_S que existe como uma propriedade *intra-PA* entre $type(membro_{G_D})$ e um vértice v_S de V_S que represente uma recurso em G_S, o vértice $membro_{flex}$ é criado em G_D no mesmo fragmento que aloca o vértice $membro_{G_D}$. Desta forma, o $membro_{flex}$ é categorizado de acordo com o vértice v_S. Note que aqui G_S não pode ser estendido.

Em contrapartida, considerando o armazenamento de triplas *classificáveis* com propriedades *inter-PA*, as seguintes situações podem ocorrer durante a inserção de uma nova tripla:

A:Inter-G_D-G_D) Esta situação ocorre quando ambos os membros, $membro_{G_D}$ e $membro_{flex}$, pertencem à V_D. Aqui, os vértices de ambos os membros são simplesmente atualizados em seus respectivos fragmentos com a requisição e G_S pode ser estendida por meio de S:Inter-G_D-G_D;

A:Inter-G_D-O) Esta situação caracteriza-se quando o $membro_{G_D}$ pertence à V_D e o $membro_{flex}$ ao *Overflow*. Sendo assim, ambos os membros são atualizados em seus respectivos repositórios e G_S pode ser estendido por meio de S:G_D-O.

Por fim, dois tipos de armazenamento podem ocorrer quando o $membro_{G_D}$ pertence à V_D e o $membro_{flex}$ não existe na aplicação. No primeiro, caso p seja uma aresta no grafo de sumarização inicial (E_S) e relacione $type(membro_{G_D})$ com algum vértice v_S de V_S, ambos os membros são inseridos em G_D. Desta forma, o novo vértice é categorizado de acordo com o vértice v_S (A:Inter-G_D-*new*G_D). No segundo, o $membro_{G_D}$ é atualizado em G_D e o $membro_{flex}$ é inserido no *Overflow* (A:G_D-newO). Aqui G_S pode ser estendido por meio de S:G_D-newO na situação A:G_D-newO.

Note que para as situações de armazenamento A:Intra-G_D-*new*G_D e A:Inter-G_D-*new*G_D é adotada a suposição de que um novo *recurso* pode ser categorizado de acordo com G_S *inicial* somente se p pertencer à E_S. Caso contrário, G_S é estendido por meio de S:G_D-newO e a tripla da requisição é armazenada por meio de A:G_D-newO.

5.2.1.3 Execução de requisições de inserção

Os Algoritmos 4 e 5 representam as operações executadas pelo Mestre e Escravos, respectivamente, para a inserção de triplas em uma base. As Tabelas 5.2, 5.3, 5.4 e 5.5 listam uma sequência de requisições de inserções de triplas que é aplicada na base exemplo da Figura 1.1 ao longo desta seção. Para cada requisição de uma tabela é definido o tipo da tripla (*não-classificável* (NC) ou *classificável*), o tipo de propriedade da tripla (*intra-PA* ou *inter-PA*), a situação de extensão (se existir) e a situação de armazenamento. Na Figura 5.2(b) é possível visualizar todas as extensões em G_S (Figura 5.2(a)) promovidas pelas requisições das tabelas mencionadas. As extensões estão destacadas com arestas tracejadas e rótulos de arestas sublinhados. A Figura 5.3 ilustra as atualizações em G_D promovidas pelas requisições das tabelas. As alterações estão sinalizadas em cinza na figura. O *Overflow* gerado pela sequência de inserções é exibido na Figura 5.4. Nas Figuras 5.3 e 5.4, os vértices branco simbolizam vértices armazenados em outro tipo de repositório. Ou seja, na Figura 5.3 os vértices branco referenciam vértices do *Overflow*, e na Figura 5.4 os vértices de G_D.

Por meio do Algoritmo 4, o Mestre envia uma requisição de inserção para todos os Escravos do *cluster* e espera por respostas por um determinado tempo pré-estipulado baseado no ambiente de execução que implementa a arquitetura da Figura 5.1. Dado que a operação de inserção baseia-se na distribuição *controlada* de dados, o Mestre também envia aos Escravos informações de G_S relacionados à requisição (T) (linhas 4 e 5). Cada elemento de T representa uma tripla de G_S que possui o predicado da tripla da

Algoritmo 4: InsertionMaster

```
1  Entrada [(s, p, o)];
2  Saída { };
3  T = { }; Answer_1 = null; Answer_2 = null;
4  Let T be data about triples of G_S with p;
5  Send ((s, p, o), T) to all slaves and wait answers;
6  if there are no answers then
7     if (s is a resource) AND (o is a resource) AND (p does not connect v_O to v_O in G_S) then
8        add p as a recursive inter-PA property on v_O;
9     end
10    create nodes s and o on Overflow updating its indexes;
11 else
12    Answer_1 ← first answer;
13    if (there is a second answer) then
14       Answer_2 ← second answer;
15    end
16    if (Answer_1.status = inserted) then
17       if (Answer_1.operation = extends) then
18          if (Answer_1.propertyType = I) then
19             extends G_S with Answer_1.tripleG_S.p as an intra-PA property between Answer_1.tripleG_S.s and
                 Answer_1.tripleG_S.o;
20          else
21             extends G_S with Answer_1.tripleG_S.p as an inter-PA property between Answer_1.tripleG_S.s and
                 Answer_1.tripleG_S.o;
22          end
23       end
24    end
25    if (Answer_1.status = pending) then
26       if (Answer_2.status = pending) then
27          if (Answer_1.triple.p does not connect Answer_1.nodeG_S to Answer_2.nodeG_S in G_S) then
28             extends G_S with Answer_1.triple.p as an inter-PA property between Answer_1.nodeG_S to Answer_2.nodeG_S;
29          end
30          send messages to slaves Answer_1.IdServer and Answer_2.IdServer updating their nodes and indexes;
31       else
32          /* If Answer_2 is null */
33          if (Answer_1.nodeG_D = s) then
34             newNode = o;
35          else
36             newNode = s;
37          end
38          if (Answer_1.nodeG_S ∈ initial G_S) then
39             if (Answer_1.triple.p connects Answer_1.nodeG_S to a node v_2 of V_S by a property E_S in G_S) then
40                create newNode according v_2 on G_D of Answer_1.IdServer ;
41                send message to Answer_1.IdServer updating its nodes and indexes;
42             else
43                if (Answer_1.triple.p does not connect Answer_1.nodeG_S to v_O in G_S) then
44                   extends G_S with Answer_1.triple.p between Answer_1.nodeG_S and v_O;
45                end
46                send messages to Answer_1.IdServer and the other slave that will allocate the newNode updating their
                    nodes and indexes;
47             end
48          else
49             if (Answer_1.nodeG_S = v_O) then
50                if (Answer_1.triple.p does not connect R1.nodeG_S to v_O in G_S) then
51                   extends G_S with Answer_1.triple.p between Answer_1.nodeG_S and v_O;
52                end
53                send messages to Answer_1.IdServer and the other slave that will allocate the newNode updating their
                    nodes and indexes;
54             end
55          end
56       end
57    end
58 end
```

requisição em questão. Cada elemento contém: uma tripla de G_S, os tipos da aresta da tripla (se é I ou E; e, E_S ou E_{Se}) e os tipos dos vértices da tripla (recurso ou valor literal). Por exemplo, na requisição $I6$ da Tabela 5.2 o Mestre informa aos Escravos que existe uma tripla em G_S com o predicado p envolvido na requisição, que p é uma propriedade *inter-PA* pré-existente em G_S *inicial*, e que p relaciona recursos, logo T={((v9, friendOf, v9), (E, E_S), (resource, resource))}. Essas informações são usadas pelos Escravos durante o processo de inserção. Todos os Escravos recebem a requisição iniciando o processamento da requisição em paralelo com o Algoritmo 5.

	Requisições: INSERT DATA	*Tipo de tripla*	*Tipo de propriedade*	*Extensão*	*Armazenamento*
I1	{*Review1 rating "9"*}	*NC*	*intra-PA*		A:newO-Lit
I2	{*Review2 rating "10"*}	*NC*	*intra-PA*		A:newO-Lit
I3	{*Review3 rating "3"*}	*NC*	*intra-PA*		A:newO-Lit
I4	{*Product6 name "Uno"*}	*NC*	*intra-PA*		A:newO-Lit
I5	{*Offer7 value "55000"*}	*NC*	*intra-PA*		A:newO-Lit
I6	{*Person7 friendOf Person8*}	*NC*	*inter-PA*	S:O-O	A:newO-newO

Tabela 5.2: Classificação, tipo de propriedade, situação de extensão e de armazenamento das triplas das requisições $I1$–$I6$ de inserção

Caso o Mestre não receba respostas dos Escravos, a tripla da requisição caracteriza-se como *não-classificável*; G_S é estendido, se necessário; e, a tripla é armazenada no *Overflow* (linhas 6–10). As requisições $I1$–$I6$ da Tabela 5.2 representam este tipo de situação. Em $I1$–$I5$ os sujeitos são recursos que não existem na aplicação e os objetos são valores literais caracterizando a situação de armazenamento $A : newO - Lit$. Já, $I6$ ilustra um exemplo que envolve a situação de extensão S:O-O, pois os membros da sua tripla representam recursos e o vértice v_O de G_S não possui p como uma propriedade recursiva. A situação de armazenamento de $I6$ é a A:newO-newO.

Continuando, se ao menos um membro da tripla existir na aplicação como *recurso*, respostas são retornadas ao Mestre, *Answer1* e/ou *Answer2* (linhas 12–15). As respostas são definidas pelos Escravos por meio do Algoritmo 5. Cada Escravo inicia em paralelo o processamento de inserção da tripla (s, p, o) de uma requisição procurando por s e o em seu repositório local (linhas 4–9). A tentativa de inserção continua somente nos Escravos que possuem pelo menos um dos membros da tripla.

A alocação de uma tripla pode ser feita localmente por um único Escravo ou distribuída e gerenciada pelo Mestre. Escravos retornam respostas ao Mestre seguindo o padrão *(IdServer, Answer)* onde: *IdServer* é um identificador que denomina um Escravo no *cluster*; e, *Answer* define a situação de uma requisição. *Answer* possui os elementos [`status, operation`] que podem assumir os valores: [`inserted, no_extends`], [`inserted, extends,` $tripleG_S$`, propertyType`] ou [`pending, triple,` $nodeG_D$`,` $nodeG_S$]. A primeira resposta sinaliza ao Mestre que o Escravo foi capaz de armazenar uma tripla de forma autônoma sem a necessidade de estender G_S. A segunda sinaliza que o Escavo armazenou a tripla e que G_S deve ser estendido. A extensão é feita com a tripla $tripleG_S$ onde o predicado deve ser do tipo `propertyType` em G_S. Por fim, `pending` sinaliza que um Escravo possui um dos membros da tripla em sua base local, porém que ele não é capaz de armazenar a tripla de forma autônoma. Pendências podem ocorrer tanto com triplas *classificáveis* quanto com triplas *não-classificáveis*. Pendências ocorrem quando: *1)* s e o representam recursos existentes em Escravos diferentes; ou, *2)* s e o representam recursos, porém um deles não existe na aplicação. O Escravo informa ao Mestre a tripla da requisição pendente (`triple`), o membro da tripla armazenado em sua base local ($nodeG_D$) e o mapeamento do seu vértice para G_S ($nodeG_S$). O Mestre usa estas respostas para gerenciar a inserção de triplas.

	Requisições: INSERT DATA	*Tipo de tripla*	*Tipo de propriedade*	*Extensão*	*Armazenamento*
I7	{*Person7 name "Paulo"*}	*NC*	*intra-PA*		A:O-Lit
I8	{*Person8 name "João"*}	*NC*	*intra-PA*		A:O-Lit
I9	{*Product6 value "58000"*}	*NC*	*intra-PA*		A:O-Lit
I10	{*Offer7 date "16/6/10"*}	*NC*	*intra-PA*		A:O-Lit
I11	{*Offer1 date "15/4/8"*}	*C*	*intra-PA*		A:G_D-Lit
I12	{*Product1 color "silver"*}	*C*	*intra-PA*	$S : G_D - Lit$	A:G_D-Lit

Tabela 5.3: Classificação, tipo de propriedade, situação de extensão e de armazenamento das triplas das requisições $I7$–$I12$ de inserção

	Requisições: INSERT DATA	*Tipo de tripla*	*Tipo de propriedade*	*Extensão*	*Armazenamento*
I13	{*Review3 reviews Offer7*}	*NC*	*inter-PA*	S:O-O	A:O-O
I14	{*Person6 fatherOf Person1*}	*C*	*intra-PA*	S:Intra-G_D-G_D	A:Intra-G_D-G_D
I15	{*Product3 indicates Product1*}	*C*	*inter-PA*	S:Inter-G_D-G_D	A:Inter-G_D-G_D
I16	{*Review1 reviews Offer1*}	*C*	*inter-PA*	S:G_D-O	A:Inter-G_D-O
I17	{*Review2 reviews Offer2*}	*C*	*inter-PA*		A:Inter-G_D-O

Tabela 5.4: Classificação, tipo de propriedade, situação de extensão e de armazenamento das triplas das requisições $I13$–$I17$ de inserção

Dando início à tentativa de inserção de uma tripla (s, p, o), cada Escravo procura por s e o, caso o represente um recurso. A existência de s no Escravo é sinalizada pelas variáveis s_L e $base_{s_L}$ (linha 5), onde a segunda variável indica o tipo de repositório que aloca s. Da mesma forma, as variáveis o_L e $base_{o_L}$ sinalizam a existência de o no Escravo (linha 8).

Caso um Escravo possua s e o seja um *valor literal*, a tripla é inserida no Escravo (linhas 10–17). $I7$–$I10$ ilustram exemplos de requisições com triplas *não-classificáveis*, e $I11$ e $I12$ com triplas *classificáveis*. As triplas de $I7$–$I10$ são armazenadas por meio da situação A:O-Lit. Já as triplas de $I11$ e $I12$ consideram a situação A:G_D-Lit. Além disso, $I12$ implica na extensão de G_S por meio da situação S:G_D-Lit.

A tentativa de inserção continua caso o seja um *recurso*. A princípio, considera-se a existência de s e o em um único Escravo (linhas 19–62). Primeiramente, o Escravo analisa o seu *Overflow* (linhas 20–26). Caso s e o pertençam ao *Overflow*, os vértices são atualizados e informações à respeito da extensão de G_S são retornadas ao Mestre, caso necessário. $I13$ ilustra este tipo de situação estendendo G_S por meio de S:O-O e armazenando a tripla por meio de A:O-O.

Na sequência, caso a tripla ainda não tenha sido inserida no Escravo, G_D é analisado (linhas 28–50). Caso s e o pertençam à G_D, propriedades *intra-PA* (linhas 37–42) ou propriedades *inter-PA* (linhas 44–48) podem estar envolvidas na requisição. $I14$ ilustra o primeiro caso estendendo G_S por meio de S:Intra-G_D-G_D e armazenando a tripla por meio de A:Intra-G_D-G_D. Já, $I15$ ilustra o segundo caso adotando A:Inter-G_D-G_D e S:Inter-G_D-G_D.

Por fim, o Escravo ainda analisa ambos os repositórios, caso a tripla ainda não tenha sido inserida por ele na aplicação (linhas 51–57). $I16$ e $I17$ ilustram este tipo de situação. $I16$ estende G_S usando S:G_D-O e ambas requisições armazenam dados por meio de A:Inter-G_D-O.

Algoritmo 5: InsertionSlave

```
1  Entrada[(s, p, o), T];
2  Saída[R];
3  R = { }; s_L = null; o_L = null; base_{s_L} = null; base_{o_L} = null;
4  if (s exists in local G_D or Overflow) then
5      s_L = s; base_{s_L} = base where s is allocated (G_D or Overflow);
6  end
7  if (o is a resource) AND (o exists in local G_D or Overflow) then
8      o_L = o; base_{o_L} = base where o is allocated (G_D or Overflow);
9  end
10 if (s_L is not null) AND (o is a literal value) then
11     if (base_{s_L} is G_D) AND (type(s_L) has not p as an intra-PA property in T) then
12         R = {IdServer, [inserted, extends, (type(s_L), p, a new value to type(o)), I]};
13     else
14         R = {Idserver, [inserted, no_extends]};
15     end
16     insert (s, p, o) on base_{s_L} and update it indexes;
17     return R;
18 else
19     if (base_{s_L} is not null) AND (base_{o_L} is not null) then
20         if ((base_{s_L} is Overflow) AND (base_{o_L} is Overflow)) then
21             if p does not exist on v_O in T then
22                 R = {IdServer, [inserted, extends, (v_O, p, v_O), E]};
23             else
24                 R = {IdServer, [inserted, no_extends]};
25             end
26             insert (s, p, o) on Overflow updatings its indexes;
27         else
28             if (base_{s_L} is G_D) then
29                 membro_{G_D} = s_L; base_G = base_{s_L}; membro_{flex} = o_L; base_{flex} = base_{o_L};
30             else
31                 if (base_{o_L} is G_D) then
32                     membro_{G_D} = o_L; base_G = base_{o_L}; membro_{flex} = s_L; base_{flex} = base_{s_L};
33                 end
34             end
35             if (base_G is G_D) then
36                 if (base_{flex} is G_D) then
37                     if (membro_{G_D} fragment = membro_{flex} fragment) then
38                         if (p exists between type(membro_{G_D}) and type(membro_{flex}) as an intra-PA property in T) then
39                             R = {IdServer, [inserted, no_extends]};
40                         else
41                             R = {IdServer, [inserted, extends, (type(membro_{G_D}), p, type(membro_{flex}), I]};
42                         end
43                     else
44                         if (p exists between type(membro_{G_D}) and type(membro_{flex}) as an inter-PA property in T) then
45                             R = {IdServer, [inserted, no_extends]};
46                         else
47                             R = {IdServer, [inserted, extends, (type(membro_{G_D}), p, type(membro_{flex}), E]};
48                         end
49                     end
50                     update s and o on G_D and theirs indexes on G_D;
51                 else
52                     if (base_{flex} is Overflow) then
53                         if (p exists between type_{membro_{G_D}} and v_O as an inter-PA property in T) then
54                             R = {IdServer, [inserted, no_extends]};
55                         else
56                             R = {IdServer, [inserted, extends, (type(membro_{G_D}), p, type(membro_{flex}), E]};
57                         end
58                         updade s on base_{s_L} and its indexes;
59                         update o on base_{o_L} and its indexes;
60                     end
61                 end
62             end
63         end
64         return R;
65     else
66         if (base_{s_L} is not null) AND (base_{o_L} is null)) then
67             membro_{exists} = s_L;
68         else
69             if (base_{o_L} is not null) AND (base_{s_L} is null)) then
70                 membro_{exists} = o_L;
71             end
72         end
73         R = {IdServer, [IdServer, pending, (s, p, o), membro_{exists}, type(membro_{exists})]};
74         return R;
75         wait instructions to finalize the triple insertion;
76     end
77 end
```

	Requisições: INSERT DATA	*Tipo de tripla*	*Tipo de propriedade*	*Extensão*	*Armazenamento*
I18	{*Offer7 offers Product6*}	*NC*	*inter-PA*	S:O-O	A:O-O
I19	{*Offer7 is_bought Person7*}	*NC*	*inter-PA*	S:O-O	A:O-O
I20	{*Person9 parentOf Person3*}	*C*	*intra-PA*		A:Intra-G_D-newG_D
I21	{*Person9 friendOf Person2*}	*C*	*inter-PA*		A:Inter-G_D-G_D
I22	{*Person4 parentOf Person5*}	*C*	*inter-PA*	S:Inter-G_D-G_D	A:Inter-G_D-G_D
I23	{*Person1 fatherOf Vendor1*}	*C*	*inter-PA*	S:Inter$-G_D$-G_D	A:Inter-G_D-G_D
I24	{*Person5 parentOf Person8*}	*C*	*inter-PA*	S:G_D-O	A:Inter-G_D-O
I25	{*Offer8 offers Product1*}	*C*	*inter-PA*		A:Inter-G_D-newG_D
I26	{*Product4 indicates Product2*}	*C*	*inter-PA*	S:G_D-newO	A:G_D-newO
I27	{*Offer9 offers Product6*}	*NC*	*inter-PA*		A:O-newO
I28	{*Review4 reviews Offer9*}	*NC*	*inter-PA*		A:O-newO

Tabela 5.5: Classificação, tipo de propriedade, situação de extensão de armazenamento das triplas das requisições $I18$–$I28$ de inserção

Em contrapartida, caso o Escravo possua somente s ou o da tripla em sua base, a tripla não pode ser inserida localmente de forma autônoma (linhas 63–74). Sendo assim, o Escravo informa a pendência de inserção da tripla ao Mestre para que ele tome as devidas providências e retorne instruções para a finalização da inserção. $I18$–$I28$ ilustram este tipo de situação. Por exemplo, em $I18$, enquanto o Escravo Y informa a pendência *[Y, pending, (Offer7, offers, Product6), Offer7,* v_O*]* para o Mestre, o Escravo X informa para pendência *[X, pending, (Offer7, offers, Product6), Product6,* v_O*]*.

Voltando ao Algoritmo 4, o Mestre pode receber sinalizações de extensão de G_S (linhas 16–24) quando ocorre a inserção de uma tripla (s, p, o) em um determinado Escravo. A extensão pode envolver uma propriedade *intra-PA* ou *inter-PA*. Continuando, caso a inserção de uma tripla esteja pendente, o Mestre define a inserção na aplicação de acordo com a(s) resposta(s) do(s) Escravo(s) (linhas 25–58). O processo é semelhante ao adotado pelos Escravos.

Caso o Mestre receba duas respostas de pendências de Escravos, três situações podem ocorrer no processo de inserção (linhas 25–31). Na primeira situação, s e o estão alocados no *Overflow* e os membros da tripla são simplesmente atualizados, como em $I18$ e $I19$. Ambas as requisições estendem G_S com S:O-O e armazenam triplas com A:O-O. Na segunda situação, s e o pertencem à G_D. Esta situação é exemplificada com $I21$–$I23$ que armazenam triplas com A:Inter-G_D-G_D e estendem G_S com S:Inter-G_D-G_D em $I22$ e $I23$. Na terceira situação, caso os repositórios envolvidos nas pendências variem entre G_D e *Overflow* (A:Inter-G_D-O), os vértices são atualizados em seus respectivos repositórios, como em $I24$. Com $I24$, G_S não é estendido.

O recebimento de uma única resposta de pendência implica na criação de um vértice na aplicação (linhas 32–56). Caso o vértice retornado na resposta (s ou o) seja mapeado para um vértice v_{S1} de V_S, $p \in E_S$ e p relacione v_{S1} com outro vértice v_{S2} de V_S, o vértice inexistente na aplicação é criado de acordo com v_{S2} no mesmo Escravo que retornou a resposta (linhas 38–42). $I20$ e $I25$ ilustram este tipo de situação, onde $I20$ envolve uma propriedade *intra-PA* (A:Intra-G_D-*new*G_D) e $I25$ uma propriedade

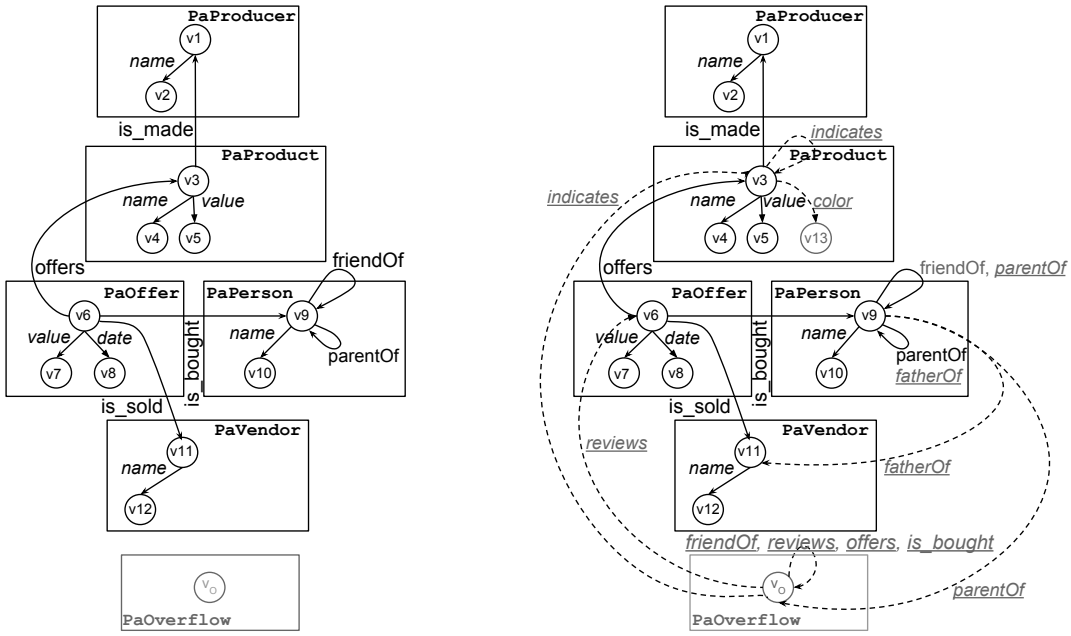


Figura 5.2: G_S (a); G_S após as inserções das tabelas de requisições (b)

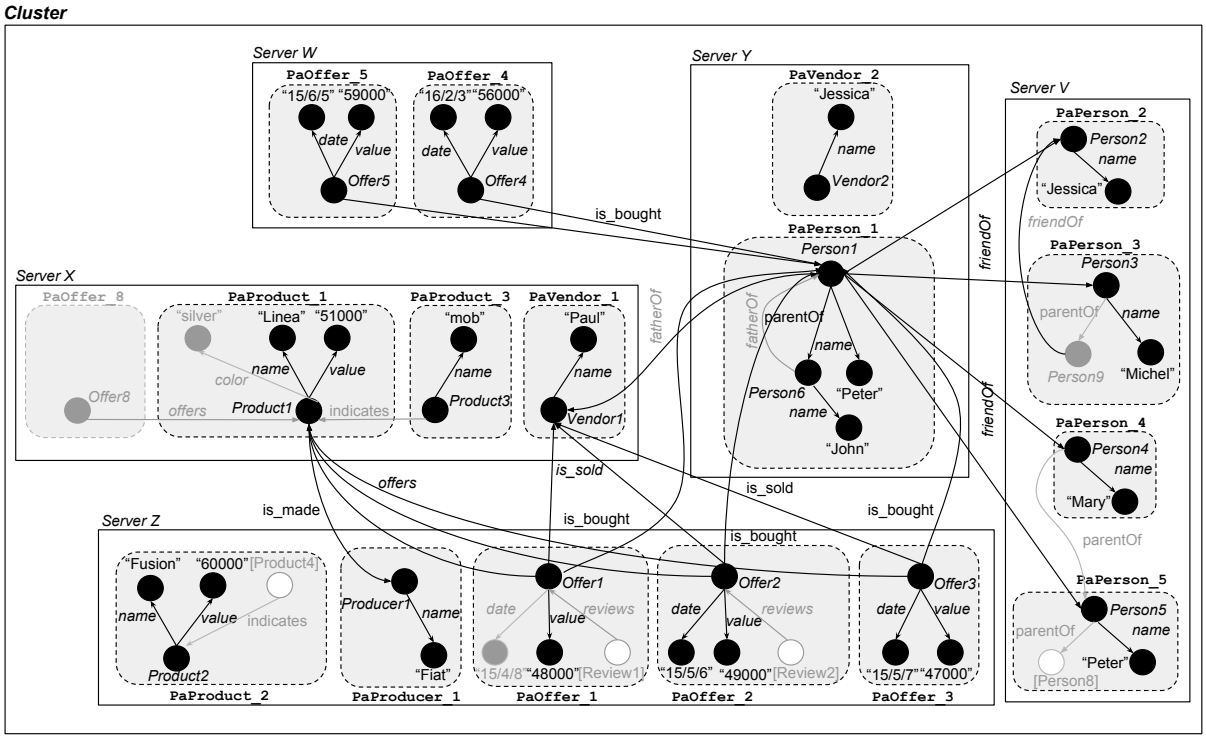


Figura 5.3: Base G_D atualizada após as inserções das tabelas de requisições

inter-PA (A:Inter-G_D-newG_D). Caso $p \notin E_S$ e $v_{S1} \in v_S$, o vértice inexistente na aplicação é criado no *Overflow* e ambos os membros são atualizados em seus respectivos repositórios, como em $I26$ (linhas 43–47) (A:Inter-G_D-newO). Continuando, caso a pendência tenha sido gerada a partir de um vértice do *Overflow*, a tripla é inserida no *Overflow* como em $I27$ e $I28$ (linhas 49–54) (A:O-newO).

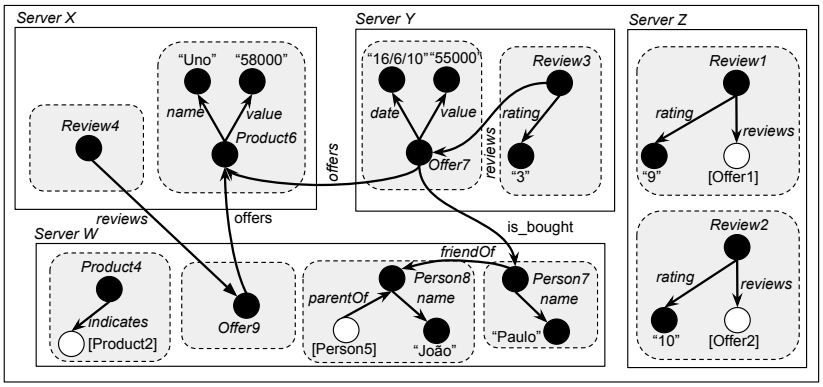


Figura 5.4: Base *Overflow* após a execução das inserções das tabelas de requisições

Note que, o *Overflow* poderia ser analisado em um determinado momento para a detecção de possíveis fragmentos de dados para G_D. Por exemplo, os fragmentos de *Product4*, *Product6*, *Offer7*, *Offer9*. *Person7* e *Person8* poderiam gerar dois fragmento do tipo *PaProduct*, dois dragmento do tipo *PaOffer* e dois fragmentos do tipo *PaPerson* em G_D, respectivamente. Além disso, novos PAs também poderiam ser detectados evoluindo G_S e auxiliando o *módulo de inserção* na categorização de novas triplas. Por exemplo, os fragmentos de *Review1–Review4* poderiam gerar um tipo de PA para revisões de ofertas. Porém, conforme citado na Seção 2.5, a evolução de esquemas RDF não é objeto de estudo deste trabalho.

5.2.2 Exclusão de dados

Dada uma requisição de exclusão de uma tripla (s, p, o), o Mestre requer que os Escravos analisem os seus repositórios e implementem a exclusão da tripla, se possível. A operação de exclusão adotada pela abordagem de atualização elimina o relacionamento entre dois vértices da aplicação e exclui vértices, se necessário.

O Algoritmo 6 ilustra a operação de exclusão adotada neste trabalho. Como na inserção, o Mestre informa aos Escravos a tripla da requisição e informações de G_S (T). Considere as requisições de exclusão $R1$: *DELETE DATA*{*Product1 name "Linea"*}, $R2$: *DELETE DATA*{*Producer6 friendOf Person1*}, $R3$: *DELETE DATA*{*Product2 indicates Product6*}, $R4$: *DELETE DATA*{*Offer2 is_sold Vendor1*}, $R5$: *DELETE DATA*{*Review4 reviews Offer9*} e $R6$: *DELETE DATA*{*Review1 reviews Offer1*}. Analisando as Figuras 5.3 e 5.4, enquanto $R1$ e $R2$ são resolvidas localmente pelos Escravos X e Y, respectivamente, as outras são resolvidas de maneira distribuída pelos escravos Z ($R3$, $R4$ e $R6$) e X ($R5$).

Dada uma requisição de exclusão de uma tripla (s, p, o), cada Escravo verifica a existência de s em sua base local (linha 4). Por exemplo, em $R1$, o Escravo X detecta a existência do recurso *Product1* em sua base G_D. Na sequência, caso p exista na lista de adjacência *out* do vértice de s "apontando" para o vértice de o, p é excluído da lista *out* do vértice de s (linha 6). Considerando que o predicado *name* existe na lista de adjacência *out* de *Product1* "apontando"para o vértice que representa o objeto

Algoritmo 6: DeletionSlave

```
1  Entrada[(s, p, o), T];
2  Saída[ ];
3  base_s = null;
4  if (s exists in local G_D or Overflow) then
5     if (p connects s to o on s.out) then
6        delete p of s.out;
7        base_s = base where s is allocated (G_D or Overflow);
8        if (s is empty) then
9           delete node s of base_s;
10          if (s fragment is empty) then
11             delete s fragment of base_s;
12          end
13       end
14       if (o is a literal value) then
15          delete node o of base_s;
16       else
17          if (base_s is G_D) AND (p connects type(s) to a resource as an intra-PA property in T) then
18             delete p of o.in;
19             if (o is empty) then
20                delete node o of base_s;
21                if (o fragment o is empty) then
22                   delete o fragment of base_s;
23                end
24             end
25          else
26             if (p connects type(s) to v_O in T) then
27                remote server R of o is defined by hash function used on Overflow distribution;
28             else
29                remote server R of o is defined by IxRemoteServer;
30             end
31             send message to R asking it to delete (s, p, o) of its repository analysing all consequences;
32          end
33       end
34       update indexes;
35    end
36 end
```

da tripla, p é excluído da lista *out* de *Product1*. Desta forma, o relacionamento entre o vértice de s e o vértice de o representado pela tripla (s, p, o) é eliminado da aplicação. Caso s não tenha mais atributos e relacionamentos, o vértice é excluído de seu repositório (G_D ou *Overflow*) (linhas 8 e 9). Caso o fragmento de s fique vazio após a exclusão do vértice, o fragmento também é excluído do repositório (linhas 10–12). A atualização de índices é considerada no final do algoritmo (linha 34), após a análise de exclusão de o.

Continuando, caso o objeto seja um valor literal, o vértice de o é excluído localmente no Escravo (linhas 14 e 15). $R1$ ilustra esta situação. A exclusão também é resolvida localmente se o representar um recurso que esteja relacionado com s por meio de uma propriedade *intra-PA* (linhas 17–24). $R2$ representa este tipo de situação onde (s, p, o) é mapeado para *(v9, fatherOf, v9)* em G_S. O vértice de o e o seu fragmento também são analisados para uma possível exclusão.

Em contrapartida, caso p represente uma propriedade *inter-PA* em G_S, a exclusão da tripla pode envolver vértices de servidores diferentes. Caso p "aponte"para o vértice v_O em G_S a partir de *type(s)*, a localização remota do vértice que representa o é definida por meio da função *hash* já citada neste capítulo (linhas 26 e 27). $R3$ e $R5$ ilustram este tipo de situação, onde os objetos de $R3$ e $R5$ pertencem ao *Overflow*. Caso p "aponte" para um vértice de G_S *inicial* a partir de *type(s)*, a localização remota do vértice que representa o é pesquisada no índice *IxRemoteServer* (linhas 26 e 27). $R4$ e $R6$ ilustram este tipo de situação, onde os objetos de $R4$ e $R6$ pertencem à G_D. Após a localização do servidor remoto de o, uma mensagem é enviada ao servidor para que ele providencie a exclusão de (s, p, o) em sua base local.

5.3 Execução de consultas com o *Overflow*

A existência de dados no *Overflow* implica em alterações na abordagem de processamento proposta no capítulo anterior, uma vez que os dados do *Overflow* também devem participar da execução de consultas. Sendo assim, além do planejamento descrito na Seção 4.3.1, a sequência *linear* S_I de uma consulta G_Q também deve ser usada para a geração de um plano para o *Overflow* ($S_{Overflow}$).

A ideia na definição de $S_{Overflow}$ é que todos os vértices de G_Q sejam mapeados para o vértice v_O de G_S. Na detecção de ocorrências de PAs, os passos de S_I devem ser agrupados de acordo com o padrão estrutural adotado na distribuição de dados do *Overflow*. Cada grupo de passos representa uma ocorrência de PA do tipo *PaOverflow* com um identificador único. A origem de cada passo define a sua ocorrência de PA. Por fim, além das condições $C1$–$C4$ apresentadas na Seção 4.3.1, a validação de $S_{Overflow}$ também deve verificar se as propriedades *inter-PA* do plano existem como arestas recursivas em v_O. Se pelo menos uma propriedade *inter-PA* não existir em v_O, o plano não é validado.

Perceba que os repositórios G_D e *Overflow* consideram diferentes *pontos iniciais de exploração* na execução de consultas. Sendo assim, a questão da recuperação de *pontos iniciais de exploração* no *Overflow* deve ser introduzida no algoritmo de exploração distribuída de grafos apresentado na Seção 4.3.2. Para isso, os índices *IxSubjects* e *IxObjects* também devem ser considerados como parte da condição estabelecida entre as linhas 2 e 8. Lembrando que *IxSubjects* e *IxObjects* viabilizam a recuperação dos *pontos iniciais de exploração* no *Overflow* indexando propriedades e referenciando sujeitos e objetos do repositório, respectivamente. Além disso, a recuperação de vértices locais em cada Escravo (linha 11) deve considerar o tipo de padrão de alocação do passo em execução. Caso o passo em execução possua o tipo *PaOverflow*, o repositório *Overflow* deve ser considerado na recuperação; caso contrário, o repositório G_D. Por fim, a definição das variáveis remotas de junção (linhas 30–35) deve considerar o tipo do padrão de alocação do passo seguinte. Caso o padrão seguinte seja *PaOverflow*, somente vértices remotos do *Overflow* devem ser considerados na junção; caso contrário, somente vértices de G_D. O resultado final da consulta é a união de todos os resultados gerados pelos planos executados.

Considerando a sequência S_I da Figura 4.2(b), a Figura 5.5 exibe o plano de execução elaborado para o *Overflow*. Neste exemplo, os Escravos recebem os planos da Figura 4.6 e da Figura 5.5. Observe que todos os passos do segundo plano consideram o padrão de alocação *PaOverflow* e que eles estão agrupados em blocos de acordo com o modelo de distribuição adotado no *Overflow*. Nesta consulta, cada Escravo detecta os *pontos iniciais de exploração* por meio do índice *IxSubjects*. A detecção de vértices remotos é feita por meio da função *hash* usada na distribuição de dados do *Overflow*.

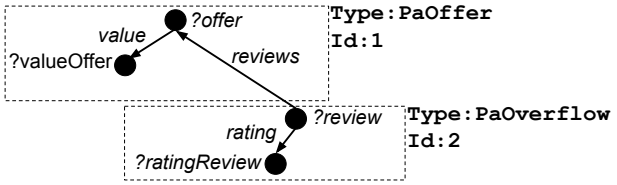
Em adição, considerando que os dados envolvidos nos padrões de tripla de uma consulta podem estar distribuídos entre G_D e o *Overflow*, um único plano pode considerar PAs de G_S *inicial* e o *PaOverflow*. Sendo assim, dada uma sequência *linear* S_I de uma consulta G_Q, o Algoritmo 2 da Seção 4.3.1 deve

$S_{Overflow}$ = [
s_1: ((*?product*, name, *?nameProduct*), *out*, PaOverflow, 1, { }),
s_2: ((*?offer*, offer, *?product*), *in*, PaOverflow, 1, { }),
s_3: ((*?offer*, value, *?offerValue*), *out*, PaOverflow, 2, {(?valueOffer<60000)}),
s_4: ((*?offer*, is_bought, *?person*), *out*, PaOverflow, 2, { }),
s_5: ((*?person*, name, *?namePerson*), *out*, *PaOverflow*, 3, { }),
s_6: ((*?person*, friendOf, *?friend*), *out*, *PaOverflow*, 3, { }),
s_7: ((*?friend*, name, *?nameFriend*), *out*, *PaOverflow*, 4, { })
]

Figura 5.5: Plano de execução para o *Overflow*

```
SELECT ?valueOffer, ?ratingReview
WHERE {
  ?offer value ?valueOffer .
  ?review reviews ?offer .
  ?review rating ?ratingReview.
}
```

(a)



(b)

Figura 5.6: Consulta com PAs de G_S *inicial* e o *PaOverflow* (a); Ocorrências de PAs da consulta (b)

S = [
s_1: ((*?offer*, value, *?valueOffer*), *out*, *PaOffer*, 1, { }),
s_2: ((*?review*, reviews, *?offer*), in, *PaOffer*, 1, { }),
s_3: ((*?review*, rating, *?ratingReview*), *out*, *PaOverflow*, 2, { }),\
]

(a)

$S_{Overflow}$ = [
s_1: ((*?offer*, value, *?offerValue*), *out*, PaOverflow, 1, { }),
s_2: ((*?review*, reviews, *?offer*), in, *PaOverflow*, 1, { }),
s_3: ((*?review*, rating, *?ratingReview*), *out*, *PaOverflow*, 2, { })
]

(b)

Figura 5.7: Planos de execução da consulta da Figura 5.6

considerar o padrão estrutural usado na distribuição de dados do *Overflow* sempre que um vértice v de G_Q for mapeado para v_O. Desta forma, vértices adjacentes à v em G_Q que compõem o padrão estrutural usado no *Overflow* também devem ser mapeados para v_O. Além disso, a origem do primeiro passo de S_I também deve ser mapeada para v_O durante a definição dos mapeamentos. Por fim, o padrão estrutural dos fragmentos do *Overflow* também deve ser considerado na detecção de ocorrências de PAs para os passos com origens mapeadas para v_O.

A consulta da Figura 5.6(a) ilustra um exemplo onde dados de revisões de ofertas de G_D estão armazenados no *Overflow* (Figura 5.6(b)). O planejamento considerou o grafo de sumarização da Figura 5.2(b). Neste caso, enquanto o primeiro plano da consulta (Figura 5.7(a)) envolve o *PaOffer* e o *PaOverflow* em seus passos, o segundo plano (Figura 5.7(b)) considera somente o *PaOverflow*. Na execução do primeiro plano, a exploração de grafos caminha de G_D (passos s_1 e s_2) para o *Overflow* (passo s_3). Em contrapartida, o segundo plano explora somente o *Overflow*. A Figura 5.8 mostra os resultados gerados por ambos os planos, onde m_1 e m_2 são gerados pelo primeiro plano e m_3 pelo segundo plano.

Por fim, dado que o tamanho médio dos fragmentos de dados do *Overflow* não é conhecido previamente pelo processador de consultas, a função de custo da estratégia de comunicação *get-frag* passa a considerar para o *PaOverflow* o tamanho médio de todos os fragmentos alocados em G_D pelo *módulo de alocação*.

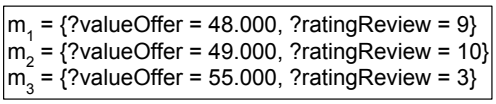


Figura 5.8: Resultados da consulta da Figura 5.6

Considerações

A abordagem de atualização de dados associada à extensão da abordagem de processamento apresentada neste capítulo possibilita a adoção do método de comunicação *2ways* no processamento de consultas SPARQL em bases RDF dinâmicas e distribuídas. A abordagem de atualização propõe situações de extensão para o grafo de sumaração G_S e de armazenamento respeitando a abordagem de processamento proposta no capítulo anterior. Dados categorizáveis por meio de PAs de G_S *inicial* são armazenados no repositório G_D da arquitetura da Figura 5.1. Em contrapartida, dados não-categorizáveis em G_S *inicial* são armazenados no repositório *Overflow*. Consequentemente, ambos os repositórios devem ser considerados no processamento de consultas.

Perceba que a execução dos planos de uma consulta pode ser otimizada. Uma alternativa é o compartilhamento de processamento entre planos. Isso pode ser feito, por exemplo, quando o conjunto de passos de uma ocorrência de PA existir em dois ou mais planos de uma consulta e esta ocorrência ter prioridade na ordenação dos passos dos planos. Assim, a ocorrência pode ser executada uma única vez e compartilhar os seus resultados com a segunda ocorrência de PAs de todos os planos. Após o compartilhamento, cada plano continuaria com a sua execução normalmente. Este tipo de otimização implicaria tanto no custo de processamento quanto de comunicação. Porém, este tipo de otimização não é tratada neste trabalho.

CAPÍTULO 6

ESTUDOS EXPERIMENTAIS

Primeiramente foi conduzido um estudo experimental para determinar o efeito das estratégias de comunicação no desempenho e na escalabilidade do processamento de consultas SPARQL considerando um processador de exploração distribuída de grafos, conforme descrito no Capítulo 4. Em particular, as estratégias *send-result*, *get-frag* e *2ways* foram comparadas, onde *send-result* representa as estratégias de comunicação usadas em (HUANG; ABADI; REN, 2011) e *SHAPE* (LEE; LIU, 2013), e *get-frag* representa a estratégia usada em *MAPSIN* (PRZYJACIEL-ZABLOCKI et al., 2012). Por fim, foi analisado o efeito da abordagem de armazenamento proposta no Capítulo 5 no desempenho da abordagem de processamento de consultas, dado que fragmentos de dados podem estar alocados em G_D e/ou no *Overflow*.

6.1 Configuração

A abordagem de processamento proposta neste trabalho foi implementada em Java adotando o seu modelo de comunicação nativo RMI. O sistema foi implantado em um *cluster* de instâncias *Amazon* EC2 m4.large dedicadas, cada uma com 8 GB de memória e 2 CPUs virtuais de 24 GHz. Para avaliar a escalabilidade dos métodos de comunicação, três *clusters* diferentes foram construídos variando o número de instâncias EC2 em: 4 ($C1$), 8 ($C2$) and 12 ($C3$) servidores. O repositório *Berkeley DB*[1] foi usado para o armazenamento de dados. Para os experimentos, o *Berkeley DB* foi implantado como um repositório baseado em memória sobre os *clusters* EC2.

As bases de dados e as consultas aplicadas no estudo foram extraídas do *benchmark Berlin* (*BSBM*) Bizer e Schultz (2009), o qual considera um caso de uso no contexto de comércio eletrônico modelando o esquema de sua base por meio de relacionamentos entre produtos, características de produtos, produtores, ofertas, vendedores e revisões de produto. O *BSBM* fornece um conjunto de consultas e um gerador de dados que usa o número de produtos com fator de escala. Foram geradas três bases de dados para os experimentos, sendo elas: com 5.000 produtos e 1.811.316 triplas ($BSBM\_1$), com 10.000 produtos e 3.567.636 triplas ($BSBM\_2$) e com 15.000 produtos e 5.323.644 triplas ($BSBM\_3$). Os fragmentos de dados de cada base foram gerados conforme o Capítulo 4 e distribuídos uniformemente no *cluster*.

Também foi considerada uma carga de trabalho com dez consultas ($Q1$–$Q10$). Neste conjunto, $Q1$–$Q5$ são baseadas no *BSBM*. As consultas $Q6$–$Q10$ foram elaboradas para tratar diferentes situações de comunicação a fim de de avaliar de maneira apropriada as estratégias de comunicação. A seguir, alguns resultados experimentais são apresentados. As definições das consultas e os resultados estão no Anexo A.

[1] http://www.oracle.com/technetwork/database/database-technologies/berkeleydb

6.2 Método de comunicação *2ways*

Primeiramente o tempo de resposta das consultas foi analisado comparando as três estratégias de comunicação. Em seguida, o *2ways* foi analisado considerando a sua escalabilidade de dados e de servidores.

6.2.1 Desempenho

O propósito deste experimento é determinar o tempo médio de resposta das consultas $Q1$–$Q10$. Para isso, as consultas foram processadas usando somente as estratégias *send-result (SR)* e *get-frag (GF)*, e o desempenho das consultas foi comparado com o desempenho do método *2ways*. A Tabela 6.1 apresenta o tempo médio de resposta das consultas em milissegundos, considerando a base $BSBM\_2$ no *cluster* $C3$. Cada consulta envolve um ($Q1$), dois ($Q2, Q4, Q7, Q9$), ou três PAs ($Q3, Q5, Q6, Q8, Q10$) e a ordem de exploração definida no plano de consulta segue a ordem em que os PAs são apresentados em cada linha, do PA_1 para o PA_3. As colunas *#fr* e *#rt* referem-se, respectivamente, ao número de fragmentos requeridos para o processamento do próximo PA e ao número de resultados intermediários gerados depois do processamento do PA corrente. Cada coluna apresenta a *soma* do número de mensagens (NM) *para todos os servidores* das funções de custo apresentadas na Seção 4.3.2 para *get-frag* e *send-result*, respectivamente.

A Tabela 6.1 mostra que para algumas consultas *send-result* apresenta um melhor desempenho quando comparado com *get-frag*, e para outras consultas *get-frag* apresenta um melhor desempenho. O desempenho de *2ways* acompanha o da melhor estratégia de comunicação, e para alguns casos o desempenho de *2ways* é melhor, como em $Q6$. $Q1$ é processada localmente dado que somente um PA é requerido. Logo, todos os modelos de comunicação apresentam o mesmo tempo de resposta. Para as consultas $Q2$, $Q3$, $Q5$, $Q7$ e $Q8$, *send-result* foi a melhor estratégia para todas as transições de PAs, logo o tempo de resposta de *2ways* é similar ao de *send-result*. Para a maioria das consultas, esta escolha é feita principalmente porque o número de resultados intermediários depois do processamento do PA corrente é menor do que o número de fragmentos requeridos do próximo PA da consulta. Por exemplo, para $Q2$ o processador de consultas pode enviar 1 resultado intermediário computado do fragmento *PaProduct* ou requisitar 17 fragmentos de *PaFeature* de servidores remotos. Logo, o processador escolhe a estratégia *send-result*. Para $Q3$, o número de transmissões é o mesmo para ambas as estratégias, porém o tamanho do resultado intermediário é menor do que o número de fragmentos a ser recuperado. Para $Q5$, os resultados de 14 resultados intermediários de *PaOffer* foram enviados entre os servidores ao invés da recuperação de 2 fragmentos de *PaVendor*. Dado que a escolha é feita localmente em cada servidor, a estratégia *send-result* foi escolhida neste conjunto de consultas.

Para as consultas $Q4$, $Q9$ e $Q10$, a estratégia *get-frag* apresentou melhor desempenho em todas as transições de PAs, logo o tempo de resposta do *2ways* é similar ao de *get-frag*. Já na consulta $Q6$, *2ways*

Tabela 6.1: PAs e tempo de resposta das três estratégias de comunicação em $BSBM\_2 - C3$

Q	PA_1			PA_2			PA_3			*Tempo de resposta (ms)*		
	padrão	*#fg*	*#rt*	*padrão*	*#fg*	*#rt*	*padrão*	*#fg*	*#rt*	*SR*	*GF*	*2ways*
Q1	*PaProduct*	10k	1	-	-	-	-	-	-	110	110	110
Q2	*PaProduct*	10k	1	*PaFeature*	17	17	-	-	-	134	161	133
Q3	*PaOffer*	200k	1	*PaProduct*	1	1	*PaProducer*	1	1	1003	1027	980
Q4	*PaReview*	100k	7036	*PaProduct*	2	7036	-	-	-	4068	1427	1430
Q5	*PaProduct*	10k	1	*PaOffer*	14	14	*PaVendor*	2	14	143	160	144
Q6	*PaVendor*	508	213	*PaOffer*	80157	80157	*PaProduct*	4	80157	15002	24917	9826
Q7	*PaProducer*	206	206	*PaProduct*	10k	10k	-	-	-	1231	2748	1230
Q8	*PaProducer*	206	206	*PaProduct*	10k	10k	*PaOffer*	200k	14330	2919	31986	2920
Q9	*PaProduct*	10k	10k	*PaProducer*	206	10k	-	-	-	1755	1618	1620
Q10	*PaOffer*	200k	14330	*PaProduct*	2	14330	*PaProducer*	2	14330	11298	2849	2853

escolheu *send-result* do PA_1 para o PA_2 e *get-frag* do PA_2 para o PA_3. A escolha mista de estratégias resultou em um tempo de resposta 35% menor do que *send-result* e 61% menor do que *get-frag*.

Os resultados mostram o seguinte. Primeiro, pode existir uma diferença de várias ordens de magnitude entre o tempo de resposta de *send-result* e de *get-frag* tais como as consultas $Q8$ e $Q4$. Logo, *2ways* apresenta um melhor desempenho do que os sistemas que usam somente uma estratégia de comunicação para todas as consultas, tais como *SHAPE* (LEE; LIU, 2013) e *MAPSIN* (PRZYJACIEL-ZABLOCKI et al., 2012). Segundo, as funções de custo de *2ways* viabilizam a escolha correta da melhor estratégia de comunicação para toda transição entre PAs de $Q1$–$Q10$. Também pode-se concluir que *2ways* é um método de otimização de consultas efetivo e eficiente.

Por fim, vale a pena destacar que os planos de execução das consultas $Q1$–$Q8$ foram "otimizados"dado que os seus PAs estão ordenados de acordo com os seus fatores de seletividade a fim de reduzir o número de resultados intermediários trocados entre servidores. Esta otimização não foi aplicada nas duas últimas consultas. De fato, $Q9$ e $Q10$ executam as mesmas consultas que $Q7$ e $Q8$, respectivamente, mas com uma diferente ordem (PA) de exploração. Comparando os tempos de respostas destas consultas, nota-se que *send-result* apresentou um melhor desempenho nos planos "otimizados", porém o mesmo não acontece para *get-frag*. Para esta estratégia, os planos "não-otimizados" foram 59% e 11 vezes mais rápidos. Entretanto, o método *2ways* tem tempo de resposta similar em ambos os planos. Isto mostra que a técnica de otimização de consultas proposta neste trabalho é ortogonal ao planejamento de consultas.

Considerando experimentos com diferentes tamanhos de *cluster* e base de dados, a estratégia de comunicação aplicada por *2ways* é similar aos resultados reportados na Tabela 6.1. Entretanto, $Q9$ teve a estratégia de comunicação predominante trocada em $BSBM\_3 - C2$ e $BSBM\_3 - C3$. A estratégia *send-result* apresentou melhor desempenho nestes censários e, consequentemente, *2ways* escolheu esta estratégia. A mudança ocorreu dado que os fragmentos de *PaProducer* era maior (em *bytes*) nestes cenários, o que aumentou o custo da estratégia *get-frag*.

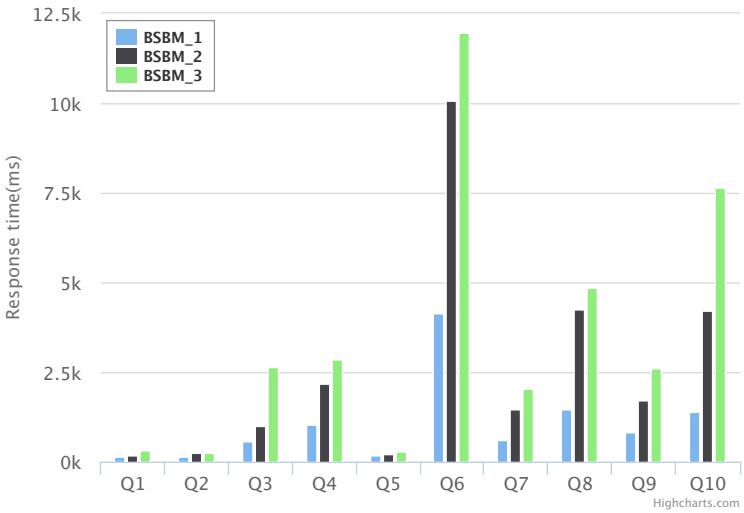


Figura 6.1: Escalabilidade de dados do uso do método *2ways* no *cluster* C2

6.2.2 Escalabilidade

A escalabilidade de *2ways* é avaliada escalando o tamanho de bases de dados e o número de servidores em um *cluster*. A Figura 6.1 mostra a escalabilidade de *2ways* executando as consultas $Q1$–$Q10$ no *cluster* $C2$ nas bases de dados $BSBM\_1$, $BSBM\_2$ e $BSBM\_3$. Como esperado, o tempo de resposta das consultas aumenta de acordo com o tamanho das bases de dados, dado que o tamanho dos dados de entrada usualmente determina o número de resultados intermediários e dados trocados entre os servidores. Além disso, este aumento também implica no tempo de processamento local de fragmentos de dados nos servidores. Isso fica mais evidente nas consultas que envolvem grandes volumes de dados, tais como $Q3$, $Q6$, $Q8$ e $Q10$. Em $Q3$, por exemplo, o número de mensagens trocadas entre servidores é o mesmo nas três bases de dados. Porém, a diferença reportada nos resultados vem do número de fragmentos de *PaOffer* processados: 100, 200 e 300 mil em $BSBM\_1$, $BSBM\_2$ e $BSBM\_3$, respectivamente.

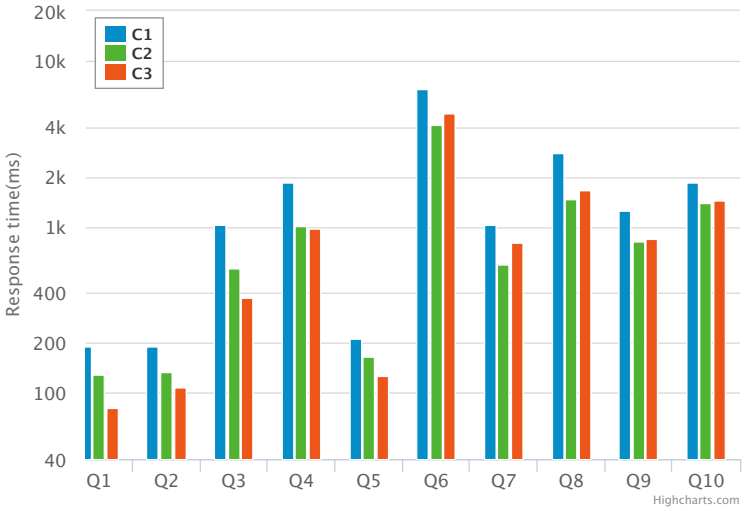
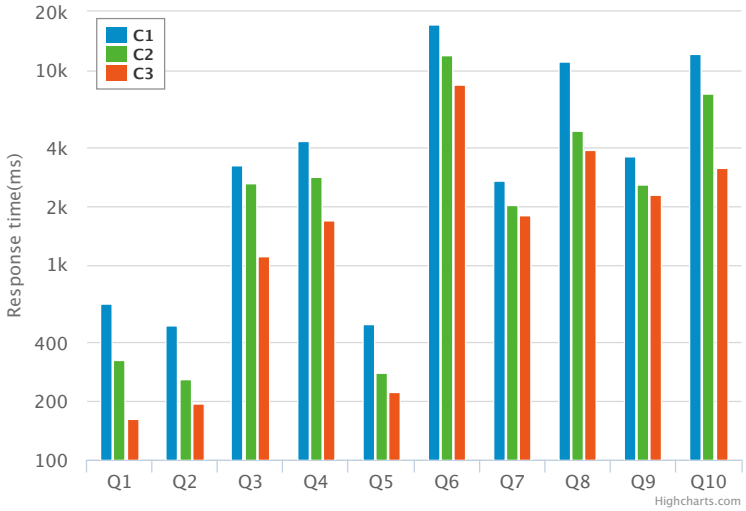


Figura 6.2: Escalabilidade de servidores com o uso do método *2ways* em $BSBM\_1$



(a) Escalabilidade de servidores - $BSBM\_3$

Figura 6.3: Escalabilidade de servidores com o uso do método *2ways* em $BSBM\_3$

O desempenho das consultas também foi analisado variando o número de servidores no *cluster*. Os resultados são exibidos nas Figuras 6.2 e 6.3 nas bases $BSBM\_1$ e $BSBM\_3$, respectivamente. A base $BSBM\_2$ comportou-se como a $BSBM\_3$. Note que conforme o número de servidores aumenta, o tempo de resposta diminui nas bases de dados. Entretanto, o aumento do número de servidores para 12 ($C3$) nos experimentos não tem o mesmo efeito para a maioria das consultas em $BSBM\_1$ e para algumas consultas em $BSBM\_3$. Isso mostra que a distribuição de dados e a execução paralela beneficiam o processamento de consultas até um determinado ponto no qual o custo de comunicação entre os servidores supera o ganho com o paralelismo. Nos experimentos realizados, este ponto foi alcançado com 8 servidores em $BSBM\_1$ e 12 servidores em $BSBM\_3$. Sendo assim, para a carga de trabalho usado neste experimento, a alocação de mais servidores diminuiria o desempenho das consultas.

6.3 Processamento de consultas com o *Overflow*

A adoção da abordagem de atualização apresentada no capítulo anterior pode afetar tanto o custo de processamento quanto o custo de comunicação na execução de consultas. O custo de processamento refere-se: *i)* à recuperação de *pontos iniciais* de exploração, dado que os fragmentos de dados de G_D são categorizados por meio de PAs, e que o mesmo não acontece com os fragmentos de dados do *Overflow*; e, *ii)* à execução do plano de execução para o *Overflow*. Já, o custo de comunicação pode sofrer alterações caso o *Overflow* considere um modelo de distribuição diferente do usado em G_D.

A fim de analisar a influência da abordagem de atualização nos resultados da abordagem de processamento de consultas, a carga de consultas $Q1$–$Q10$ foi submetida ao cenário $BSBM\_1 - C1$ onde a alocação de dados foi simulada de três maneiras, sendo elas: *D1)* todos os dados da aplicação em G_D; *D2)* 50% dos dados em G_D e 50% dos dados no *Overflow*; e, *D3)* todos os dados no *Overflow*. As consultas

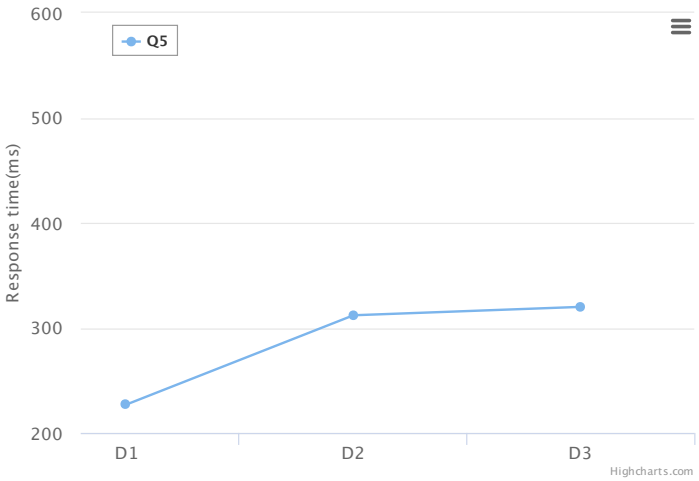


Figura 6.4: Escalabilidade da consulta Q5 com a adoção do *Overflow* em $BSBM\_1–C1$

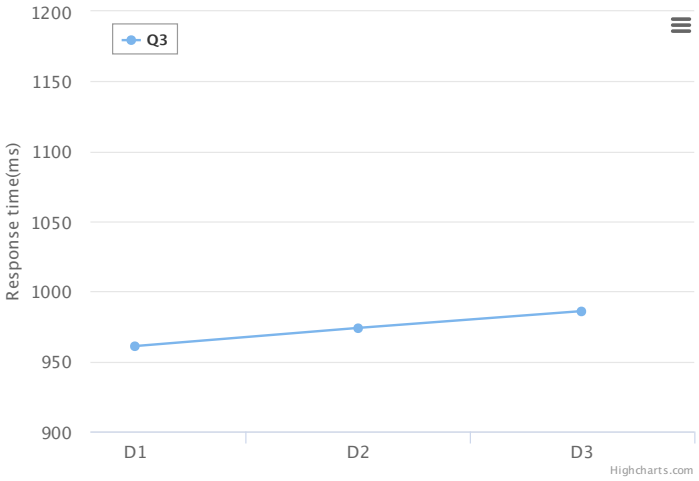


Figura 6.5: Escalabilidade da consulta Q3 com a adoção do *Overflow* em $BSBM\_1–C1$

$Q1$–$Q10$ foram divididas em dois grupos, de acordo com os volumes do espaço inicial de pesquisa de G_D e do *Overflow* usados no experimento. Vale destacar que o espaço inicial de pesquisa de cada consulta contou com todas os recursos dos fragmentos de dados dos tipos de PAs envolvidos no seu processamento, independente do tipo de repositório explorado. No primeiro grupo, o volume do espaço inicial do *Overflow* é maior do que o de G_D dado o tipo de indexação usado nos repositórios. Enquanto G_D considera vértices de um tipo de *PA*, o *Overflow* considera vértices de três tipos de *PAs*. Neste grupo estão as consultas $Q1$, $Q2$, $Q5$, $Q7$, $Q8$ e $Q9$. No segundo grupo, o espaço inicial de ambos os repositórios é igual dado que todos os planos das consultas iniciam o processamento usando somente um tipo de PA. Neste grupo estão as consultas $Q3$, $Q4$, $Q6$ e $Q10$. Considerando que as consultas de um mesmo grupo comportaram-se de maneira similar durante os experimentos, $Q5$ representa as consultas do primeiro grupo na Figura 6.4 e $Q3$ as do segundo grupo na Figura 6.5.

Analisando o custo de processamento, o tempo médio das respostas de $Q3$ e $Q5$ cresce conforme há mais dados no *Overflow*, como esperado. Perceba que a taxa de crescimento do tempo médio de execução

das consultas entre $D1$, $D2$ e $D3$ não segue o mesmo padrão, dado que o espaço inicial de pesquisa de $Q5$ envolve uma maior carga de dados. Além disso, enquanto $D1$ exige um plano para G_D e $D3$ um plano para o *Overflow*, $D2$ exige o processamento de um plano para G_D e outro para o *Overflow*.

Por fim, considerando que a distribuição de dados em G_D e no *Overflow* adota o mesmo padrão estrutural nos experimentos realizados, o custo de comunicação não implica no processamento de consultas de ambos os repositórios. Entretanto, é importante ressaltar que os PAs de G_S *inicial* podem assumir qualquer tipo de composição e que, inclusive, as composições destes PAs podem variar entre si. Sendo assim, experimentos futuros devem analisar a questão da granularidade dos PAs no custo de processamento e de comunicação da abordagem de processamento proposta.

CAPÍTULO 7

CONCLUSÃO

Este trabalho propõe um método de comunicação denominada de *2ways* que tem o objetivo de reduzir o custo de comunicação no processamento de consultas SPARQL em bases RDF distribuídas. Assume-se que fragmentos de dados distribuídos entre servidores são gerados a partir de padrões de alocação (PAs), baseados em estruturas RDF, definidos por um método de distribuição *controlada* de dados. Desta forma, fragmentos de dados podem ser indexados por meio de PAs e informações sobre a composição de PAs usadas no planejamento de consultas. Consequentemente, durante a execução dos planos nos servidores, dados de um mesmo fragmento são explorados localmente em bloco antes da exploração considerar outro fragmento de dado, que pode estar alocado remotamente. Sempre que um dado de um fragmento remoto é requerido, *2ways* viabiliza aos servidores a escolha entre duas estratégias de comunicação, *send-result* e *get-frag*. A decisão de *2ways* é baseada em funções de custo que consideram o número de mensagens e o volume de dados a serem transmitidos entre servidores. A estratégia com o menor custo estimado é escolhida para a comunicação. A análise experimental da abordagem de processamento de consultas mostrou que *2ways* pode efetivamente e eficientemente reduzir o tempo de respostas de consultas quando comparado com as estratégias *send-result* e *get-frag* usadas de maneira isolada.

Além da abordagem de processamento de consultas, este trabalho também propõe uma abordagem de atualização de dados visando dar suporte à abordagem de processamento, dado que ela baseia-se em estruturas RDF que podem tornar-se inadequadas conforme novas triplas são inseridas em uma aplicação. A abordagem considera as operações de exclusão e de inserção de dados. A operação de exclusão simplesmente elimina dados de repositórios não implicando em alterações em estruturas RDF. Já, a operação de inserção estende estruturas RDF a fim de categorizar novas triplas em uma dada aplicação. Basicamente, dados que não são categorizados por meio da estrutura RDF da aplicação são armazenados por meio de um tipo especial de padrão de alocação, o *PaOverflow*. Consequentemente, a abordagem de processamento de consultas citada anteriormente exige alguns ajustes para que o *PaOverflow* seja considerado tanto no planejamento quanto na execução de consultas. O custo de processamento de consultas considerando a abordagem de processamento estendida foi analisado. Resultados experimentais mostram que, como esperado, a inserção de dados não-categorizáveis em uma dada aplicação pode implicar no aumento do custo de processamento da abordagem. Já, o custo de comunicação não foi afetado nos experimentos, dado que ambos os repositórios adotaram o mesmo padrão estrutural na fragmentação.

Este trabalho oferece contribuições no contexto de grandes bases de dados, desde que a estratégia de comunicação proposta pode ser explorada por sistemas de gerenciamento de grandes bases de dados

RDF a fim de fornecer escalabilidade no processamento de consultas. Dado que grande parte do custo envolvido no processamento distribuído de consultas SPARQL resulta do custo de comunicação para a obtenção de dados entre diferentes servidores, diferentes técnicas de otimização têm sido propostas com o propósito de minimizar esse custo. As técnicas variam entre métodos de fragmentação e alocação de dados, além de otimização no planejamento de consultas. Este trabalho propõe um modelo *híbrido* de comunicação como um método de otimização para consultas SPARQL em bases RDF distribuídas. Ressaltando que a técnica de otimização proposta tem como pré-requisito que métodos de fragmentação usem padrões pré-definidos no particionamento de bases. De acordo com o nosso conhecimento, nenhum outro trabalho na literatura adota um modelo *híbrido* de comunicação para a otimização de um modelo de execução de consultas paralelo e distribuído. Em adição, este trabalho também contribui no contexto de gerenciamento de dados em sistemas RDF, dado que a abordagem de atualização proposta trata do dinamismo de estruturas RDF. A abordagem de atualização viabiliza a manipulação (armazenamento e recuperação) de quaisquer triplas de uma aplicação, independentemente dos padrões de alocação (PAs) existentes em sua estrutura RDF.

Trabalhos futuros incluem estudos sobre a ordenação e o sincronismo de ocorrências de PAs no planejamento de consultas a fim de otimizar o processamento de consultas da abordagem proposta. Além disso, uma análise sobre a indexação de dados também pode potencializar a redução do espaço inicial de pesquisa que envolve fragmentos de dados categorizados. Continuando, faz-se necessário um estudo sobre a calibração dos custos dependentes de hardware, de rede e de repositório relacionados às funções de custo do *2ways*. Desta forma, os resultados das funções podem acompanhar variações do ambiente de execução. Por fim, a efetividade da abordagem de atualização deve ser analisada de maneira detalhada, dado que a execução de operações de exclusão e de inserção envolvem custos de processamento e de comunicação entres servidores. Além disso, experimentos futuros devem analisar a questão da granularidade dos PAs no custo de processamento e de comunicação da abordagem de processamento proposta.

PUBLICAÇÕES REALIZADAS NO DOUTORADO

A lista a seguir registra os artigos publicados durante o período deste doutorado e referentes ao tema desta tese.

- R. Schroeder, R. R. M. Penteado e C. S. Hara. Partitioning RDF Exploiting Workload Information. International Conference on World Wide Web Companion (WWW), Rio de Janeiro, RJ, Brasil, 2013.
- R. R. M. Penteado, R. Schroeder, D. Hoss, J. Nande, R. M. Maeda, W. O. Couto e C. S. Hara. Um Estudo sobre Bancos de Dados em Grafos Nativos. X Escola Regional de Banco de Dados (ERBD), São Francisco do Sul, SC, Brasil, 2014.
- R. R. M. Penteado, R. Schroeder e C. S. Hara. Exploração de Grafos em Bases RDF com Distribuição Controlada. XXX Simpósio Brasileiro de Banco de Dados, Petrópolis, RJ, Brasil, 2015.
- R. R. M. Penteado, R. Schroeder e C. S. Hara. Exploring Controlled RDF Distribution. 8th IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Luxemburgo, Luxemburgo, 2016.

ANEXO A

A seguir são apresentadas as dez consultas e todos os resultados gerados pelos seis cenários usados nos experimentos. Cada cenário combina um tamanho de *cluster* ($C1$: 4 servidores, $C2$: 8 servidores e $C3$: 12 servidores) com um tamanho de base ($BSBM\_1$: com 5.000 produtos e 1.811.316 triplas, $BSBM\_2$: com 10.000 produtos e 3.567.636 triplas, e $BSBM\_3$: com 15.000 produtos e 5.323.644 triplas).

Consultas SPARQL

Q1: Encontrar produtos que possuem uma palavra específica no seu rótulo.

SELECT ?labelProduct, ?propertyTextual1

WHERE {

[PA_1] ?product rdfs:label ?labelProduct .

[PA_1] ?product bsbm:productPropertyTextual1 ?propertyTextual1 .

[PA_1] FILTER regex(?labelProduct, % word %)

}

Q2: Recuperar características específicas de um determinado produto.

SELECT ?labelProduct, ?labelFeature

WHERE {

[PA_1] ?product rdfs:label % labelProduct % .

[PA_1] ?product bsbm:productFeature ?feature .

[PA_2] ?feature rdfs:label ?labelFeature .

}

Q3: Recuperar informações sobre uma oferta específica.

SELECT ?deliveryDaysOffer, ?priceOffer, ?labelProducer

WHERE {

[PA_1] ?offer bsbm:offerWebpage % homepageOffer% .

[PA_1] ?offer bsbm:deliveryDays ?deliveryDaysOffer .

[PA_1] ?offer bsbm:price ?priceOffer .

[PA_1] ?offer bsbm:product ?product .

[PA_2] ?product bsbm:producer ?producer.

[PA_3] ?producer rdfs:label ?labelProducer .

}

Q4: Encontrar produtos que possuem revisões com notas maiores que 9.5 .

SELECT ?labelProduct

WHERE {

[PA_1] ?review bsbm:rating1 ?ratingReview .

[PA_1] FILTER (ratingReview > 9.5) .

[PA_1] ?review bsbm:reviewFor ?product .

[PA_2] ?product rdfs:label ?labelProduct .

}

Q5: Encontrar vendedores de um produto específico.

SELECT ?commentProduct, ?labelVendor

WHERE {

[PA_1] ?product rdfs:label % labelProduct % .

[PA_1] ?product rdfs:comment ?commentProduct .

[PA_1] ?offer bsbm:product ?product .

[PA_2] ?offer bsbm:vendor ?vendor.

[PA_3] ?vendor rdfs:label ?labelVendor .

}

Q6: Recuperar produtos vendidos por vendedores americanos.

SELECT ?labelProduct

WHERE {

[PA_1] ?vendor bsbm:country % US % .

[PA_2] ?offer bsbm:vendor ?vendor .

[PA_3] ?offer bsbm:product ?product .

[PA_3] ?product rdfs:label ?labelProduct .

}

Q7: Recuperar produtos e os seus respectivos produtores.

SELECT ?labelProduct, ?commentProduct, ?labelProducer

WHERE {

[PA_1] ?producer rdfs:label ?labelProducer .

[PA_1] ?product bsbm:producer ?producer .

[PA_2] ?product rdfs:label ?labelProduct .

[PA_2] ?product rdfs:comment ?commentProduct .

}

Q8: Encontrar informações sobre ofertas com tempo de entrega igual a 1 dia.

SELECT ?labelProducer, ?labelProduct, ?valueOffer

WHERE {

[PA_1] ?producer rdfs:label ?labelProducer .

[PA_1] ?product bsbm:producer ?producer .

[PA_2] ?product rdfs:label ?labelProduct .

[PA_2] ?offer bsbm:product ?product .

[PA_3] ?offer bsbm:deliveryDays % 1 % .

[PA_3] ?offer bsbm:price ?valueOffer .

}

Q9: Recuperar produtos e seus respectivos produtores.

SELECT ?labelProduct, ?commentProduct, ?labelProducer

WHERE {

[PA_1] ?product rdfs:label ?labelProduct .

[PA_1] ?product rdfs:comment ?commentProduct .

[PA_1] ?product bsbm:producer ?producer .

[PA_2] ?producer rdfs:label ?labelProducer .

}

Q10: Encontrar informações sobre ofertas com tempo de entrega igual a 1 dia.

SELECT ?labelProducer, ?labelProduct, ?valueOffer

WHERE {

[PA_1] ?offer bsbm:deliveryDays % 1 % .

[PA_1] ?offer bsbm:price ?valueOffer .

[PA_1] ?offer bsbm:product ?product .

[PA_2] ?product rdfs:label ?labelProduct .

[PA_2] ?product bsbm:producer ?producer .

[PA_3] ?producer rdfs:label ?labelProducer .

}

Resultados

Tabela 7.1: PAs e tempo de resposta das três estratégias de comunicação em $BSBM\_1$–$C1$

	PA_1			PA_2			PA_3			*Communication*		
Query	*pattern*	*frags*	*result*	*pattern*	*frags*	*result*	*pattern*	*frags*	*result*	*SR*	*GF*	*2ways*
Q1	*PaProduct*	5k	1	-	-	-	-	-	-	192	192	192
Q2	*PaProduct*	5k	1	*PaFeature*	17	17	-	-	-	185	209	190
Q3	*PaOffer*	100k	1	*PaProduct*	1	1	*PaProducer*	1	1	1025	1145	1026
Q4	*PaReview*	50k	3519	*PaProduct*	1	3519	-	-	-	2949	1863	1850
Q5	*PaProduct*	5k	1	*PaOffer*	26	26	*PaVendor*	1	26	208	241	214
Q6	*PaVendor*	254	90	*PaOffer*	41031	41031	*PaProduct*	2	41031	12201	19706	6727
Q7	*PaProducer*	106	106	*PaProduct*	5k	5k	-	-	-	1079	2383	1023
Q8	*PaProducer*	106	106	*PaProduct*	5k	5k	*PaOffer*	100k	7383	2502	24808	2608
Q9	*PaProduct*	5k	5k	*PaProducer*	106	5k	-	-	-	1244	1264	1250
Q10	*PaOffer*	100k	7383	*PaProduct*	1	7383	*PaProducer*	1	7383	6803	1855	1858

Tabela 7.2: PAs e tempo de resposta das três estratégias de comunicação em $BSBM\_1$–$C2$

	PA_1			PA_2			PA_3			*Communication*		
Query	*pattern*	*frags*	*result*	*pattern*	*frags*	*result*	*pattern*	*frags*	*result*	*SR*	*GF*	*2ways*
Q1	*PaProduct*	5k	1	-	-	-	-	-	-	128	128	128
Q2	*PaProduct*	5k	1	*PaFeature*	17	17	-	-	-	138	162	134
Q3	*PaOffer*	100k	1	*PaProduct*	1	1	*PaProducer*	1	1	565	709	568
Q4	*PaReview*	50k	3081	*PaProduct*	1	3081	-	-	-	2096	1020	1021
Q5	*PaProduct*	5k	1	*PaOffer*	26	26	*PaVendor*	1	26	164	188	165
Q6	*PaVendor*	254	90	*PaOffer*	33839	33839	*PaProduct*	2	33839	9965	10959	4133
Q7	*PaProducer*	106	106	*PaProduct*	5k	5k	-	-	-	527	1644	592
Q8	*PaProducer*	106	106	*PaProduct*	5k	5k	*PaOffer*	100k	6198	1382	12944	1473
Q9	*PaProduct*	5k	5k	*PaProducer*	106	5k	-	-	-	981	813	818
Q10	*PaOffer*	100k	6198	*PaProduct*	1	6198	*PaProducer*	1	6198	5059	1385	1395

Tabela 7.3: PAs e tempo de resposta das três estratégias de comunicação em $BSBM\_1$–$C3$

	PA_1			PA_2			PA_3			*Communication*		
Query	*pattern*	*frags*	*result*	*pattern*	*frags*	*result*	*pattern*	*frags*	*result*	*SR*	*GF*	*2ways*
Q1	*PaProduct*	5k	1	-	-	-	-	-	-	82	82	82
Q2	*PaProduct*	5k	1	*PaFeature*	17	17	-	-	-	106	152	108
Q3	*PaOffer*	100k	1	*PaProduct*	1	1	*PaProducer*	1	1	372	405	374
Q4	*PaReview*	50k	3519	*PaProduct*	1	3519	-	-	-	2051	969	969
Q5	*PaProduct*	5k	1	*PaOffer*	26	26	*PaVendor*	1	26	125	166	126
Q6	*PaVendor*	254	90	*PaOffer*	40858	40858	*PaProduct*	2	40858	12110	13719	4873
Q7	*PaProducer*	106	106	*PaProduct*	5k	5k	-	-	-	805	1850	804
Q8	*PaProducer*	106	106	*PaProduct*	5k	5k	*PaOffer*	100k	7082	1571	12093	1670
Q9	*PaProduct*	5k	5k	*PaProducer*	106	5k	-	-	-	1278	845	846
Q10	*PaOffer*	100k	7082	*PaProduct*	1	7082	*PaProducer*	1	7082	6098	1454	1455

Tabela 7.4: PAs e tempo de resposta das três estratégias de comunicação em $BSBM\_2$–$C1$

	PA_1			PA_2			PA_3			*Communication*		
Query	*pattern*	*frags*	*result*	*pattern*	*frags*	*result*	*pattern*	*frags*	*result*	*SR*	*GF*	*2ways*
Q1	*PaProduct*	10k	1	-	-	-	-	-	-	300	300	300
Q2	*PaProduct*	10k	1	*PaFeature*	17	17	-	-	-	327	355	330
Q3	*PaOffer*	200k	1	*PaProduct*	1	1	*PaProducer*	1	1	2744	3010	2755
Q4	*PaReview*	100k	6740	*PaProduct*	2	6740	-	-	-	5786	3092	3064
Q5	*PaProduct*	10k	1	*PaOffer*	14	14	*PaVendor*	2		337	362	342
Q6	*PaVendor*	508	213	*PaOffer*	79580	79580	*PaProduct*	4	79580	17507	30971	12971
Q7	*PaProducer*	206	206	*PaProduct*	10k	10k	-	-	-	1769	4191	1691
Q8	*PaProducer*	206	206	*PaProduct*	10k	10k	*PaOffer*	200k	14068	9975	57178	9980
Q9	*PaProduct*	10k	10k	*PaProducer*	206	10k	-	-	-	2338	2450	2350
Q10	*PaOffer*	200k	14068	*PaProduct*	2	14068	*PaProducer*	2	14068	20659	8933	8935

Tabela 7.5: PAs e tempo de resposta das três estratégias de comunicação em $BSBM\_2$–$C2$

	PA_1			PA_2			PA_3			*Communication*		
Query	*pattern*	*frags*	*result*	*pattern*	*frags*	*result*	*pattern*	*frags*	*result*	*SR*	*GF*	*2ways*
Q1	*PaProduct*	10k	1	-	-	-	-	-	-	161	161	161
Q2	*PaProduct*	10k	1	*PaFeature*	17	17	-	-	-	240	253	241
Q3	*PaOffer*	200k	1	*PaProduct*	1	1	*PaProducer*	1	1	1002	1015	1003
Q4	*PaReview*	100k	7036	*PaProduct*	2	7036	-	-	-	3658	2191	2191
Q5	*PaProduct*	10k	1	*PaOffer*	14	14	*PaVendor*	2		224	266	225
Q6	*PaVendor*	508	213	*PaOffer*	80157	80157	*PaProduct*	4	80157	14993	23804	10075
Q7	*PaProducer*	206	206	*PaProduct*	10k	10k	-	-	-	1469	3296	1475
Q8	*PaProducer*	206	206	*PaProduct*	10k	10k	*PaOffer*	200k	14330	4244	35871	4250
Q9	*PaProduct*	10k	10k	*PaProducer*	206	10k	-	-	-	1878	1705	1712
Q10	*PaOffer*	200k	14330	*PaProduct*	2	14330	*PaProducer*	2	14330	12533	4209	4209

Tabela 7.6: PAs e tempo de resposta das três estratégias de comunicação em $BSBM\_2$–$C3$

	PA_1			PA_2			PA_3			*Communication*		
Query	*pattern*	*frags*	*result*	*pattern*	*frags*	*result*	*pattern*	*frags*	*result*	*SR*	*GF*	*2ways*
Q1	*PaProduct*	10k	1	-	-	-	-	-	-	110	110	110
Q2	*PaProduct*	10k	1	*PaFeature*	17	17	-	-	-	134	161	133
Q3	*PaOffer*	200k	1	*PaProduct*	1	1	*PaProducer*	1	1	1003	1027	980
Q4	*PaReview*	100k	6846	*PaProduct*	2	6846	-	-	-	4068	1427	1430
Q5	*PaProduct*	10k	1	*PaOffer*	14	14	*PaVendor*	2		143	160	144
Q6	*PaVendor*	508	213	*PaOffer*	76047	76047	*PaProduct*	4	76047	15002	24917	9826
Q7	*PaProducer*	206	206	*PaProduct*	10k	10k	-	-	-	1231	2748	1230
Q8	*PaProducer*	206	206	*PaProduct*	10k	10k	*PaOffer*	200k	14499	2919	31986	2920
Q9	*PaProduct*	10k	10k	*PaProducer*	206	10k	-	-	-	1755	1618	1620
Q10	*PaOffer*	200k	14499	*PaProduct*	2	14499	*PaProducer*	2	14499	11298	2849	2853

Tabela 7.7: PAs e tempo de resposta das três estratégias de comunicação em $BSBM\_3$–$C1$

	PA_1			PA_2			PA_3			*Communication*		
Query	*pattern*	*frags*	*result*	*pattern*	*frags*	*result*	*pattern*	*frags*	*result*	*SR*	*GF*	*2ways*
Q1	*PaProduct*	15k	2	-	-	-	-	-	-	630	630	630
Q2	*PaProduct*	15k	1	*PaFeature*	17	17	-	-	-	487	516	492
Q3	*PaOffer*	300k	1	*PaProduct*	1	1	*PaProducer*	1	1	3264	3304	3270
Q4	*PaReview*	150k	9884	*PaProduct*	4	9884	-	-	-	7947	4319	4329
Q5	*PaProduct*	15k	1	*PaOffer*	14	14	*PaVendor*	4	14	509	512	499
Q6	*PaVendor*	759	282	*PaOffer*	111602	111602	*PaProduct*	8	111602	20537	52235	17243
Q7	*PaProducer*	308	308	*PaProduct*	15k	15k	-	-	-	2725	7016	2703
Q8	*PaProducer*	308	308	*PaProduct*	15k	15k	*PaOffer*	300k	21567	11101	84105	11100
Q9	*PaProduct*	15k	15k	*PaProducer*	308	15k	-	-	-	3591	3586	3598
Q10	*PaOffer*	300k	21567	*PaProduct*	4	21567	*PaProducer*	4	21567	21316	12106	12105

Tabela 7.8: PAs e tempo de resposta das três estratégias de comunicação em $BSBM\_3$–$C2$

	PA_1			PA_2			PA_3			*Communication*		
Query	*pattern*	*frags*	*result*	*pattern*	*frags*	*result*	*pattern*	*frags*	*result*	*SR*	*GF*	*2ways*
Q1	*PaProduct*	15k	2	-	-	-	-	-	-	326	326	326
Q2	*PaProduct*	15k	1	*PaFeature*	17	17	-	-	-	257	292	258
Q3	*PaOffer*	300k	1	*PaProduct*	1	1	*PaProducer*	1	1	2592	1651	2650
Q4	*PaReview*	150k	10085	*PaProduct*	4	10085	-	-	-	6248	2836	2850
Q5	*PaProduct*	15k	1	*PaOffer*	14	14	*PaVendor*	4	14	272	299	281
Q6	*PaVendor*	759	282	*PaOffer*	78133	78133	*PaProduct*	8	78133	13823	28097	11980
Q7	*PaProducer*	308	308	*PaProduct*	15k	15k	-	-	-	1939	5359	2027
Q8	*PaProducer*	308	308	*PaProduct*	15k	15k	*PaOffer*	300k	19394	4840	62111	4871
Q9	*PaProduct*	15k	15k	*PaProducer*	308	15k	-	-	-	2543	2673	2598
Q10	*PaOffer*	300k	19394	*PaProduct*	4	19394	*PaProducer*	4	19394	18359	7637	7639

** A estratégia *send-result* (SR) predominou em $Q9$ neste cenário.

Tabela 7.9: PAs e tempo de resposta das três estratégias de comunicação em $BSBM\_3$–$C3$

	PA_1			PA_2			PA_3			*Communication*		
Query	*pattern*	*frags*	*result*	*pattern*	*frags*	*result*	*pattern*	*frags*	*result*	*SR*	*GF*	*2ways*
Q1	*PaProduct*	15k	4	-	-	-	-	-	-	162	162	162
Q2	*PaProduct*	15k	1	*PaFeature*	17	17	-	-	-	193	219	195
Q3	*PaOffer*	300k	1	*PaProduct*	1	1	*PaProducer*	1	1	1099	1126	1105
Q4	*PaReview*	150k	6080	*PaProduct*	4	6080	-	-	-	3807	1674	1684
Q5	*PaProduct*	15k	1	*PaOffer*	8	8	*PaVendor*	4	14	248	336	222
Q6	*PaVendor*	759	282	*PaOffer*	65411	65411	*PaProduct*	8	65411	9268	23253	8396
Q7	*PaProducer*	308	308	*PaProduct*	15k	15k	-	-	-	1795	3618	1798
Q8	*PaProducer*	308	308	*PaProduct*	15k	15k	*PaOffer*	300k	12581	3901	32621	3901
Q9	*PaProduct*	15k	15k	*PaProducer*	308	15k	-	-	-	2283	2375	2291
Q10	*PaOffer*	300k	12581	*PaProduct*	4	12581	*PaProducer*	4	12581	8914	3129	3133

** A estratégia *send-result* (SR) predominou em $Q9$ neste cenário.

BIBLIOGRAFIA

ABADI, D. J. et al. SW-Store: A Vertically Partitioned DBMS for Semantic Web Data Management. *The VLDB Journal*, v. 18, n. 2, p. 385–406, 2009.

ABITEBOUL, S.; HULL, R.; VIANU, V. (Ed.). *Foundations of Databases: The Logical Level*. 1. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

ABOUZEID, A. et al. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *Proc. VLDB Endow.*, VLDB Endowment, v. 2, n. 1, p. 922–933, 2009.

ALUÇ, G. et al. *chameleon-db: a Workload-Aware Robust RDF Data Management System*. [S.l.], 2013.

ARENAS, M.; GUTIERREZ, C.; PÉREZ, J. On the Semantics of SPARQL. In: VIRGILIO, R. D.; GIUNCHIGLIA, F.; TANCA, L. (Ed.). *Semantic Web Information Management – A Model Based Perspective*. [S.l.]: Springer, 2009. cap. 13, p. 280–307.

BIZER, C.; SCHULTZ, A. The Berlin SPARQL Benchmark. *International Journal on Semantic Web and Information Systems*, v. 5, n. 2, p. 1–24, 2009.

CHIRKOVA, R.; FLETCHER, G. H. L. Towards Well-Behaved Schema Evolution. In: *12th International Workshop on the Web and Databases, WebDB 2009, Providence, Rhode Island, USA, June 28, 2009*. [S.l.: s.n.], 2009.

CORDELLA, L. P. et al. An Improved Algorithm for Matching Large Graphs. In: *3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition, Cuen*. [S.l.: s.n.], 2001. p. 149–159.

DEAN, J.; GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM - 50th Anniversary Issue: 1958 - 2008*, v. 51, n. 1, p. 107–113, 2008.

DEUTSCH, A.; FERNANDEZ, M.; SUCIU, D. Storing Semistructured Data with STORED. In: *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 1999. (SIGMOD '99), p. 431–442.

FLOURIS, G. et al. Formal Foundations for RDF/S KB Evolution. *Knowl. Inf. Syst.*, v. 35, n. 1, p. 153–191, 2013.

GAI, L.; CHEN, W.; WANG, T. A partition-based summary-graph-driven method for efficient RDF query processing. *CoRR*, abs/1510.07749, 2015.

GALÁRRAGA, L.; HOSE, K.; SCHENKEL, R. Partout: A Distributed Engine for Efficient RDF Processing. *CoRR*, abs/1212.5636, 2012.

GIMéNEZ-GARCIA, J. M.; FERNáNDEZ, J. D.; MARTíNEZ-PRIETO, M. A. MapReduce-based Solutions for Scalable SPARQL Querying. *Open Journal of Semantic Web (OJSW)*, v. 1, n. 1, p. 1–18, 2014.

GOASDOUÉ, F. et al. CliqueSquare: efficient Hadoop-based RDF query processing. In: *BDA'13 - Journées de Bases de Données Avancées*. [S.l.: s.n.], 2013.

GONZALEZ, J. E. et al. PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In: *10th USENIX Conference on Operating Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2012. p. 17–30.

GONZALEZ, J. E. et al. Graphx: Graph processing in a distributed dataflow framework. In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2014. (OSDI'14), p. 599–613.

GURAJADA, S. et al. TriAD: A Distributed Shared-nothing RDF Engine Based on Asynchronous Message Passing. In: *ACM SIGMOD International Conference on Management of Data*. New York, USA: ACM, 2014. p. 289–300.

GUTIERREZ, C.; HURTADO, C.; VAISMAN, A. The Meaning of Erasing in RDF under the Katsuno-Mendelzon Approach. In: *Ninth International Workshop on the Web and Databases, WebDB 2006, Chicago, Illinois, USA, June 30, 2006*. [S.l.: s.n.], 2006.

GUTIERREZ, C.; HURTADO, C.; VAISMAN, A. *RDFS Update: From Theory to Practice*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. 93–107 p.

HAMMOUD, M. et al. DREAM: Distributed RDF Engine with Adaptive Query Planner and Minimal Communication. *Proc. VLDB Endow.*, VLDB Endowment, v. 8, n. 6, p. 654–665, 2015.

HARBI, R. et al. Accelerating SPARQL Queries by Exploiting Hash-based Locality and Adaptive Partitioning. *The VLDB Journal*, v. 25, n. 3, p. 1–26, 2016.

HAUGLID, J. O.; RYENG, N. H.; NØRVÅG, K. DYFRAM: dynamic fragmentation and replica management in distributed database systems. *Distributed and Parallel Databases*, v. 28, n. 2, p. 157–185, 2010.

HOSE, K.; SCHENKEL, R. WARP: Workload-aware replication and partitioning for RDF. In: *4th International Workshop on Data Engineering meets Semantic Web (DESWeb 2013)*. [S.l.]: IEEE Computer Society, 2013. p. 1–6.

HUANG, J.; ABADI, D. J.; REN, K. Scalable SPARQL Querying of Large RDF Graphs. *PVLDB*, v. 4, n. 11, p. 1123–1134, 2011.

HUSAIN, M. F. et al. Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing. *IEEE Trans. Knowl. Data Eng.*, v. 23, n. 9, p. 1312–1327, 2011.

KAOUDI, Z.; MANOLESCU, I. RDF in the Clouds: A Survey. *The VLDB Journal*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, v. 24, n. 1, p. 67–91, 2015.

Karypis Lab. *Family of Graph and Hypergraph Partitioning Software*. 2014. ⟨http://glaros.dtc.umn.edu/gkhome/metis/metis/overview⟩.

KIM, J. et al. Taming Subgraph Isomorphism for RDF Query Processing. *Proc. VLDB Endow.*, VLDB Endowment, v. 8, n. 11, p. 1238–1249, 2015.

LEE, J. et al. An In-depth Comparison of Subgraph Isomorphism Algorithms in Graph Databases. In: . [S.l.]: VLDB Endowment, 2012. v. 6, n. 2, p. 133–144. ISSN 2150-8097.

LEE, K.; LIU, L. Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning. *Proc. VLDB Endow.*, VLDB Endowment, v. 6, n. 14, p. 1894–1905, set. 2013.

MINH-DUC, P. et al. Deriving an Emergent Relational Schema from RDF Data. In: *Proceedings of the 24th International Conference on World Wide Web, WWW 2015, Florence, Italy, May 18-22, 2015*. [S.l.: s.n.], 2015. p. 864–874.

MOERKOTTE, G.; NEUMANN, T. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In: *Proceedings of the 32Nd International Conference on Very Large Data Bases*. [S.l.]: VLDB Endowment, 2006. (VLDB '06), p. 930–941.

NEUMANN, T.; WEIKUM, G. The RDF-3X Engine for Scalable Management of RDF Data. *The VLDB Journal*, v. 19, n. 1, p. 91–113, 2010.

NISAR, M. U.; FARD, A.; MILLER, J. A. Techniques for graph analytics on big data. In: *IEEE International Congress on Big Data, BigData Congress 2013, June 27 2013-July 2, 2013*. [S.l.]: IEEE, 2013. p. 255–262.

OZSU, M. T.; VALDURIEZ, P. *Principles of Distributed Database Systems, 3rd Edition*. [S.l.: s.n.], 2011.

PAPAILIOU, N. et al. H2RDF+: An efficient data management system for big rdf graphs. In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 2014. (SIGMOD '14), p. 909–912.

PENG, P. et al. Processing sparql queries over distributed rdf graphs. *The VLDB Journal*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, v. 25, n. 2, p. 243–268, 2016. ISSN 1066-8888.

PÉREZ, J.; ARENAS, M.; GUTIERREZ, C. Semantics and complexity of sparql. *ACM Trans. Database Syst.*, ACM, New York, NY, USA, v. 34, n. 3, p. 16:1–16:45, 2009.

PRZYJACIEL-ZABLOCKI, M. et al. Cascading Map-side Joins over HBase for Scalable Join Processing. *CoRR*, abs/1206.6293, 2012.

PUNNOOSE, R.; CRAINICEANU, A.; RAPP, D. Rya: A Scalable RDF Triple Store for the Clouds. In: *Proceedings of the 1st International Workshop on Cloud Intelligence*. New York, NY, USA: [s.n.], 2012. p. 4:1–4:8.

ROHLOFF, K.; SCHANTZ, R. E. High-performance, Massively Scalable Distributed Systems Using the MapReduce Software Framework: The SHARD Triple-store. In: *Programming Support Innovations for Emerging Distributed Applications*. New York, NY, USA: ACM, 2010. p. 4:1–4:5.

SCHATZLE, A. et al. S2x: Graph-parallel querying of rdf with graphx. In: WANG, F. et al. (Ed.). [S.l.]: Springer, 2015. (Lecture Notes in Computer Science, v. 9579), p. 155–168.

SCHROEDER, R. *Uma Abordagem para o Particionamento de Dados na Nuvem Baseada em Relações de Afinidade em Grafos*. Tese (Doutorado) — Universidade Federal do Paraná, 2014.

SCHROEDER, R.; HARA, C. S. Partitioning Templates for RDF. In: TADEUSZ, M.; VALDURIEZ, P.; BELLATRECHE, L. (Ed.). *Advances in Databases and Information Systems: 19th East European Conference*. [S.l.]: Springer, 2015. (Lecture Notes in Computer Science, v. 9282), p. 305–319.

SEABORNE, A. et al. SPARQL/Update: A language for updating RDF graphs. *W3C member submission 15*, 2008.

SHANG, H. et al. Taming Verification Hardness: An Efficient Algorithm for Testing Subgraph Isomorphism. *Proc. VLDB Endow.*, VLDB Endowment, v. 1, n. 1, p. 364–375, 2008.

SHAO, B.; WANG, H.; LI, Y. Trinity: A Distributed Graph Engine on a Memory Cloud. In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 2013. (SIGMOD '13), p. 505–516.

STOCKER, M. et al. SPARQL Basic Graph Pattern Optimization Using Selectivity Estimation. In: *17th International Conference on World Wide Web (WWW 2008)*. New York, USA: [s.n.], 2008. p. 595–604.

ULLMANN, J. R. An Algorithm for Subgraph Isomorphism. *Journal of the ACM (JACM)*, v. 23, n. 1, p. 31–42, 1976.

WILKINSON, K. et al. Efficient RDF Storage and Retrieval in Jena2. In: *Exploiting Hyperlinks 349*. [S.l.: s.n.], 2003. p. 35–43.

WU, B.; JIN, H.; YUAN, P. Scalable SAPRQL Querying Processing on Large RDF Data in Cloud Computing Environment. In: ZU, Q.; HU, B.; ELÇI, A. (Ed.). *Pervasive Computing and the Networked World: Joint International Conference, ICPCA/SWS 2012*. [S.l.]: Springer, 2012. (Lecture Notes in Computer Science, v. 7719), p. 631–646.

YANG, T. et al. Efficient SPARQL Query Evaluation via Automatic Data Partitioning. In: MENG, W. et al. (Ed.). *DASFAA (2)*. [S.l.]: Springer, 2013. (Lecture Notes in Computer Science), p. 244–258.

ZAHARIA, M. et al. Fast and interactive analytics over hadoop data with spark. *USENIX Login*, v. 37, n. 4, p. 45–51, 2012.

ZENG, K. et al. A distributed graph engine for web scale RDF data. In: *Proceedings of the 39th international conference on Very Large Data Bases*. [S.l.]: VLDB Endowment, 2013. (PVLDB'13), p. 265–276.

ZHAO, P.; HAN, J. On Graph Query Optimization in Large Networks. *Proc. VLDB Endow.*, VLDB Endowment, n. 1-2, p. 340–351, 2010.

ZILIO, D. C. *Physical Database Design Decision Algorithms and Concurrent Reorganization for Parallel Database Systems*. Tese (Doutorado) — University of Toronto, 1998.