Comunicação em bloco na exploração de grafos em bases RDF distribuídas

Eduardo Villas Boas Santos[1], **Carmem S. Hara**[2], **Raqueline R. M. Penteado**[1]

[1] Departamento de Informática – Universidade Estadual de Maringá
Avenida Colombo 5.790 – 87.020-900 – Maringá, PR – Brasil

[2]Departamento de Informática – Universidade Federal do Paraná
Caixa Postal 19.081 – 81.531-990 – Curitiba, PR – Brasil
ra62973@uem.br, carmem@inf.ufpr.br, raque@din.uem.br

Abstract. *Distributed SPARQL query processing involves the exchange of intermediate results among RDF storage servers. This paper analyzes the impact of grouping these results into blocks in order to reduce the number of transmissions. The proposed communication strategy has been implemented on a SPARQL query processor based on a graph exploration algorithm. Experimental results showed that block-based communication can improve the performance of distributed query processing. Future works include extension of the SPARQL processor by considering the cost of block communication in query planning and optimization.*

Resumo. *Sistemas RDF distribuídos adotam estratégias de comunicação que trocam resultados intermediários entre servidores durante a execução distribuída de consultas. Este artigo analisa o impacto do agrupamento de resultados intermediários em blocos, com a finalidade de reduzir a quantidade de transmissões. A comunicação em blocos foi implementada em um processador de consultas SPARQL baseado em um algoritmo de exploração de grafos. Resultados experimentais mostram que esta estratégia de comunicação pode melhorar o desempenho de consultas distribuídas. Trabalhos futuros envolvem alterações no processador analisado a fim de considerar o agrupamento no cálculo do custo de comunicação do otimizador de consultas.*

1. Introdução

A flexibilidade e a simplicidade do modelo de dados RDF (*Resource Description Framework*) tem motivado a proliferação de bases de dados que o adotam. Uma base RDF é composta por um conjunto de triplas *(sujeito,predicado,objeto)* e pode ser vista como um grafo, no qual sujeitos e objetos são representados por nodos ligados por arestas que representam os predicados, conforme mostra a Figura 1(a). De acordo com o consórcio *W3C*[1], bases de dados RDF comerciais têm alcançado o tamanho de 1 trilhão de triplas[2]. Logo, o processamento eficiente de consultas SPARQL (*SPARQL Protocol and RDF Query Language*) neste tipo de base tornou-se um grande desafio para os sistemas gerenciadores de dados RDF.

A adoção de abordagens centralizadas para o armazenamento de dados penaliza a escalabilidade em sistemas que processam grandes volumes de dados. Sendo assim,

[1]http://www.w3.org
[2]http://www.w3.org/wiki / LargeTripleStores

sistemas com armazenamento distribuído têm sido propostos. Nestes sistemas, tanto os dados quanto as consultas são distribuídas entre servidores a fim de promover o processamento distribuído e paralelo de consultas. Porém, a troca de dados entre servidores durante o processamento atinge diretamente no custo de comunicação no processamento de consultas [Ozsu and Valduriez 2011].

Em geral, sistemas RDF distribuídos adotam estratégias de comunicação que trocam resultados intermediários entre servidores durante o processamento distribuído de consultas SPARQL. [Penteado et al. 2016] denomina este tipo de estratégia como *send-result*. A unidade de comunicação utilizada nestes sistemas pode considerar um ou mais resultados intermediários. Segundo [Galárraga et al. 2012], o envio em blocos (grupos de resultados intermediários) pode minimizar o custo de comunicação nestes sistemas.

Este artigo implementa o conceito de blocos na estratégia *send-result* do sistema proposto em [Penteado et al. 2016] que processa consultas SPARQL por meio de um algoritmo de exploração distribuída de grafos. Estudos experimentais mostram que a agrupamento de resultados intermediários pode minimizar o tempo de processamento de consultas distribuídas em alguns cenários específicos.

O artigo está organizado da seguinte forma: a Seção 2 apresenta os trabalhos relacionados; a Seção 3 apresenta os conceitos envolvidos no artigo; a Seção 4 apresenta a abordagem de agrupamento de resultados intermediários implementada no sistema analisado; a Seção 5 mostra experimentos baseados na abordagem de agrupamento, bem como os resultados alcançados; e, a Seção 6 conclui o artigo apresentando trabalhos futuros.

2. Trabalhos relacionados

A unidade de comunicação adotada no envio de resultados intermediários entre servidores pode variar entre os sistemas RDF distribuídos. Uma unidade de comunicação pode ser um único resultado intermediário ou um grupo de resultados intermediários.

[Galárraga et al. 2012] propõe um sistema que adota como unidade de comunicação um lote de 1024 resultados intermediários. Os autores justificam que o uso de blocos como unidade minimiza o custo de comunicação em sistemas distribuídos. Em contrapartida, sistemas tais como os propostos em [Rohloff:2010 2010] e [Goasdoué et al. 2013] adotam um único resultado intermediário como unidade de comunicação na comunicação *send-result*. Os dois sistemas usam o *framework MapReduce Hadoop*[3] no processamento distribuído de consultas. Neste *framework*, um plano de execução é composto por tarefas de mapeamento e redução. Uma tarefa de mapeamento gera um conjunto de pares `<chave,valor>` em um servidor. Cada par deste servidor é enviado e processado por uma tarefa de redução executada em um servidor remoto.

O sistema proposto em [Penteado et al. 2016] propõe um processador de consultas baseado em um algoritmo de exploração distribuída de grafos que viabiliza a escolha entre duas estratégias de comunicação durante a execução de consultas: o envio de resultados intermediários para outros servidores (*send-result*) ou a requisição de fragmentos necessários de outros servidores (*get-frag*). Entende-se por fragmento um conjunto de triplas RDF que segue uma estrutura pré-definida no momento da distribuição de dados nos servidores do sistema. A estratégia *send-result* adota um (único) resultado intermediário

[3] https://hadoop.apache.org/docs/r1.2.1/index.html

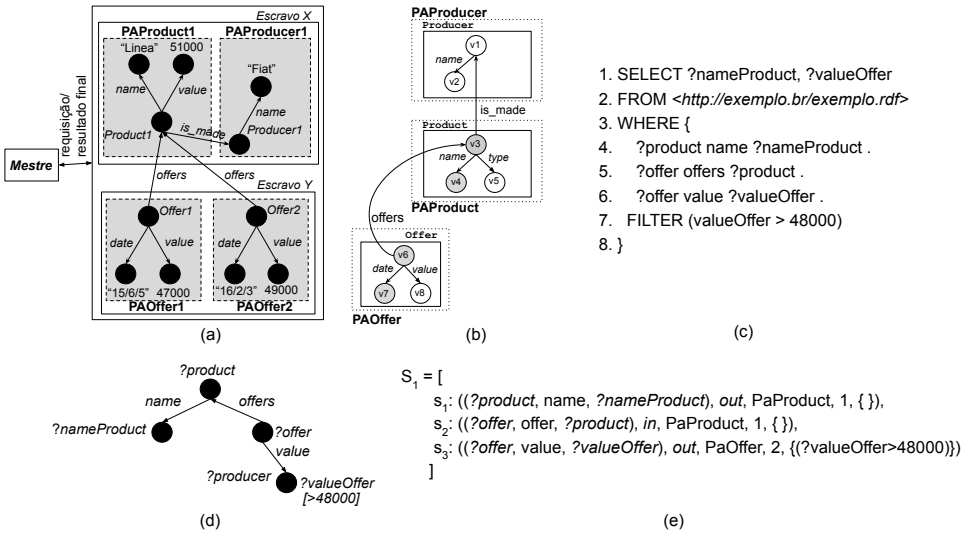


Figura 1. Base RDF (a); Grafo de sumarização da base RDF (b); Consulta SPARQL (c); Padrão de grafo básico da consulta (d); Plano de execução da consulta (e)

como unidade de comunicação e *get-frag*, um fragmento de dados. Este artigo estende o processador implementando o conceito de agrupamento de resultados intermediários em sua estratégia *send-result*. Assim, a estratégia pode utilizar blocos como unidade de comunicação, que resulta a redução do custo de comunicação no sistema.

3. Definições preliminares

Uma base RDF é composta por um conjunto de triplas *(sujeito,predicado,objeto)* representando que um sujeito tem uma propriedade com um objeto. Bases RDF podem ser definidas como um conjunto de nodos com suas listas de adjacência. Por exemplo, na base RDF da Figura 1(a), o conjunto de arestas incidentes em *Producer1* é *in*:{(is_made, {Product1})} e o conjunto de arestas que partem, *out*:{(name, *"Fiat"*)}.

O núcleo da sintaxe da linguagem SPARQL é baseado em um conjunto de padrões de triplas semelhantes às triplas RDF, que admitem variáveis em seus termos. A consulta da Figura 1(c) recupera, para cada produto, o seu nome e os valores de suas ofertas maiores que *48000*. O conjunto de padrões de triplas de uma consulta SPARQL constitui um padrão de grafo denominado de padrão de grafo básico (PGB) da consulta. A Figura 1(d) representa o PGB da consulta da Figura 1(c), onde os sujeitos e os objetos dos padrões de triplas são representados por nodos e os predicados por arestas. O filtro da consulta está representado entre colchetes abaixo do nodo apropriado. O processamento de consultas SPARQL pode ser transformado para um problema de casamento de grafos onde subgrafos da base RDF que são homomórficos ao PGB de uma consulta são recuperados.

O processador de consultas SPARQL proposto em [Penteado et al. 2016] tem como base um algoritmo de exploração distribuída de grafos que adota um método de comunicação híbrido. Ele adota uma arquitetura mestre-escravo, na qual fragmentos de dados são distribuídos nos escravos de um *cluster* por meio de padrões de alocação (PAs) previamente definidos a partir do grafo de sumarização da base, que representa seu esquema. Fragmentos de dados são instâncias de PAs que seguem sua estrutura. Os fragmentos garantem co-alocação no armazenamento, além de serem usados como unidades de comunicação no sistema. A Figura 1(b) apresenta o grafo de sumarização G_S da base da

Figura 1(a). G_S possui três PAs que estão representados por figuras tracejadas e nomeados de acordo com a classe que os compõem. Os PAs adotam o padrão estrutural do tipo estrela, garantindo que cada sujeito que representa um recurso seja armazenado com os seus respectivos objetos literais. Fragmentos estão representados por meio de figuras cinzas na Figura 1(a). Por exemplo, os fragmentos *PaOffer1* e *PaOffer2* da Figura 1(a) são instâncias do padrão de alocação *PAOffer* da Figura 1(b).

O planejamento de consultas do processador baseia-se nos PAs de G_S a fim de minimizar a comunicação entre servidores durante o processamento de consultas. O subgrafo destacado na Figura 1(b) representa o subgrafo homomórfico ao grafo PGB da Figura 1(d). A Figura 1(e) mostra o plano de consulta gerado. Cada passo s_i do plano é uma tupla (a, dir, pat, id, f), onde: (**1**) a é um padrão de tripla da consulta, (**2**) $dir \in \{in, out\}$; se $dir = out$ a exploração é do sujeito para o objeto. Caso contrário, a exploração é do objeto para o sujeito; (**3**) pat é o PA que contém a; (**4**) id é um identificador único associado ao PA. Todos os passos com um mesmo id compõem um bloco de exploração dentro de um mesmo fragmento. (**5**) f é um conjunto de filtros definidos em a.

Uma vez gerado o plano da consulta, o mestre inicia a execução enviando o plano para todos os escravos do *cluster*. Cada escravo inicia a execução em paralelo recuperando os pontos iniciais de exploração de acordo com o plano, que correspondem a nodos de fragmentos do primeiro passo do plano (s_1). Ou seja, nodos de fragmentos de $s_1.pat$ alocados no escravo são recuperados para obter as triplas que casam com o padrão $s_1.a$. Caso a direção de exploração $s_1.dir$ seja out, os pontos iniciais de exploração são os sujeitos das triplas; caso contrário, os pontos iniciais são os objetos das triplas. O resultado da execução de um passo é a associação de variáveis a nodos da base. Na execução do passo s_1 do exemplo, o escravo X recupera *Product1* gerando o mapeamento $[\![s_1]\!]_X$: $\{(?product \mapsto Product1, ?nameProduct \mapsto \text{“}Linea'')\}$. Na sequência, cada passo s_i do plano adiciona novos mapeamentos. Se $s_i.a$ não é mapeado para a base local ou se o novo mapeamento não satisfaz $s_1.f$, o mapeamento correspondente é removido do conjunto resultante. No exemplo, a execução dos passos s_1 e s_2 é feita no escravo X uma vez eles possuem o mesmo id. Após o passo s_2, o resultado intermediário gerado por X é $[\![s_2]\!]_X$: $[\![s_1]\!]_X \cup \{(?offer \mapsto \{Offer1, Offer2\})\}$. s_3 pode ser executado ou não em X, uma vez que $s_2.id \neq s_3.id$. No exemplo, as ofertas estão armazenadas no servidor Y. Logo, X se comunica com Y para continuar a execução. Neste momento, ou X requisita os fragmentos das ofertas para Y e continua o processamento localmente (usando a estratégia *get-frag*), ou o resultado intermediário gerado é enviado para Y continuar a execução do plano (usando a estratégia *send-result*). O otimizador de consultas escolhe uma das estratégia por meio de funções de custo. A escolha depende do número de mensagens e do volume de dados a ser transmitido entre os escravos. Continuando a execução, o resultado intermediário gerado para *Offer1* é descartado na execução de s_3, uma vez que o valor da oferta não satisfaz o filtro $s_3.f$. Por fim, quando um escravo explora o último passo do plano, o seu conjunto de mapeamentos é projetado usando as variáveis da cláusula *SELECT* da consulta e o resultado final é enviado para o mestre.

4. Agrupamento de resultados intermediários

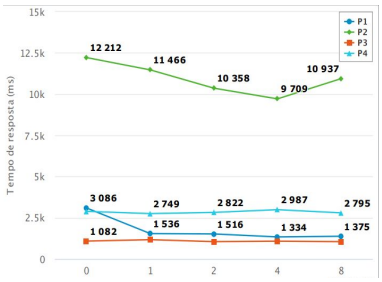
A abordagem proposta neste artigo viabiliza a escolha do tamanho do bloco de resultados intermediários usado na estratégia *send-result*. A abordagem envolve duas operações, uma de agrupamento e outra de desagrupamento de resultados intermediários.

```
Algoritmo: agrupamento de resultados intermediários
Entrada: [L, fGroup]
1. accSize := 0;
2. groupedResults := { };
3. For each l in L
4. {
5.      currentSize := size(l);
6.      accSize := accSize + currentSize;
7.      groupedResults := groupedResults ∪ {l} ;
8.      If (index(l)=size(L)) or (accSize>=fGroup)
9.      {
10.             send(header,groupedResults);
11.             accSize := 0;
12.             groupedResults := { };
13.     }
14.}
```

(a)

P	#fGroup	#Ag	#Msg	#Vol	#Rtime
$P1$	0	1	1222	296	3086
$P1$	1	4	306	1184	1536
$P1$	2	7	175	2072	1516
$P1$	4	14	88	4144	1334
$P1$	8	28	44	8288	1375
$P2$	0	1	1222	1284	12212
$P2$	1	1	1222	1284	11466
$P2$	2	2	611	2444	10358
$P2$	4	4	306	4888	9709
$P2$	8	7	175	8554	10937
$P3$	0	1	57	902	1082
$P3$	1	2	29	1804	1167
$P3$	2	3	19	2706	1049
$P3$	4	5	15	4510	1078
$P3$	8	10	6	7216	1047
$P4$	0	1	57	1233	2884
$P4$	1	1	57	1233	2749
$P4$	2	2	29	2466	2822
$P4$	4	4	15	4932	2987
$P4$	8	7	9	8631	2795

(b)



(c)

Figura 2. Algoritmo de agrupamento (a); Dados sobre agrupamento variando *fGroup* de 0–8 nos planos $P1$–$P4$ (b); Relação entre *fGroup* e tempo de execução dos planos (c)

O algoritmo da Figura 2(a) descreve a operação de agrupamento usada por um escravo que usa um fator de agrupamento (*fGroup*) no momento do envio de uma lista de resultados intermediários (L). Este fator define um limiar para o volume da mensagem em *bytes* que será enviada ao servidor remoto implicando diretamente no número de resultados intermediários agrupado em cada mensagem. A variável *accSize* monitora o volume da mensagem (linha 6) e *groupedResults* o agrupamento dos resultados intermediários (linha 7). Caso o último resultado intermediário da lista L seja considerado no agrupamento ou o tamanho da mensagem alcance ou ultrapasse o limiar definido para a mensagem (linha 8), o grupo é enviado para o escravo remoto (linha 10). Cada envio tem um cabeçalho a fim de auxiliar a operação de desagrupamento. O cabeçalho indica dados como o número de resultados intermediários do bloco e a finalização do envio do bloco.

A operação de desagrupamento é iniciada no servidor remoto com o recebimento de uma mensagem. O desagrupamento ocorre em paralelo com a operação de recebimento de mensagens. Cada mensagem recebida é desagrupada e os seus resultados intermediários são armazenados no servidor remoto. A exploração de grafos continua no servidor remoto assim que todos os resultados intermediários são recebidos do escravo emissor.

5. Estudo experimental

O objetivo dos experimentos foi a análise do impacto do fator de agrupamento no tempo de execução de consultas. A abordagem de agrupamento foi implementada no sistema proposto em [Penteado et al. 2016] usando Java e o protocolo de comunicação TCP/IP. A biblioteca GSON[4] foi usada: no envio de mensagens, mapeando listas de resultados intermediários para o formato JSON[5]; e, no desagrupamento, convertendo um fluxo de *bytes* em uma lista de resultados intermediários. *fGroup* variou entre 0, 1, 2, 4 e 8 *kilobytes*.

O experimento foi realizado em um *cluster* com três servidores, sendo um mestre e dois escravos. Uma base com 1.875.815 triplas foi gerada a partir do benchmark Berlin[6] e distribuída entre os dois escravos. a Figura 1(b) mostra o grafo de sumarização da base. Quatro planos de execução foram usados no experimento envolvendo *Product* e *Producer*.

Os escravos comportaram-se de maneira similar no experimento. A Figura 2(b) refere-se às mensagens enviadas por um escravo durante a execução dos planos. Para cada

[4]https://google.github.io/gson/apidocs/com/google/gson/Gson.html
[5]https://www.oracle.com/technetwork/articles/java/json-1973242.html
[6]http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/

execução, a figura mostra: o plano executado (P), o fator de agrupamento (*#fGroup*), o número de resultados intermediários agrupados (*#Ag*), o número de mensagens enviadas (*#Msg*), o volume médio de cada mensagem (*#Vol*) e o tempo médio de execução do plano em milisegundos (*#Rtime*). Por exemplo, quando *#fGroup*=1 em $P1$, o servidor enviou 306 mensagens, cada uma com 4 resultados intermediários e volume médio de 1.184 *bytes*. O tempo médio da execução de $P1$ foi 1.536 ms. Observe que os planos variaram o *número* e o *volume* de resultados intermediários enviados do escravo, conforme mostra a Figura 2(b) quando *#fGroup*=0. Em $P1$ e $P2$ foram enviados 1.222 resultados intermediários de volume 296 e 1.284 *bytes*, respectivamente. Em $P3$ e $P4$ foram enviados 57 resultados intermediários de volumes 902 e 1.233 *bytes*, respectivamente. Vale destacar que o fator de agrupamento 0 corresponde ao *send-result* original do sistema.

A Figura 2(c) mostra o comportamento do tempo de resposta das execuções com a variação de *fGroup* nos quatro planos. É possível notar que $P1$ e $P2$ obtiveram vantagem com o agrupamento. Em contrapartida, o mesmo não ocorreu com $P3$ e $P4$. O desempenho de $P1$ e $P2$ melhorou conforme *fGroup* aumentou de 0 até 4. Porém, o mesmo não ocorreu quando *fGroup* foi de 4 para 8. Apesar da redução do número de mensagens (*#Msg*) conforme pode ser constatado na Figura 2(b), notou-se o custo da operação de desagrupamento quando o número de resultados intermediários agrupados (*#Ag*) aumentou. Em relação à $P1$, o volume (*#Vol*) penalizou $P2$ aumentando o número mensagens trocado entre os escravos e o tempo de execução. Por fim, pode ser observado que $P2$ e $P4$ possuem o volume médio de cada resultado intermediário similar, 1.284 e 1.233 *bytes*, respectivamente. $P2$ enviou 1.222 resultados intermediários para o escravo remoto e $P4$, 57. O tempo de execução de $P4$ não melhorou com o aumento de *fGroup*, como ocorreu com $P2$. Logo, notou-se que o agrupamento melhorou o desempenho de planos que trocaram um número significativo de resultados intermediários entre pares de escravos.

6. Conclusão

Este artigo apresenta uma análise sobre o agrupamento de resultados intermediários pela estratégia de comunicação *send-result* de um processador de consultas SPARQL baseado na exploração distribuída de grafos. A abordagem de agrupamento analisada viabiliza o uso de diferentes tamanhos de bloco na estratégia. Resultados experimentais mostram que a abordagem pode otimizar o processamento de consultas em algumas situações. Trabalhos futuros envolvem o uso de agrupamento no cálculo do custo de comunicação da estratégia *send-result* do sistema analisado. Em cada comunicação, o otimizador do processador poderá optar ou não pelo agrupamento, escolhendo o volume de bloco adequado.

Referências

Galárraga, L., Hose, K., and Schenkel, R. (2012). Partout: A Distributed Engine for Efficient RDF Processing. volume abs/1212.5636.

Goasdoué, F., Kaoudi, Z., Manolescu, I., Quiané-Ruiz, J., and Zampetakis, S. (2013). CliqueSquare: efficient Hadoop-based RDF query processing. In *BDA'13 - Journées de Bases de Données Avancées*.

Ozsu, M. T. and Valduriez, P. (2011). *Principles of Distributed Database Systems, 3rd Ed.*

Penteado, R. R. M., Schroeder, R., and Hara, C. S. (2016). Exploring controlled RDF distribution. In *IEEE CloudCom 2016, Luxembourg, December 12-15, 2016*, pages 160–167.

Rohloff:2010 (2010). High-performance, Massively Scalable Distributed Systems Using the MapReduce Software Framework: The SHARD Triple-store. In *Programming Support Innovations for Emerging Distributed Applications*, pages 4:1–4:5, New York, NY, USA. ACM.