REBECA SCHROEDER FREITAS

UMA ABORDAGEM PARA O PARTICIONAMENTO DE DADOS NA NUVEM BASEADA EM RELAÇÕES DE AFINIDADE EM GRAFOS

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.
Orientadora: Prof. Dra. Carmem Satie Hara

CURITIBA
2014

Ministério da Educação
Universidade Federal do Paraná
Programa de Pós-Graduação em Informática

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa da aluna de Doutorado em Ciência da Computação, Rebeca Schroeder Freitas, avaliamos a tese de doutorado intitulada *"Uma Abordagem para o Particionamento de Dados na Nuvem Baseada em Relações de Afinidade em Grafos"*, cuja defesa pública foi realizada no dia 21 de julho de 2014, às 09:00 horas, no Departamento de Informática do Setor de Ciências Exatas da Universidade Federal do Paraná. Após avaliação, decidimos pela:

(X) **aprovação** da candidata. ( ) **reprovação** da candidata.

Curitiba, 21 de julho de 2014.

Profa. Dra. Carmem Satie Hara
DINF/UFPR – Orientadora

Prof. Dr. Javam Machado
UFC – Membro externo

Prof. Dr. Sergio Lifschitz
PUC-RIO – Membro Externo

Prof. Dr. Elias Procópio Duarte Júnior
DINF/UFPR – Membro Interno

Prof. Dr. André Vignatti
DINF/UFPR – Membro Interno

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

À Diego, Juanir, Cristina e Timóteo

AGRADECIMENTOS

Agradeço a Deus por me presentear com a vida e com Sua presença sempre constante no caminho que Ele preparou para que eu trilhasse. Sua onipresença e onisciência proveu tudo que precisei para chegar até aqui e desejar ir além. Nesta longa jornada acadêmica, impossível não olhar para trás e deixar de honrar meus dois primeiros professores, e que são também meus pais. Sou imensamente grata ao meu pai pelo tempo que dedicou para me motivar a gostar das ciências exatas durante a minha formação no ensino fundamental e médio. Esta gratidão se estende a minha mãe por ter acompanhado com muito zelo minha educação. Nunca esquecerei do tempo em que ela me acordava cedo antes de provas para passar a matéria comigo, e sempre com um café reforçado e um sorriso incomparável. Estou certa de que este zelo e dedicação durante as etapas iniciais da minha formação me permitiram chegar até aqui.

Faço um agradecimento muito especial a minha orientadora Carmem Satie Hara pelo seu indispensável acompanhamento durante todo o desenvolvimento deste trabalho. Certamente não haveria sucesso na conclusão desta tese sem a sua dedicação na forma de valiosas contribuições e constantes direcionamentos. Agradeço por todo o esforço dispensado para obter os recursos necessários para desenvolver o trabalho e para apresentar nossos artigos em conferências. Me sinto muito honrada de ter sido sua primeira aluna de doutorado, e de ter a certeza de que não poderia haver orientador melhor! Levo comigo o seu exemplo e estarei sempre grata e disposta a continuar esta agradável parceria de amizade e projetos.

Aos membros da banca examinadora, professores Javam Machado, Sérgio Lifschitz, Elias Procópio Duarte Júnior e André Luis Vignatti, os meus sinceros agradecimentos pela revisão da minha tese e pelas inúmeras sugestões que contribuíram para vislumbrar o futuro deste trabalho. Agradeço também aos professores e funcionários do Departamento de Informática da UFPR que direta ou indiretamente contribuíram na minha formação. Em especial, agradeço ao professor André Guedes por me ajudar a situar o problema desta tese no contexto de grafos. Agradeço à CAPES pela concessão da bolsa de doutorado, ao CNPQ por custear parte deste projeto e à Amazon AWS pelo *grant* recebido.

Agradeço aos diversos colegas e amigos que compartilharam comigo as mesmas disciplinas, laboratórios e histórias durante este período. Sou imensamente grata as minhas amigas Cinara e Elisa, com as quais tive o prazer de dividir alegrias, dificuldades, cafés, idas e vindas de Joinville a Curitiba, sempre regadas de boas risadas. Agradeço aos colegas do grupo GDAI, Raqueline, Marco Aurélio, Diego, Ricardo, Patrick e Jaqueline, do qual muito me orgulho em fazer parte. Estendo meus agradecimentos aos colegas João Eugenio, Edson Ramiro, Bruno Velasco, Nuno e Tarcízio, por tornarem a convivência no laboratório de BD tão agradável desde o primeiro ano de doutorado.

Sou grata de todo meu coração a minha família pelo suporte durante todo este processo. Agradeço ao meu esposo Diego, pelo amor e compreensão que foram indispensáveis para que eu me sentisse bem mesmo nos momentos mais intensos deste curso. A meus pais Juanir e Cristina e a meu irmão Timóteo, meu sinceros e carinhosos agradecimentos por tudo que fizeram e ainda fazem por mim. Aos meus sogros e cunhadas, agradeço pelo apoio de sempre. Aos meus queridos amigos, meu afetuoso muito obrigada pela torcida!

"Porque Dele e por Ele, e para Ele, são todas as coisas"

Romanos 11:36a

RESUMO

Os desafios atuais do gerenciamento de dados vêm sendo frequentemente associados ao termo *Big Data*. Este termo refere-se a um número crescente de aplicações caracterizadas pela produção de dados com alta *variedade*, grande *volume*, e que exigem *velocidade* em seu processamento. Ao mesmo tempo em que estes requisitos são identificados, o amadurecimento tecnológico associado à computação em *nuvem* alavancou uma mudança nos aspectos operacionais e econômicos da computação, sobretudo através de infraestruturas para o desenvolvimento de serviços escaláveis. A iniciativa do gerenciamento de dados sobre estas plataformas mostra-se adequada para tratar os desafios do *Big Data* através de um serviço de banco de dados em nuvem (*Database as a Service*). Uma forma de escalar aplicações que processam uma quantidade massiva de informações é através da fragmentação de grandes conjuntos de dados alocados sobre servidores de um sistema em nuvem. O principal problema associado a esta abordagem está em particionar os dados de forma que consultas possam ser preferencialmente executadas de forma local para evitar o custo da troca de mensagens entre servidores. Em conjunto com este problema, a variedade de dados e o volume crescente associado às bases de dados em nuvem desafiam as soluções tradicionais para o particionamento de dados.

Esta tese propõe um novo método para o particionamento de dados que tem como objetivo promover a escalabilidade de repositórios em nuvem. Para minimizar o custo da execução de consultas distribuídas, heurísticas sobre informações de carga de trabalho são utilizadas para identificar afinidades entre dados e estabelecer o agrupamento de itens fortemente relacionados em um mesmo servidor. O problema do particionamento é tratado pelos processos de fragmentação e alocação. O processo de fragmentação define unidades de armazenamento que contêm itens de dados fortemente relacionados. Na fase seguinte, o processo de alocação utiliza o mesmo critério de agrupamento para co-alocar fragmentos nos servidores do repositório. A replicação é utilizada para maximizar a quantidade de dados relacionados em um mesmo servidor, porém, a quantidade de réplicas gerada é controlada por todo o processo. A metodologia proposta está focada em modelos em grafo estabelecidos pelos formatos RDF e XML, e que permitem representar uma variedade de outros modelos. A principal contribuição desta tese está em definir o particionamento sobre uma visão sumarizada de um banco de dados similar a um esquema de banco de dados. Além de evitar a exaustão do processo de particionamento sobre grandes bases, esta solução permite reaplicar a estratégia obtida sobre novas porções de dados que estejam de acordo com o esquema e a carga de trabalho assumidos pelo processo. Esta metodologia se mostra adequada para acomodar o volume crescente de dados associado a repositórios em nuvem. Resultados experimentais mostram que a solução proposta é efetiva para melhorar o desempenho de consultas, se comparada a abordagens alternativas que tratam o mesmo problema.

Palavras-chave: RDF, XML, Partitionamento, Fragmentação, Alocação, Banco de Dados Distribuído

ABSTRACT

The new challenges in data management have been referred to as *Big Data*. This term is related to an increasing number of applications characterized by generating data with a *variety* of types, huge *volume*, and by requiring high *velocity* processing. At the same time, cloud computing technologies are transforming the operational and economic aspects of computing, mainly due to the introduction of infrastructures to deploy scalable services. Cloud platforms have been properly applied to support data management and address *Big Data* challenges through a database service in the cloud (DaaS - Database as a Service). One approach to scale applications that process massive amounts of information is to fragment huge datasets and allocate them across distributed data servers. In this context, the main problem is to apply a partitioning schema that maximizes local query processing and avoids the cost of message passing among servers. Besides this problem, data variety and the ever-increasing volume of cloud datastores pose new challenges to traditional partitioning approaches.

This thesis provides a new partitioning approach to scale query processing on cloud datastores. In order to minimize the cost of distributed queries, we apply heuristics based on workload data to identify the affinity among data items and cluster the most correlated data in the same server. We tackle the data partitioning problem as a twofold problem. First, data fragmentation defines storage units with strongly correlated items. Further, data allocation aims to collocate fragments that share correlated items. Data replication is applied to cluster related data as much as possible. However, data redundancy is controlled throughout the process. We focus on graph models given by the RDF and XML formats in order to support data variety. Our main contribution is a partitioning strategy defined over a summarized view of the dataset given as a database schema. The result of the process consists of a set of partitioning templates, which can be used to partition an existing dataset, as well as maintain the partitioning process when new data that conform to the schema and the workload are inserted to the dataset. This approach is suitable to deal with the increasing volume of data related to cloud datastores. Experimental results show that the proposed solution is effective for improving the query performance in cloud datastores, compared to related approaches.

Keywords: RDF, XML, Partitioning, Fragmentation, Allocation, Distributed databases

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

AWS	- *Amazon Web Services*
BD	- Banco de Dados
BSBM	- *Berlin SPARQL Benchmark*
DHT	- *Distributed Hash Table*
OLAP	- *On-line Analytical Processing*
P2P	- Par-a-Par
RDF	- *Resource Description Framework*
SGBD	- Sistema Gerenciador de Banco de Dados
URI	- *Uniform Resource Identifier*
XML	- *eXtensible Markup Language*

LISTA DE TERMOS E DEFINIÇÕES

$\mathcal{U}$ - Conjunto de URIs

$\mathcal{L}$ - Conjunto de valores literais

$\mathcal{V}$ - Variáveis em uma consulta SPARQL

D - Grafo RDF, ou conjunto de triplas RDF

$G = (V, E, r)$ - Padrão de Grafo que representa uma consulta

$V(G)$ - Conjunto de nós de um padrão de grafo

$E(G)$ - Conjunto de arestas de um padrão de grafo

$\mathcal{P}$ - Particionamento de um grafo RDF D

Q - Conjunto de consultas SPARQL

$B(q)$ - Conjunto de subgrafos de D que representam o resultado de uma consulta q

$\hat{P}(q, \mathcal{P})$ - Segmentação de $B(q)$ que representa o número de partições a mais que precisam ser acessadas para responder a q sobre o particionamento $\mathcal{P}$

f - Frequência de execução de uma consulta q em um dado período de tempo

$S = (C, L, l, A, s, o)$ - *Estrutura RDF*

$Q_{(a,p,b)}$ - Consultas que acessam nós relacionados por uma propriedade p de forma que a é o nó origem e b é o nó destino

$aff(n_i, p, n_j)$ - Afinidade entre nós relacionados por uma propriedade p dada pelo somatório das frequências de consultas em $Q_{(i,p,j)}$

$\mathcal{A} = (N, \hat{E}, aff)$ - Grafo de afinidades

u - Quantidade máxima de dados replicados permitida para cada instância de um item de dado

Γ - Limiar para tamanho de fragmentos

T - Conjunto de *templates* de fragmentação

$Occ \downarrow (i)$ - Número médio de instâncias de um item de dado i quando aninhado a uma estrutura em árvore de um *template* de fragmento

$size(t_i)$ - Tamanho de t_i

$Occ \uparrow (i)$ - Número médio de instâncias replicadas de um item de dado i quando aninhado a uma estrutura em árvore de um *template* de fragmento

$sce(n)$ - Conjunto de arestas fortemente correlacionadas a partir de um nó n em um grafo de afinidades

$U = \{e_1, ..., e_m\}$ - Conjunto de *links* de alocação

$root(T)$ - Conjunto de nós que atuam como raiz de *templates*

H - Grupos de alocação

φ - Retorna um valor lógico “Sim”indicando que um determinado nó não pode ser replicado, e “Não”caso contrário.

SUMÁRIO

CAPÍTULO 1

INTRODUÇÃO

A computação nas nuvens, ou *cloud computing*, é o termo comumente associado à tendência tecnológica de hospedar um número crescente de aplicações, serviços e dados em *datacenters* de larga escala [Agrawal et al., 2013]. Esta tendência está promovendo transformações em diversos aspectos da computação através de novas formas de prover serviços. Infraestruturas, plataformas e softwares são oferecidos como *commodities* através de um modelo de custo sob-demanda que os disponibiliza em tempo real e prontos para o consumo. Em especial, um número crescente de aplicações está se beneficiando de plataformas nas nuvens [Kossmann et al., 2010]. Entretanto, esta proliferação tem como consequência o aumento do volume de dados sendo produzidos e consumidos por estas aplicações. Diversos desafios estão associados a este cenário conhecido como *Big Data*. Dentre eles, promover a escalabilidade de sistemas de gerenciamento de dados constitui parte fundamental para infraestruturas em nuvem.

O termo *Big Data* foi inicialmente definido por Douglas Laney [Laney, 2012] como aplicações de grande volume, alta velocidade e grande variedade de informações que requerem novas formas de processamento para permitir a tomada de decisão, a descoberta de conhecimento e otimização de processos. Esta definição é conhecida como modelo *3 Vs*, pois estabelece as 3 dimensões características do *Big Data*, isto é, Volume, Velocidade e Variedade. Embora o termo tenha sido definido em 2012, a necessidade de escalar estas 3 dimensões do gerenciamento de dados foi identificada em 2001 [Laney, 2001] impulsionada pelos efeitos do *e-commerce* e da integração de diversas aplicações. Uma década depois, o efeito *Big Data* ganhou nome e espaço especialmente devido à evolução e maturação de tecnologias envolvidas com a computação nas nuvens. Na prática, *Big Data* representa as características e desafios de um conjunto expressivo de aplicações, enquanto a computação

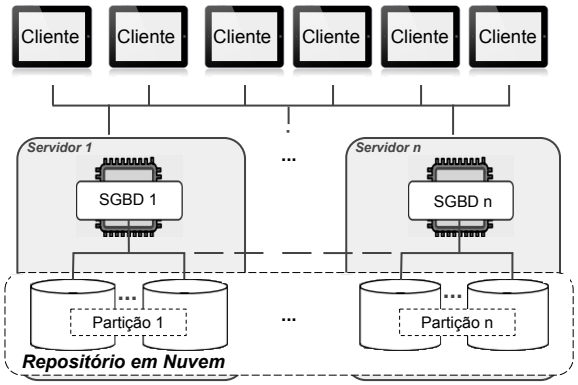


Figura 1.1: Arquitetura para o Gerenciamento de Dados Distribuídos

nas nuvens atua como a infraestrutura a ser utilizada e desenvolvida para responder aos diversos desafios impostos por estas aplicações.

Em relação ao suporte da *Variedade* de informações, destacam-se os modelos de dados XML (*eXtensible Markup Language*) e RDF (*Resource Description Framework*). O modelo XML se tornou o padrão para o intercâmbio de informações em aplicações Web [Angles and Gutierrez, 2008]. Sua ascensão está relacionada a aplicações de *e-commerce*, onde XML passou a ser utilizado como formato de integração de documentos e aplicações. Com advento da *web semântica*, conjuntos de dados provenientes de diferentes aplicações passaram a ser vinculados através de uma pilha de padrões abertos chamados de RDF. Atualmente, RDF é o padrão estabelecido pelo projeto *Linked Open Data*[1] para publicação de conjuntos de dados na Web, e assim habilitar o acesso a diferentes fontes que juntas formam uma base global de informações.

Uma quantidade cada vez maior de conjuntos de dados, ou *datasets*, vem sendo disponibilizada em diferentes domínios de aplicação. A DBpedia[2], por exemplo, atingiu um tamanho de 2.460 milhões de triplas RDF extraídas da Wikipedia. De acordo com o repositório *Large Triple Stores*[3], alguns *datasets* comerciais podem ser ainda maiores alcançando a casa de trilhões de triplas. Haja vista a tendência de crescimento constante destas fontes de dados, reconhece-se a necessidade eminente de sistemas de gerenciamento

[1] http://linkeddata.org/
[2] http://wiki.dbpedia.org/Datasets
[3] http://www.w3.org/wiki/LargeTripleStores

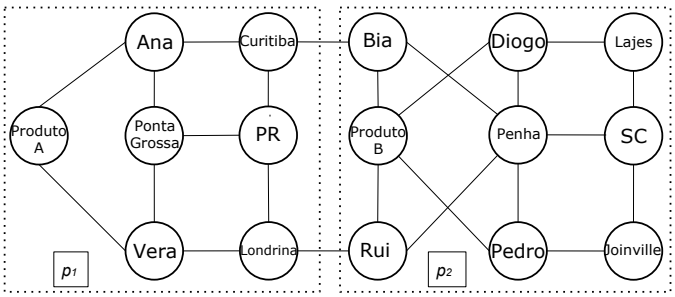


Figura 1.2: Um Exemplo do Particionamento de Dados

de dados capazes de dar suporte a esta demanda. Embora existam resultados significativos em escalar esses sistemas em uma única máquina [Franz Inc, 2013], esta abordagem não é realista, em particular quando a quantidade de dados continua a expandir. Uma forma de gerenciar estes *datasets* é através de uma arquitetura baseada em repositórios em nuvem [Kossmann et al., 2010], como a apresentada pela Figura 1. Nesta composição, um Sistema Gerenciador de Banco de Dados (SGBD) atua a partir de um servidor e compartilha seu repositório local e processamento com os demais servidores de um sistema distribuído. Este compartilhamento tende a promover a escalabilidade de aplicações sobre esta arquitetura, uma vez que a adição de novos servidores ao sistema aumenta a capacidade tanto de armazenamento quanto de processamento. Desta forma, é dito que o sistema é capaz de escalar horizontalmente pela distribuição de sua carga de forma a não comprometer, ou até melhorar, o desempenho de suas aplicações. Para que esta distribuição seja possível, *datasets* devem ser particionados sobre servidores distribuídos.

A estratégia de particionamento de dados adotada nestes ambientes determina a escalabilidade do SGBD [Cong et al., 2007]. O problema relacionado ao particionamento está em reduzir a latência de consultas através da minimização da comunicação e transferência de resultados intermediários entre servidores do sistema. Embora o processamento possa ser distribuído sobre os n servidores do sistema em paralelo, um particionamento de dados inadequado pode anular os benefícios desta estratégia quando a transferência de resultados intermediários entre servidores é inevitável.

1.1 Definição do Problema

Considere como exemplo de introdução ao problema, o grafo de dados da Figura 1.2, que representa um conjunto de dados relacionando produtos com seus representantes, bem como as respectivas cidades e estados atendidos por eles. Suponha que este conjunto de dados tenha sido particionado, e que a partição p_1 seja atribuída ao *Servidor 1* na arquitetura da Figura 1, e que a partição p_2 seja atribuída ao *Servidor 2*. Considere ainda que uma consulta deseja recuperar o nome dos representantes de cada um dos produtos. Neste caso é possível formar sub-consultas para recuperar os dados de cada um dos produtos de forma que a recuperação possa ser desempenhada paralelamente e independentemente sobre os dois servidores. Assim, nenhuma troca de resultados se faz necessária entre os servidores, uma vez que os resultados parciais gerados por cada unidade poderão ser simplesmente unidos. Neste caso, é dito que as partições oferecem uma cobertura completa para a consulta.

Trocas de resultados entre servidores seriam necessárias se além dos nomes dos representantes se desejasse saber as cidades atendidas por eles. Nesta situação, em especial a sub-consulta submetida ao *Servidor n*, as cidades atendidas por *Bia* e *Rui* não fazem parte da mesma partição do *Produto B*. Execuções distribuídas como esta elevam o número de mensagens trocadas entre servidores e, por consequência, aumentam a latência e diminuem a vazão do sistema. Caso a execução distribuída de uma consulta não possa ser evitada, o tamanho das mensagens trocadas entre servidores constitui um fator a ser controlado. Para tanto, o tamanho de fragmentos gerados no processo de particionamento deve refletir um tamanho adequado para troca de mensagens na rede. Minimizar o custo do processamento distribuído através de uma estratégia de particionamento que considera o padrão de acesso das consultas é o foco desta tese.

Existem diversos trabalhos que propõem soluções para o problema da escalabilidade do processamento de consultas sobre repositórios distribuídos e particionados. Porém, existem dois principais problemas envolvidos. O primeiro deles diz respeito à cobertura de

consultas inadequada provida pelas partições geradas. O raciocínio dos principais métodos relacionados baseia-se na estrutura do grafo de dados que representa um BD para gerar partições que acabam por não expressar padrões de consulta da carga de trabalho. Como resultado, o desempenho da consulta diminui quando dados requeridos pela mesma consulta são distribuídos por servidores distintos. Algumas soluções tentam resolver este problema através de um método de replicação de dados. Porém, em sua maioria, a replicação quase total permitida por estas abordagens acaba por tornar grandes conjuntos de dados ainda maiores. O segundo problema envolve o procedimento de particionamento em si. Grande parte dos métodos propostos baseiam-se em algoritmos que requerem analisar todo o grafo que corresponde ao conjunto de dados. Além de impraticável em um contexto em que *petabytes* de dados são assumidos, um BD em nuvem pode ser progressivamente formado à medida que novos servidores ou aplicações são introduzidos no sistema e passam a compartilhar seus dados. Deste modo, um procedimento de particionamento deveria ser acionado sempre que se caracterizar um novo estado de formação do BD.

Com o intuito de superar as deficiências de soluções correntes, esta tese propõe um método para o particionamento de dados baseado na hipótese de que é possível promover a escalabilidade do processamento de consultas sobre bases de dados particionadas de acordo com informações de carga de trabalho aplicadas a estruturas de bancos de dados. A aplicação da carga de trabalho visa produzir partições que ofereçam uma cobertura adequada para consultas, enquanto a estrutura de BD atua como uma visão sumarizada do grafo de dados. Desta forma, a previsão da carga de trabalho sobre a estrutura do BD permite definir uma estratégia de particionamento sem que haja a necessidade de se analisar todo o grafo de dados. Além disto, a estratégia de particionamento previamente definida é capaz de ser aplicada a cada nova porção de dados introduzida no repositório sem que haja necessidade de um reparticionamento.

1.2 Objetivos

O objetivo geral desta tese é definir uma metodologia de particionamento de dados capaz de promover a escalabilidade de sistemas de bancos de dados em ambientes sujeitos a grandes volume de dados e dispostos sobre a arquitetura em nuvem idealizada pela Figura 1.2.

Para atingir este objetivo geral, foram definidos os seguintes objetivos específicos:

— Propor uma abordagem de caracterização da carga de trabalho sobre a qual um banco de dados está sujeito;

— Prover uma solução para o particionamento de dados baseado em informações de carga de trabalho sobre uma estrutura sumarizada de um BD;

— Definir a composição de partições pela fragmentação dos dados em unidades de armazenamento com tamanhos compatíveis ao tamanho adequado para a troca de mensagens na rede.

— Avaliar a solução produzida através de estudos experimentais que efetuam a comparação com trabalhos relacionados.

1.3 Contribuições

Esta tese contribui com uma metodologia de particionamento de dados a ser aplicada por um sistema de banco de dados ofertado como um serviço em nuvem (*Database as Service*). As principais contribuições desta tese são:

— Uma abordagem de caracterização da carga de trabalho capaz de sumarizar a carga prevista sobre um esquema de banco de dados em termos de frequências de execução de consultas e seus respectivos padrões de acesso;

— Um método para o particionamento de dados baseado em heurísticas sobre a caracterização de uma carga de trabalho. O problema do particionamento é tratado

em duas etapas. Na primeira, o processo considera a carga de trabalho juntamente com um limiar de armazenamento para produzir fragmentos de dados compatíveis com tamanhos adequados para a troca de mensagens na rede. No segundo e último passo, fragmentos que estão relacionados pela carga de trabalho são co-alocados nos servidores do repositório de dados distribuído;

— Um sistema, nomeado *ClusterRDF*, que implementa o método de particionamento proposto para *datasets* RDF sobre a arquitetura de processamento idealizada;

— Estudos experimentais que validam o método proposto sobre *datasets* XML e RDF através de comparações com métodos fortemente relacionados.

1.4 Estrutura do Documento

Este trabalho está organizado em mais 6 capítulos. No **Capítulo 2** são apresentadas características de repositórios de dados em *nuvem*, arquiteturas de processamento para sistemas de banco de dados neste contexto e a definição do problema do particionamento de dados a ser tratado por esta tese. O **Capítulo 3** dedica-se à análise de abordagens relacionadas à fragmentação e alocação de dados, juntamente com a apresentação de modelos e técnicas de particionamento de grafos adequadas ao contexto deste trabalho. O **Capítulo 4** apresenta a metodologia de particionamento de dados que atende aos objetivos anteriormente citados. O sistema *ClusterRDF* desenvolvido para simular a execução de consultas sobre a arquitetura idealizada é apresentado no **Capítulo 5**. No **Capítulo 6** são apresentados estudos experimentais que validam a eficácia da solução proposta por esta tese. Por fim, o **Capítulo 7** dedica-se as conclusões deste trabalho e identifica os trabalhos futuros.

CAPÍTULO 2

GERENCIAMENTO DE DADOS EM NUVEM

Cloud computing, ou computação em nuvem, é o termo associado à tendência tecnológica de hospedar um número crescente de aplicações, serviços e dados em *data centers* de grande porte [Agrawal et al., 2013]. *Nuvem* constitui uma metáfora para infraestruturas computacionais que são oferecidas de forma transparente ao seu utilizador. O amadurecimento tecnológico associado à computação em *nuvem* alavancou uma mudança nos aspectos operacionais e econômicos da computação. Esta mudança de paradigma se deve às novas formas de manusear e disponibilizar dados através de arquiteturas distribuídas focadas em prover serviços, de forma que o acesso aos dados seja facilitado e ubíquo [Buyya et al., 2008]. Estes serviços são oferecidos em diferentes níveis, desde a infraestrutura computacional até softwares específicos.

A Infraestrutura como um Serviço (*Infrastructure as a Service* - IaaS) oferece servidores, rede, armazenamento e outros recursos. Seu uso facilita a construção de plataformas e aplicações ao transferir os riscos que envolvem o ambiente para provedores de infraestrutura. Neste contexto, técnicas de virtualização tem se mostrado uma solução efetiva para gerenciar e compartilhar infraestruturas, assim como aplicado por grandes provedores como Google, Amazon e Microsoft [Clark et al., 2005]. Por outro lado, provedores de softwares encontram novos desafios para desenvolver e entregar softwares como um serviço (*Software as a Service*). Ao dispor softwares sobre infraestruturas em nuvem, abre-se a oportunidade para um crescimento imprevisível das aplicações, tanto em relação ao número de usuários, quanto ao volume de dados processado. Neste contexto, garantir a escalabilidade destas aplicações se tornou prioridade e um requisito crítico para operacionalizar os mais diversos tipos de software em nuvem.

Em geral, um sistema é dito escalável quando seu desempenho não degrada após a

adição de recursos computacionais. Existem duas maneiras de escalar um sistema através da adição de recursos. Na primeira abordagem, recursos são adicionados a um servidor que compõe o ambiente, ou então ele é substituído por outro com maior capacidade. Esta prática permite dimensionar o sistema verticalmente, isto é, confere escalabilidade vertical ao sistema subjacente. No entanto, este provisionamento pode se tornar caro e ineficiente, uma vez que sistemas em nuvem estão sujeitos a picos constantes na carga de trabalho. A abordagem alternativa corresponde à adição de recursos de forma horizontal dado pelo incremento de mais servidores de forma transparente. Esta prática permite distribuir a carga entre os diversos servidores, conferindo escalabilidade horizontal aos sistemas. Os custos de infra-estrutura para aumentar a capacidade de forma horizontal são considerados lineares [Baker et al., 2011], tornando-se economicamente mais viável para a construção de infraestruturas computacionais de grande porte. No entanto, essa escala horizontal requer metodologias eficientes para gerenciar estes sistemas distribuídos.

A Figura 2.1 apresenta uma visão geral da pilha de serviços sobre a qual aplicações são construídas em nuvem [Agrawal et al., 2013]. Os usuários destas aplicações se conectam pela Internet através de um *gateway* que redireciona as requisições para servidores apropriados na camada ocupada por servidores de aplicação. Esta camada é responsável por encapsular a lógica da aplicação e processar as requisições de usuário. Para prover acesso rápido a itens de dados frequentemente acessados, um conjunto de servidores na camada de *cache* é responsável por manter e gerenciar o acesso a esses dados. Os dados da aplicação são fisicamente armazenados e gerenciados por um ou mais servidores de banco de dados que compõem a última camada desta pilha. Grande parte das aplicações são dirigidas a dados, o que torna os sistemas gerenciadores de banco de dados a base operacional das aplicações e o componente principal desta pilha de serviços. Portanto, prover um serviço de banco de dados escalável sobre esta infraestrutura constitui um fator crítico para aplicações em nuvem.

A escalabilidade de SGBDs tem sido um desafio para a comunidade de banco de dados por mais de duas décadas [Agrawal et al., 2010]. Os sistemas de banco de dados

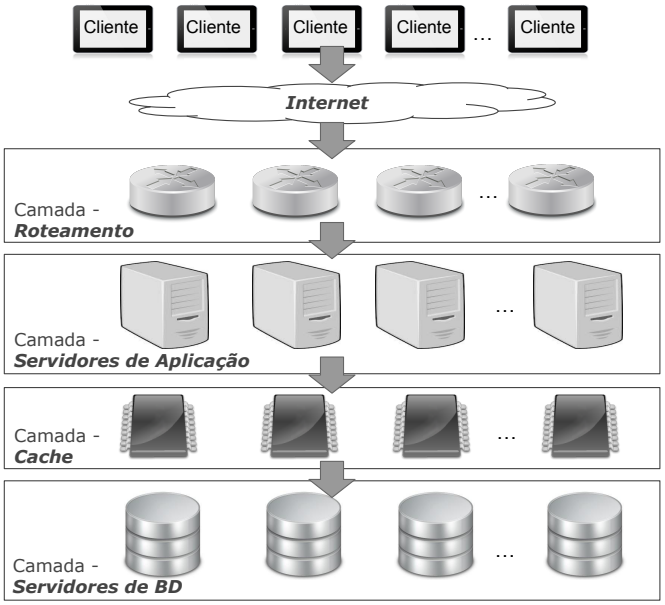


Figura 2.1: Pilha de Serviços para Aplicações em Nuvem

distribuídos foram a primeira solução a tratar o gerenciamento de dados além de uma única máquina. No entanto, grande parte destes sistemas nunca foram extensivamente utilizados na indústria devido a problemas de desempenho causados por falhas, sobrecargas de sincronização e mecanismos de consistência [Agrawal et al., 2009]. Apesar de um SGBD tradicional oferecer uma mistura de simplicidade, flexibilidade, robustez e confiabilidade, algumas dessas funcionalidades resultam no aumento significativo da complexidade quando se tenta assegurá-las em um sistema distribuído em centenas ou milhares de nós. Nos últimos anos observou-se a emergência de uma nova classe de sistemas escaláveis para o gerenciamento de grandes volumes de dados sobre infraestruturas em nuvem. Os sistemas Bigtable da Google [Chang et al., 2008], PNUTS do YAHOO! [Cooper et al., 2008] e Dynamo [DeCandia et al., 2007] da Amazon, são exemplos de sistemas que ganharam destaque neste contexto por habilitar o gerenciamento distribuído de *petabytes* de dados.

Existem diversos fatores que contribuíram para a escalabilidade destes sistemas. Em especial, destaca-se a definição de novos tipos de repositórios de dados sobre modelos de dados mais simples do que aplicados por SGBDs tradicionais. Outro diferencial está relacionado à composição destes repositórios sobre arquiteturas em nuvem. Esta composição,

atrelada à estratégia de particionamento de dados sobre estes repositórios, determina o desempenho do processamento de consultas a partir destes sistemas. Este capítulo discute estes fatores com o objetivo de esclarecer seus respectivos papéis para a definição de um serviço de banco de dados em nuvem. Os conceitos aqui discutidos são empregados para justificar a arquitetura e os componentes sobre os quais uma metodologia para o particionamento de dados é proposta no Capítulo 4.

2.1 Repositório de Dados em *Nuvem*

Segundo Cattell [Cattell, 2011], existem quatro características comuns a repositórios de dados que são escaláveis: (i) são baseados em uma interface simples; (ii) conferem a habilidade de escalar horizontalmente um sistema sobre muitos servidores; (iii) dispõem de uso eficiente de índices distribuídos e de memória; e (iv) permitem um ajuste dinâmico da distribuição da carga de trabalho. Todas estas características estão associadas à natureza *shared-nothing* de redes par-a-par (p2p) empregadas por estes repositórios [Stonebraker, 1986]. Em especial, neste contexto as redes de sobreposição p2p são estruturadas através de uma tabela de dispersão distribuída (DHT - *Distributed Hash Table*) capaz de prover uma interface de uso geral para a indexação, armazenamento e distribuição de dados [Rowstron and Druschel, 2001]. A principal função de uma DHT está associada a sua função de dispersão responsável por particionar um espaço de chaves entre um conjunto de nós distribuídos, semelhante à associação entre chaves e valores de uma tabela de dispersão tradicional. A Figura 2.2 apresenta um exemplo de distribuição de dados através de uma tabela de dispersão distribuída. No exemplo, a chave de um dado é submetida a uma função *hash* cujo resultado endereça um servidor no sistema distribuído no qual o dado deve ser alocado. Aplicações distribuídas baseadas em DHT herdam aspectos de escalabilidade, robustez e facilidade de operação [Stoica et al., 2003].

Sistemas DHT proveêm uma interface genérica com três operações. A operação *put* $(chave, valor)$, armazena um valor associado a uma dada chave em um nó da rede p2p. A

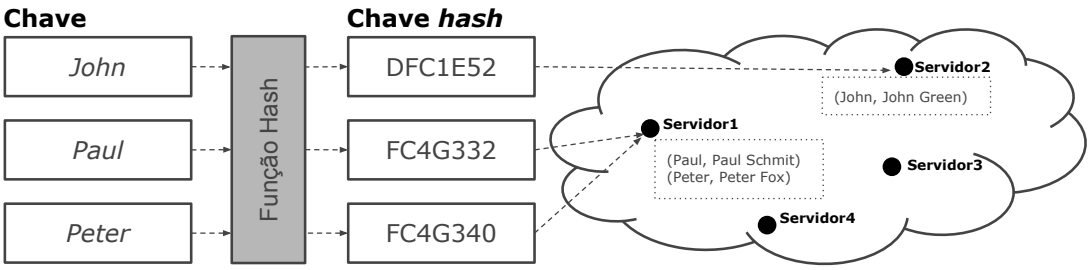


Figura 2.2: Distribuição de Dados através de Tabelas de Dispersão

recuperação de um valor associado a uma chave ocorre através da operação $get(chave)$. A terceira operação, $remove(chave)$, remove um par chave-valor associado à chave dada. Sistemas p2p estruturados formam redes de sobreposição nas quais os tempos de busca e manutenção da DHT necessitam apenas de um número de acessos de ordem logarítmica sobre os nodos que compõem a rede. Por esta razão, aplicações baseadas em redes p2p estruturadas estão entre as aplicações mais escaláveis [Valduriez and Pacitti, 2004]. Repositórios de dados chamados de *NoSQL* em geral são constituídos sobre sistemas p2p estruturados.

Sistemas *NoSQL* estão focados em remover a complexidade de SGBDs tradicionais para alavancar sistemas escaláveis através de modelos de armazenamento mais simples, se comparado aos tradicionais. A simplicidade deste modelo tem influência sobre diversos aspectos do gerenciamento de dados. Um destes aspectos está relacionado ao processamento de consultas. Uma comparação entre modelos de armazenamento tradicionais e *NoSQL* para dados RDF é apresentada em [Zeng et al., 2013]. No modelo tradicional, dados RDF são armazenados como um conjunto de triplas em tabelas relacionais. O RDF-3X [Neumann and Weikum, 2009] é um exemplo deste tipo de sistema que ficou conhecido como *triple store*. Na maioria dos casos, esta forma de armazenamento leva a operações de junção custosas no processamento de consultas. Isto porque os casamentos para padrões de triplas são gerados separadamente e depois combinados por junções. Este procedimento gera uma quantidade expressiva de resultados intermediários na fase de casamentos de triplas, e que depois são descartados após as junções. O estudo experimental apresentado por [Zeng et al., 2013] mostra que este problema limita a escalabilidade

do processamento de consultas sobre *triple stores*, mesmo sobre uma arquitetura paralela e distribuída. Este mesmo estudo aponta que é possível habilitar um processamento de consulta escalável quando dados são armazenados por um modelo mais simples, neste caso, o modelo chave-valor. Os modelos de armazenamento empregados e as arquiteturas para repositórios em *nuvem* são apresentados pelas seções a seguir.

2.1.1 Modelos de Armazenamento

Embora a grande maioria dos repositórios de dados em *nuvem* dê suporte ao armazenamento de dados complexos, eles diferem na estrutura de armazenamento. Os modelos de armazenamento podem ser classificados em modelos chave-valor, documentos ou registros extensíveis [Cattell, 2011]. O modelo *chave-valor* é a forma de representação mais aproximada àquela imposta por uma rede de sobreposição baseada em DHT. Neste caso, a unidade de armazenamento é um par chave-valor, no qual a chave atua como um índice para endereçamento do valor associado. O modelo de *documentos* é similar ao modelo chave-valor, mas neste caso o valor pode ser estruturado, por exemplo através um conjunto de pares chave-valor. Os documentos são indexados por uma única chave e organizados em domínios. Por fim, os *registros extensíveis* permitem um aninhamento entre os dados através dos conceitos de colunas e super-colunas.

A Figura 2.3 mostra exemplos para os modelos de dados apresentados. Os sistemas Voldemort [Voldemort, 2012], Dynamo [DeCandia et al., 2007], Riak [Klophaus, 2010], Redis[1], Scalaris [Zuse Institute Berlin, 2012] e Bamboo[2] são exemplos de repositórios chave-valor. Dentre os repositórios que aderem ao modelo de documentos, pode-se citar o SimpleDB [AWS, 2014], CouchDB[3] e MongoDB[4]. Por fim, HBase[5], HyperTable[6] e Cassandra [Lakshman and Malik, 2009] são exemplos de repositórios com registros extensíveis.

[1] http://redis.io/
[2] http://www.bamboodb.com/
[3] http://couchdb.apache.org/
[4] http://www.mongodb.org/
[5] http://hadoop.apache.org/hbase/
[6] http://www.hypertable.org/

```
(1, "(RG, 123),(nome, João)")
(2, "(RG, 456),(nome, Maria)")
(2, "(RG, 789),(nome, Pedro)")
```

(a) Chave-valor

```
Domínio Aluno
(1, {(RG,123),(nome, João)})
(2, {(RG, 456),(nome, Maria)})
(3, {(RG, 789),(nome, Pedro)})
```

```
Domínio Curso
(1, {(nome, CC)})
(2, {(sigla, EE)})
```

(b) Documento

Aluno	coluna		super-colunas			
A1	nome	curso	A1BD		A1EE	
	João	CC	disc	nota	disc	nota
			BD	95	EE	78
An	...					

(c) Registro Extensível

Figura 2.3: Modelos de Dados para Repositórios na *Nuvem*

O modelo chave-valor é um modelo simples e flexível [Cattell, 2011], o que é ideal para um modelo físico neste contexto. Repositórios chave-valor são frequentemente associados ao termo *NoSQL* por apresentarem um modelo simples para garantir melhor desempenho e flexibilidade de representação. No entanto, como um modelo lógico, ele é pouco expressivo e inadequado para representar a complexidade dos dados provenientes da camada de aplicação. A simplicidade da representação do modelo *chave-valor* também se estende para os baseados em *documentos* e *registros extensíveis*, pois repositórios baseados nestes modelos apresentam uma interface bastante simples e de pouca expressividade. Esta simplicidade gera a necessidade de implementar certas operações no nível da aplicação. Para maior expressividade das operações sobre estes repositórios, alguns trabalhos propõem o mapeamento destes modelos de implementação e suas respectivas operações para um modelo lógico de alto nível. A ideia de desenvolver uma interface sobre repositórios existentes tem sido adotada por diversos trabalhos [Arnaut et al., 2011, Curino et al., 2011, Bunch et al., 2010, Egger, 2009, Cooper et al., 2008]. Em [Arnaut et al., 2011] é demonstrado que o custo em se manter uma camada relacional aplicada sobre o repositório chave-valor Scalaris é mínimo.

Considerando que grande parte das aplicações que utilizam SGBDs escaláveis são orientadas à Web, assume-se que XML e RDF sejam modelos adequados para atuar no nível lógico. Além de constituírem padrões para o intercâmbio e publicação de dados na Web, a flexibilidade destes modelos permite representar uma grande variedade de

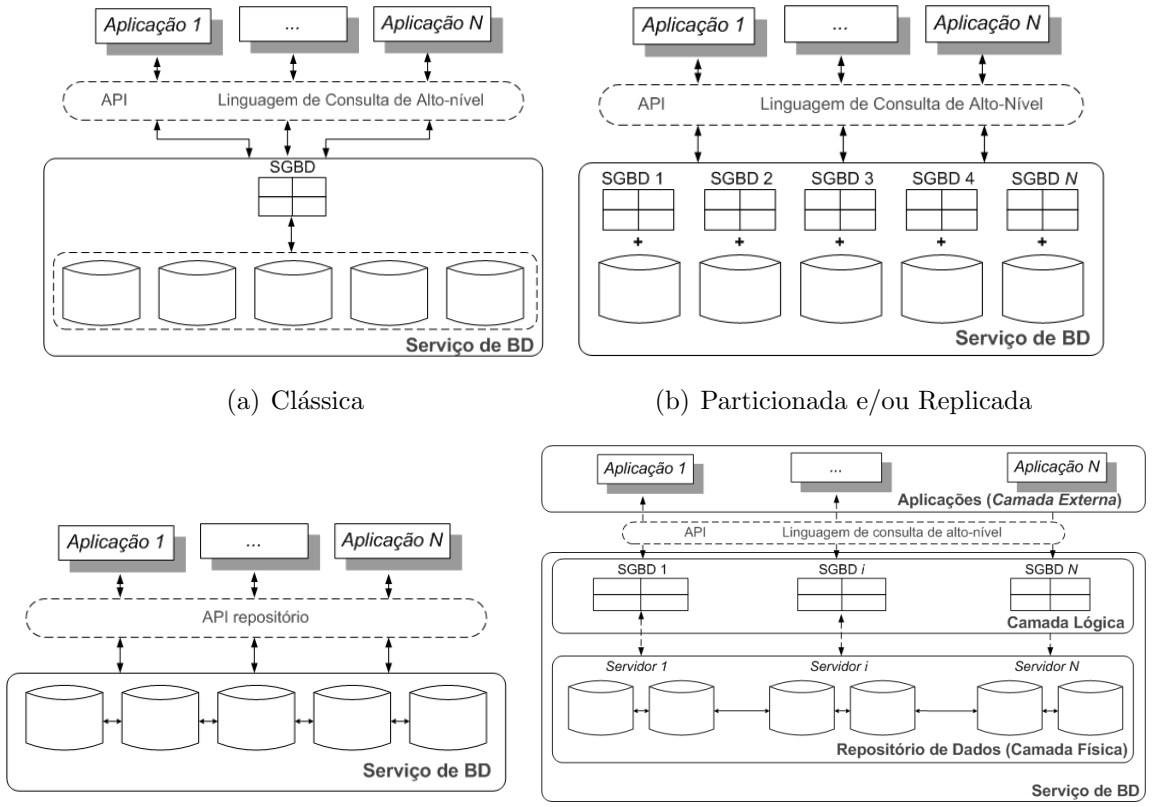
informações. Assim, para manter a simplicidade do modelo físico e a expressividade e flexibilidade no nível da aplicação, considera-se que dados definidos através destes modelos podem ser adequadamente convertidos para um destes modelos de armazenamento.

2.1.2 Arquiteturas para Repositórios de Dados em Nuvem

Segundo [Kossmann et al., 2010], existem 3 variações de arquiteturas que utilizam repositórios de dados em *nuvem*. As três primeiras ilustrações da Figura 2.4 representam as arquiteturas definidas em [Kossmann et al., 2010]. A primeira delas corresponde à arquitetura *clássica* em que um servidor de BD centralizado é responsável por processar e submeter as operações a um sistema de armazenamento escalável. Embora as principais propriedades de um SGBD tradicional possam ser asseguradas sobre um sistema de armazenamento escalável através do controle centralizado do SGBD, o servidor de BD representa um gargalo quando submetido a sobrecargas. Neste caso, a solução para evitar a sobrecarga é tentar escalar o sistema verticalmente, isto é, substituir o servidor por uma máquina de maior poder computacional. No entanto, este provisionamento pode se tornar caro e ineficiente, uma vez que sistemas em *nuvem* estão sujeitos a picos constantes na carga de trabalho. O AWS MySQL e o AWS RDS [Amazon, 2014] são exemplos de sistemas que empregam esta arquitetura clássica.

Uma arquitetura alternativa para eliminar este ponto de contenção é a utilização de técnicas de particionamento e replicação de bancos de dados distribuídos sobre diversos SGBDs. Nesta solução, cada SGBD poderá executar sobre máquinas menos custosas para gerenciar um sub-conjunto do BD e assim sustentar a carga. No entanto, variações na carga de trabalho ou alterações na composição da rede podem levar a um remanejamento custoso das partições de dados entre os SGBDs. Além disto, operações de atualização envolvendo bases replicadas poderão requerer um controle centralizado de réplicas principais para manter as demais réplicas consistentes. Neste caso, ainda assim existe um ponto de contenção capaz de limitar a escalabilidade do sistema. Como exemplos de sistemas que aplicam este tipo de arquitetura, pode-se citar o MySQL/R e SimpleDB da

Amazon Web Services [Amazon, 2014], assim como o Google AppEng [Google, 2012] e o MS Azure [Sengupta, 2008].



(a) Clássica (b) Particionada e/ou Replicada

(c) Controle Distribuído (d) Arquitetura Estratificada

Figura 2.4: Arquiteturas para o Processamento de Transações na *Nuvem*

A terceira e última arquitetura relacionada em [Kossmann et al., 2010], refere-se a uma solução encontrada para escalar estas aplicações através de uma composição do tipo *shared-nothing*, onde o controle do sistema ocorre de forma completamente distribuída sobre nós independentes e auto-suficientes. Através deste tipo de composição é possível atingir escalabilidade quase ilimitada pela adição de novos nós à rede [Baker et al., 2011]. Escalar um sistema horizontalmente é mais barato do que substituir os recursos computacionais vigentes. Em sistemas em *nuvem*, este conceito é interessante pois adere ao conceito *on-demand* e *pay-as-you-go* proposto por estes ambientes ou serviços. No entanto, os sistemas existentes que são baseados em controle distribuído apresentam um conjunto de funcionalidades bastante limitada. O S3 [Brantner et al., 2008] é um exemplo deste tipo de sistema de armazenamento escalável.

A quarta arquitetura da Figura 2.4 representa uma extensão da arquitetura com controle distribuído que considera as camadas de um SGBD [Arnaut et al., 2011]. Nesta composição, as características de um SGBD tradicional, como uma interface de alto-nível, são asseguradas por um conjunto de SGBDs distribuídos e que gerenciam de forma distribuída e independente um sistema de armazenamento escalável. Esta arquitetura estratificada oferece independência de dados física, permitindo que diferentes abordagens para mapeamento lógico-físico sejam empregadas, enquanto o repositório de dados distribuído é responsável por fornecer escalabilidade, disponibilidade e tolerância a falhas. Esta independência de modelos também habilita o emprego de diferentes estratégias de particionamento de dados sobre os modelos lógico e físico adotados. A seção a seguir apresenta detalhes sobre o processo de consultas sobre repositórios de dados particionados e baseada em arquiteturas com controle distribuído.

2.2 Processamento de Consultas sobre Repositórios em Nuvem

Em repositórios particionados, cada servidor detém uma porção da base de dados denominada *partição*. O grafo da Figura 2.5 representa um conjunto de dados que relaciona dados de produtos (*product*) e suas respectivas características (*prFeature*). Como exemplo, assume-se que este grafo foi particionado sobre um repositório distribuído constituído de 3 servidores na Figura 2.6. As arestas pontilhadas correspondem a cortes efetuados sobre o grafo de dados que deram origem às partições. O processamento de consultas sobre estes repositórios pode ser determinado através de buscas paralelas sobre cada partição. Como exemplo, considere que uma consulta requer obter a *label* de nós do tipo *product* que venceram no mês de Maio, isto é, *dueDate* = "2014-05". Para obter a recuperação dos dados que respondem a esta consulta, uma linha de execução, ou *thread*, pode ser estabelecida de forma concorrente sobre cada servidor. Na Figura 2.6, cada seta numerada corresponde a cada uma destas *threads*. Neste exemplo, a data de vencimento de cada produto é avaliada em paralelo de acordo com a consulta. Deste modo, as partições

contidas nos servidores 1 e 2 retornam os respectivos rótulos (*label*) de seus produtos visto que ambos fazem parte do resultado final. Esta abordagem é amplamente utilizada sobre repositórios em nuvem, especialmente devido ao volume de dados envolvido neste tipo de recuperação [Jiewen Huang, 2011].

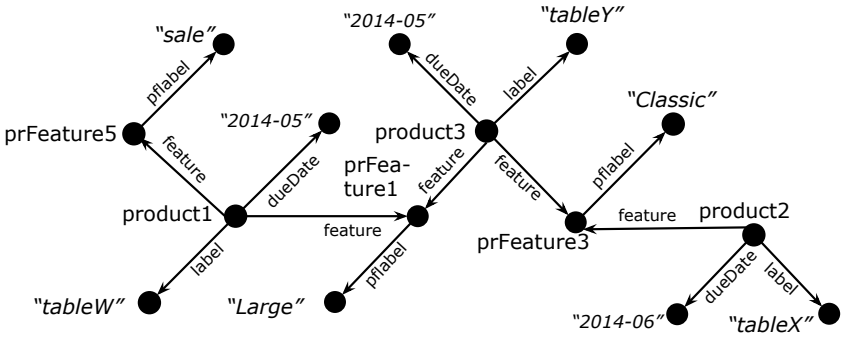


Figura 2.5: Grafo de Dados [Bizer and Schultz, 2009]

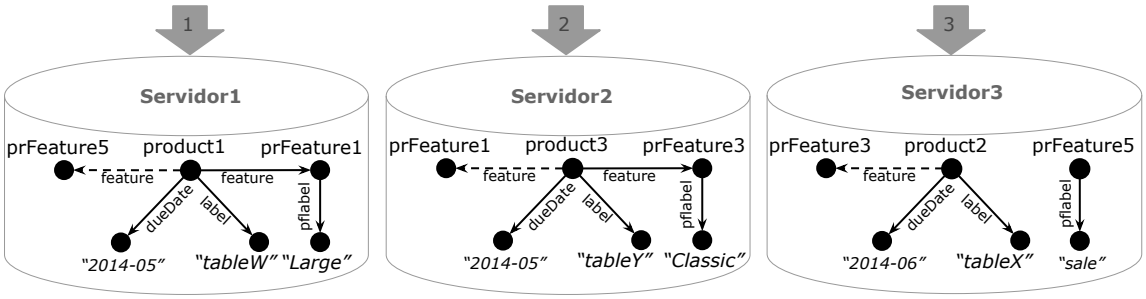


Figura 2.6: Processamento Paralelo sobre Repositório Particionado

Apesar dos benefícios da execução paralela, a troca de dados entre servidores pode ser necessária em algum momento. Considere novamente a mesma base de dados particionada e uma nova consulta que requer listar os dados de todas as características de um produto (*prFeature*). Observe que algumas características foram separadas de seus produtos em virtude de cortes efetuados sobre o grafo de dados. Neste caso, trocas entre servidores serão necessárias para obter dados que estão fora do limite das partições, como mostra a Figura 2.7. Por exemplo, a *thread* iniciada pelo *Servidor 1* necessitará comunicar-se com o *Servidor 3* para obter dados de *prFeature5*. A mesma situação ocorre para os demais servidores, conforme indicado pelas setas numeradas entre servidores. Em geral, este encadeamento pode ser gerenciado por mecanismos de processamento específi-

cos controlados pelos próprios SGBDs, ou por *frameworks* de processamento baseados em *MapReduce* [Zeng et al., 2013]. Para ambas soluções, trocas como estas podem comprometer o desempenho do processamento de consultas e anular os benefícios da paralelização.

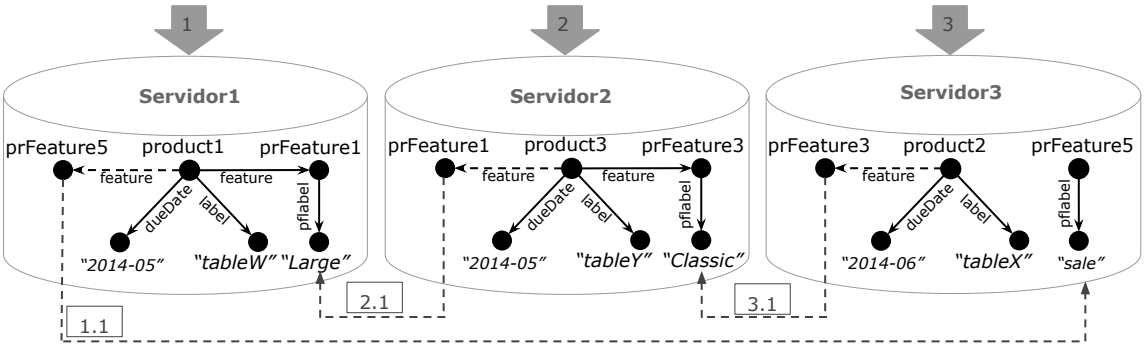


Figura 2.7: Processamento Paralelo com Troca de Dados

Quando a distribuição do processamento de consultas é necessária como no exemplo, seu desempenho é afetado pelo custo da comunicação e pelo tamanho das mensagens transmitidas nestas trocas [Ozsu and Valduriez, 2011] [Fan, 2012]. Do ponto de vista do projeto físico de um BD, o custo da comunicação pode ser neutralizado pela definição de uma estratégia de particionamento adequada em conjunto com otimizações relacionadas às consultas [Ozsu and Valduriez, 2011]. Quanto ao tamanho das mensagens transmitidas, considera-se que o conjunto de dados deve ser fragmentado de tal forma que o tamanho destes fragmentos corresponde a um tamanho de mensagem adequado para a troca de mensagens na rede. Esta tese está focada na definição de uma metodologia de particionamento capaz de minimizar estes custos. Uma carga de trabalho prevista para o BD é considerada para a definição da estratégia de particionamento. Um conjunto de dados é fragmentado em unidades de armazenamento que correspondem a um tamanho adequado para as mensagens que eventualmente podem ser trocadas entre servidores do sistema. Fragmentos relacionados são agrupados e alocados em servidores para formar as partições do repositório. A Figura 2.8 apresenta o particionamento recomendado por esta metodologia com relação ao exemplo anterior. Neste caso, um limiar de 3 nós foi assumido como capacidade máxima para determinar o tamanho de fragmentos. Observe

que os fragmentos estão representados por formas pontilhadas e alocados em conjunto com fragmentos relacionados em um mesmo servidor. Nesta composição, a recuperação dos produtos e suas características podem ser obtidos de forma concorrente sobre cada uma das partições. A replicação de dados é habilitada para corresponder às intenções de busca das principais consultas de uma carga de trabalho.

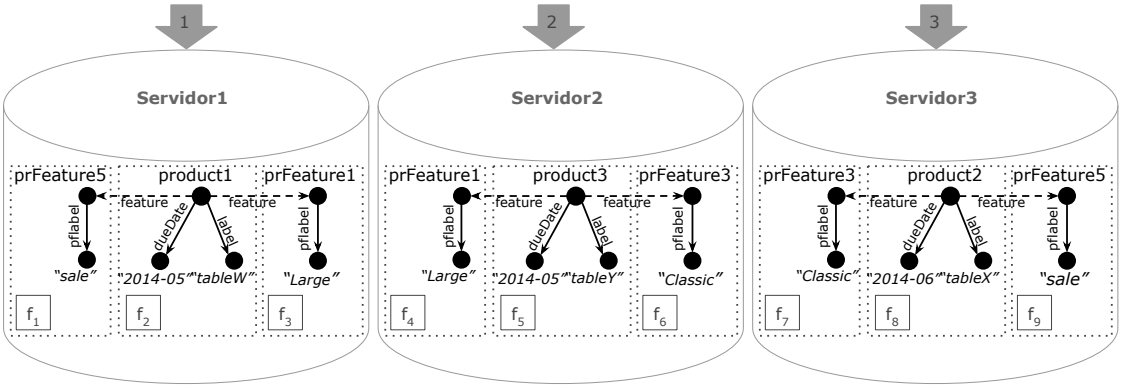


Figura 2.8: Fragmentação e Alocação de Dados

Quando um conjunto de dados pode ser representado como um grafo de dados assim como mostrado pela Figura 2.5, diversos algoritmos de particionamento de grafos podem ser aplicados. Entretanto, trata-se de um problema NP-difícil [Guttmann-Beck and Hassin, 2000]. Neste sentido, soluções que se baseiam em heurísticas poderão ser utilizadas para evitar a exaustão do processo. Além do volume de dados envolvido, um problema específico de repositórios em nuvem está relacionado à volatilidade de servidores que compõe o sistema distribuído. Por exemplo, um BD em nuvem pode ser progressivamente formado à medida que novos servidores ou aplicações são introduzidos no sistema e passam a compartilhar seus dados. Desta forma, um novo procedimento de particionamento deve ser acionado sempre que se caracterizar um novo estado de formação do BD. Neste sentido, é importante que a estratégia de particionamento não esteja baseada na formação completa do grafo de dados. O capítulo seguinte discute estas questões e trata do estado da arte com relação a abordagens aplicadas ao contexto geral de particionamento de dados, bem como as destinadas para composições em nuvem e para modelos de dados específicos.

CAPÍTULO 3

ABORDAGENS PARA O PARTICIONAMENTO DE DADOS

Este capítulo apresenta alguns dos principais trabalhos que propõem soluções para o problema de particionamento de dados. Soluções genéricas para o particionamento de grafos são inicialmente introduzidas e relacionadas ao contexto de banco de dados. Na sequência, abordagens dedicadas aos modelos Relacional, XML e RDF são apresentadas e comparadas. A análise destes trabalhos está focada em identificar aspectos que possam contribuir para a solução do problema que esta tese se propõe a resolver. O objetivo de cada solução é evidenciado para determinar suas limitações e aplicabilidade às especificidades de seus modelos de dados. O processo de particionamento é igualmente avaliado e relacionado ao contexto de repositórios em nuvem.

3.1 Particionamento de Grafos

De forma genérica, um banco de dados e sua respectiva carga de trabalho de consultas podem ser representados por um grafo não-direcionado definido por $G = (V, E, c)$, onde (i) V é o conjunto de nós que representam dados, (ii) E é o conjunto de arestas que relacionam pares de nós acessados em conjunto pela carga de trabalho, e (iii) $c : E \rightarrow \mathbb{N}$ define o peso de cada aresta que representa a quantidade de vezes que um par de nós é acessado em conjunto. O particionamento de um banco de dados pode ser definido como um problema de corte em grafos com a finalidade de gerar com conjunto de partições dado por $P = \{p_1, p_2, ..., p_k\}$, tal que $p_i \subseteq V$ e $\bigcup_{i=1}^{k}(p_i) = V$ e $p_i \cap p_j = \oslash$ para todo $i \neq j$.

Embora existam diferentes modelos associados ao problema de corte em grafos, esta seção dedica-se ao problema do corte mínimo. O custo de cortes em G definido por um particionamento P é dado por $custo(P) = \sum c(v_1, v_2)$, tal que $v_1 \in p_i$ e $v_2 \in p_j$ para todo $i \neq j$. Ao considerar os problemas do processamento de consultas relatados

pela Seção 2.2, intuitivamente, ao minimizar o custo do corte minimiza-se a troca de mensagens entre servidores distintos para responder uma dada consulta, uma vez que dados frequentemente acessados em conjunto serão mantidos em uma mesma partição. Neste capítulo são descritos modelos de corte mínimo para grafos, bem como algoritmos para os problemas relacionados.

3.1.1 Modelos de Corte Mínimo

O problema base de corte mínimo de grafos é conhecido como *s-t Minimum Cut*, que visa particionar um grafo em dois subconjuntos que separam um nó origem ($s \in V$) e um nó destino ($t \in V$) de forma que o custo do corte seja mínimo. O algoritmo *Ford-Fulkerson* [Ford and Fulkerson, 1956] é capaz de determinar o corte mínimo computando o fluxo máximo entre s e t. Através do teorema *max-flow min-cut*, Ford e Fulkerson provaram que o valor do fluxo máximo é igual ao valor do corte mínimo [Papadimitriou and Steiglitz, 1998]. Este problema está relacionado a problemas de multi-corte, e que constitui um problema NP-difícil para $k \geq 3$ onde $k = |P|$ [Dahlhaus et al., 1994].

Existem quatro classificações principais para problemas de multi-corte mínimo: *Minimum Multicut*, *k-Multicut*, *Multiway cut* e *k-cut*. O objetivo do problema *Minimum Multicut* segue o mesmo princípio do problema *s-t Minimum Cut*. Seja um grafo G e $A = \{(s_1, t_1), ..., (s_k, t_k)\}$ um conjunto específico de pares de nodos origem-destino distintos, o problema é determinar um conjunto de arestas cuja remoção separa cada um dos pares de nodos em A, gerando um particionamento P onde o valor de $custo(P)$ é minimizado. Um problema similar é o *Multiway cut*, cuja a finalidade é encontrar um corte mínimo que separa um conjunto de terminais.

O problema *k-Multicut* pode ser reduzido ao problema de *Minimum Multicut*, porém neste caso o número de partições geradas deve ser igual ou superior a k, ou seja, $|P| \geq k$. O problema que requer encontrar um conjunto de arestas cuja a remoção deixa k fragmentos é denominado *k-cut* ou *k-way*. O objetivo é encontrar o valor de k ($1 < k \leqslant |V|$) para o qual o valor de $custo(P)$ é mínimo. Dentre os modelos de corte mínimo apresentados,

k-cut é o que mais se aproxima ao problema de particionamento em banco de dados. Neste contexto, deseja-se obter um conjunto de partições em que $|P|$ é inicialmente desconhecido.

3.1.2 Técnicas de Particionamento

Para resolver o problema do corte mínimo, um grafo pode ser particionado por bisseções recursivas ou por uma estratégia de particionamento em múltiplos níveis, conforme descrito a seguir. No paradigma de bisseções recursivas, um grafo é recursivamente dividido em duas partições até que o número de partições desejado seja obtido, ou um custo de corte adequado. Em cada passo, são aplicadas heurísticas para refinar as bisseções e diminuir o custo do corte. No entanto, a execução destes refinamentos iterativos tende a deteriorar o resultado, gerando soluções não aproximadas à solução ótima uma vez que não possuem uma visão global do problema [Karypis and Kumar, 1999, Aykanat et al., 2008]. Diversos trabalhos provam que o particionamento em múltiplos níveis produz melhores resultados que bisseções recursivas para diversos tipos de aplicação do problema [Karypis and Kumar, 1999, Çatalyürek and Aykanat, 1999, Abou-Rjeili and Karypis, 2006].

O particionamento em múltiplos níveis obtém uma sequência de aproximações sucessivas do grafo original a fim de reduzir o problema e, em seguida, refinar a solução para encontrar a solução mais aproximada. O processo compreende 3 etapas. A fase de *coarsening* executa uma sequência de aproximações (reduções) sucessivas do grafo. Em seguida, na fase de *particionamento inicial* o grafo reduzido é particionado. Na etapa final, o particionamento encontrado é então refinado sequencialmente para níveis em que a granularidade do grafo vai se tornando cada vez mais fina. Em cada nível, um algoritmo de refinamento iterativo é executado para melhorar a qualidade do particionamento.

Na fase de *coarsening*, pares de vértices adjacentes são sequencialmente combinados por um método de *matching*. As combinações identificadas determinam a fusão de nós que visam a redução do grafo. Dentre estes métodos, destaca-se o método aleatório e o *heavy edge*. No método aleatório, a fusão ocorre entre nodos adjacentes que se conectam aos mesmos nodos, ou seja, suas respectivas listas de adjacência são equivalentes [Abou-Rjeili

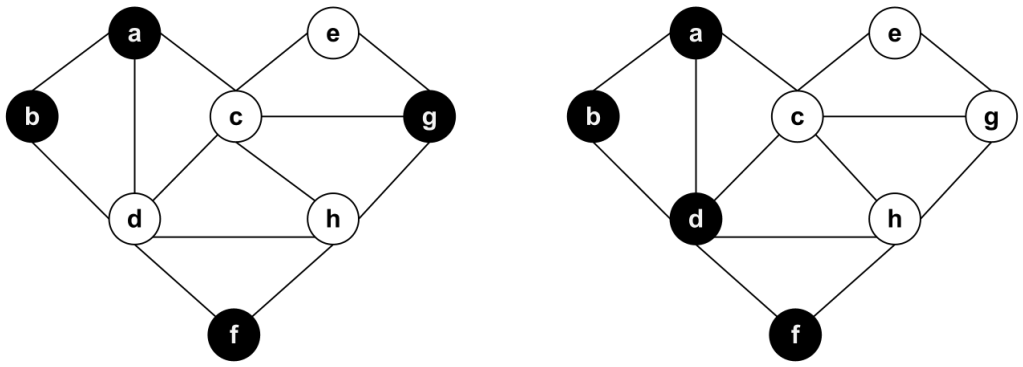
and Karypis, 2006]. Diferentemente, o método *heavy edge* seleciona as arestas de maior peso para executar a fusão dos nodos conectados por elas [Karypis and Kumar, 1998b]. Na fase do *particionamento inicial*, algoritmos de bisseções podem ser executados. A seguir, são apresentados alguns destes algoritmos bem como algoritmos de refinamentos aplicados na última fase do particionamento em múltiplos níveis.

Existem diversos algoritmos que fornecem soluções para problemas de corte mínimo através de bisseções recursivas. Dentre as soluções para o problema de *k-cut*, destaca-se o algoritmo definido em [Saran and Vazirani, 1995], cuja complexidade de execução é baseada no processamento de fluxos máximos. Este trabalho estende o problema para considerar a restrição de balanceamento da quantidade de nós nos fragmentos gerados. Entretanto, trata-se de uma solução pouco aproximada para o problema de k-cut balanceado se comparada a uma solução equivalente proposta em [Guttmann-Beck and Hassin, 2000]. O algoritmo proposto para resolver o problema do *s-t Cut* em [Galbiati, 2011], considera pesos atribuídos aos nodos do grafo e um limiar B que determina o peso máximo para a partição s a ser gerada.

O surgimento da classe de algoritmos de refinamento foi iniciada pelos algoritmos de *Kernighan-Lin e Fiduccia-Mattheyses* (KL e FM). Dado um grafo com um particionamento preliminar, o problema é melhorar a qualidade da partição enquanto mantém a restrição de balanceamento referente à quantidade de nós em cada partição. Portanto, dada uma bisseção de um grafo que separa os vértices nos conjuntos V' e V'', deseja-se encontrar dois sub-conjuntos de igual tamanho, R' e R'' a partir de V' e V'', respectivamente, de tal forma que substituindo-se nós entre V' e V'' diminui-se o custo do corte do grafo. Este processo pode ser efetuado repetidas vezes até que não seja mais possível diminuir o custo do corte. Os algoritmos KLFM trabalham sobre heurísticas para encontrar os sub-conjuntos R' e R''.

A Figura 3.1 exemplifica o processo efetuado pelos algoritmos de refinamento do tipo KLFM. O particionamento preliminar apresentado pela Figura 3.1(a) divide o grafo em dois conjuntos de vértices V' e V'', V' representado pelos nós com preenchimento e V''

pelos nós sem preenchimento. O custo do corte é dado pela quantidade de arestas entre V' e V''. No caso do particionamento preliminar, o custo do corte é igual a 8 arestas. Um algoritmo do tipo KLFM efetua a troca dos nós d e g entre as partições e assim reduz o custo do corte para 4 arestas no grafo refinado da Figura 3.1(b).



(a) Particionamento Preliminar (b) Particionamento Refinado

Figura 3.1: Refinamento KLFM

O algoritmo KL [Kernighan and Lin, 1970] efetua trocas de todos os vértices entre as partições V' e V''. Durante cada passo, o algoritmo tenta encontrar um par de vértices $a \in V'$ e $b \in V''$ cuja troca entre as partições pode reduzir o custo do corte. Para determinar a viabilidade da troca, alguns custos são previamente calculados. O custo externo de um nó $a \in V'$ é dado por $E(a) = \sum_{j \in V''} c(a, j)$, onde $c(a, j)$ corresponde ao custo do corte que separa a e j. O custo interno de um nó é dado por $I(a) = \sum_{j \in V'} c(a, j)$ e a diferença entre eles é dada por $D(a) = E(a) - I(a)$. Estas mesmas medidas são igualmente calculadas para b. Por fim, o ganho em se trocar os vértices a e b é dado por $g(a, b) = D(a) + D(b) - 2.c(a, b)$. Neste caso, escolhe-se um par de vértices a e b cujo valor de $g(a, b)$ seja máximo. A troca é efetuada e o mesmo processo é repetido para os demais pares de V' e V'' até que todos os pares sejam movidos. Em cada troca, contabiliza-se o custo geral do corte do grafo para que no final o corte mínimo seja identificado. Na finalização deste passo, a composição do grafo correspondente ao corte mínimo é recuperada. Após finalizada uma execução do algoritmo KL, outra execução pode então ser iniciada no contexto da etapa de refinamento, em que a bisseção resultante é utilizada

como entrada. O passo inicial do algoritmo possui complexidade de tempo $\Theta(n^2)$, onde n é a quantidade de nós. A busca pelos melhores ganhos na troca de pares custa $\Theta(n^2 log\ n)$, conforme demonstrado em [Kernighan and Lin, 1970].

O algoritmo *Fiduccia-Mattheyses* (FM) [Fiduccia and Mattheyses, 1982] é considerado um melhoramento do KL. Para casos gerais, o FM é capaz de reduzir o tempo de execução utilizando estruturas de dados mais apropriadas. O algoritmo apresenta tempo de execução de $\Theta(nlogn + |E|)$ e apresenta a mesma efetividade do algoritmo KL. No entanto, para casos específicos o FM pode gerar um corte maior e desempenho inferior que o apresentado pelo KL. Diversos algoritmos foram derivados das abordagens de KL e FM, dentre eles destaca-se uma combinação das abordagens conhecida como KLFM [Hauck and Borriello, 1995].

Existem diversas aplicações que se beneficiam de técnicas de particionamento baseadas em corte mínimo em grafos, mas que estão sujeitas a alterações frequentes em seu grafo de dados. Neste contexto, um procedimento de reparticionamento deve ser aplicado periodicamente para manter a qualidade do particionamento. Segundo Catalyurek U. et. al. [Catalyurek et al., 2009], abordagens dinâmicas baseadas em corte mínimo em grafos podem ser classificadas em três categorias principais: *scratch-remap*, *incremental* e *reparticionamento*. Na abordagem *scratch-remap*, o grafo é particionado do início, isto é, sem considerar o particionamento anterior como referência. Em contrapartida, a estratégia *incremental* preza por considerar o particionamento inicial e escolher entre minimizar ou o custo da migração de dados ou o custo do corte do grafo de dados. Por fim, na abordagem de *reparticionamento* ambos os objetivos são considerados em conjunto.

No contexto de banco de dados, grande parte das soluções que aderem às categorias *incremental* ou de *reparticionamento* são baseadas em replicação [Kossmann, 2000]. Uma solução proposta em [Yang et al., 2012] aplica replicações em fragmentos para acomodar o acesso não-uniforme a dados e gera fragmentos complementares para ajustar-se a alterações na carga de trabalho. Abordagens similares são apresentadas em outros trabalhos como [Pujol et al., 2010] e em [Brocheler et al., 2010]. Porém, a geração de novos frag-

mentos e a replicação destes pode se tornar prejudicial para o desempenho do sistema, uma vez que ocorre um aumento significativo de volume do BD e de dados replicados. Existem algumas outras soluções que preocupam-se em monitorar e identificar a necessidade de reavaliar o esquema de particionamento de um BD, mas acabam por executar um processo *scratch-remap*. Um exemplo é a ferramenta PIXSAR [Shnaiderman et al., 2008] que atualiza periodicamente as afinidades sobre conexões de elementos pai-filho e irmãos de documentos XML de acordo com as transações que estão sendo executadas. Quando detecta-se que a qualidade do particionamento foi prejudicada, o algoritmo de particionamento proposto em [Bordawekar and Shmueli, 2008] é reaplicado sobre toda a coleção de documentos XML. Certamente, o custo de iniciar um novo processo de particionamento é impraticável em BDs que processam um grande volume de dados [Hauglid et al., 2010]. Sobretudo, a migração de dados em BDs distribuídos é um procedimento caro não só pelo volume de dados, mas por mecanismos de consistência que devem ser utilizados para garantir a integridade dos dados durante o processo.

Como mencionado anteriormente, o problema do particionamento de um BD pode ser endereçado como um problema de otimização de corte mínimo em grafos. Para o contexto de BDs em nuvem, a formação progressiva do banco de dados poderá comprometer a todo instante a qualidade do corte. Embora técnicas de reparticionamento estejam disponíveis, este procedimento se mostra prejudicial para um contexto em que grandes volumes de dados desafiam a escalabilidade das aplicações a todo instante. A seguir, são apresentadas abordagens de particionamento propostas para o projeto físico de banco de dados. Uma prática comum neste contexto é a utilização de heurísticas relacionadas à estrutura de dados e à carga de trabalho do BD.

3.2 Abordagens para o Particionamento de Banco de Dados

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 razão

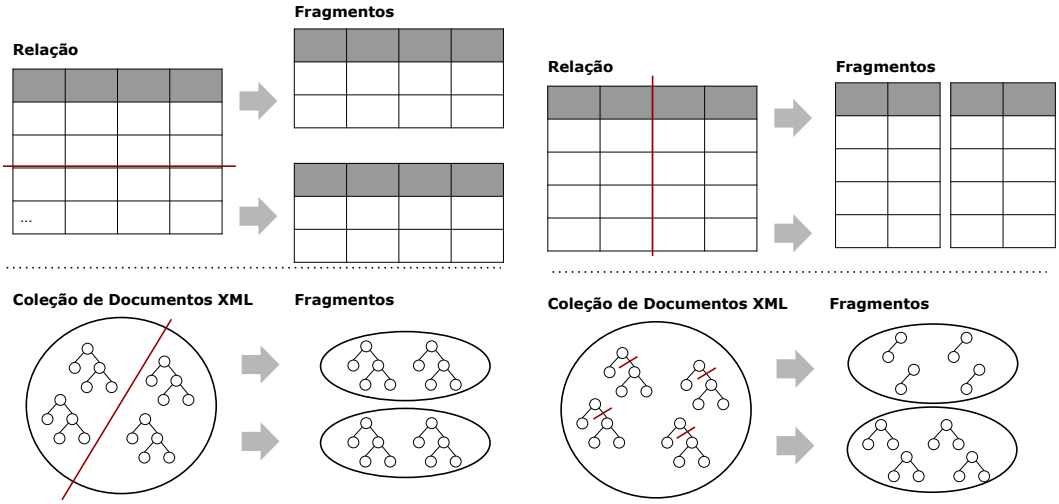
para esta divisão em dois passos está relacionada ao melhor tratamento da complexidade do problema [Ozsu and Valduriez, 2011]. A fragmentação divide o banco de dados em fragmentos de acordo com um modelo de dados específico. Em seguida, a alocação de dados envolve encontrar uma distribuição adequada para os fragmentos no repositório enquanto satisfaz restrições envolvendo o tempo de resposta, armazenamento e processamento. Para BDs centralizados, estratégias de particionamento são adotadas para minimizar o número de acesso a discos, enquanto que em um sistema distribuído elas são aplicadas para minimizar o número de operações distribuídas que estão associadas ao custo da comunicação de dados na rede. Nesta seção são apresentadas abordagens para solucionar o problema da fragmentação e da alocação de dados para os modelos Relacional, XML e RDF.

3.2.1 Fragmentação de Dados

A fragmentação adequada de dados em sistemas distribuídos pode melhorar a vazão do sistema e o desempenho de consultas. Estes objetivos podem ser atingidos especialmente pela redução do número de fragmentos requeridos pelas consultas e a consequente transmissão de dados entre nodos [Son and Kim, 2004]. Além disto, com uma alocação apropriada destes fragmentos, a vazão do sistema pode ser melhorada em termos do processamento paralelo de consultas sobre fragmentos alocados em servidores distintos [Ozsu and Valduriez, 2011, Sacca and Wiederhold, 1985].

Os modelos de distribuição de dados geralmente dão suporte à fragmentação horizontal e vertical. Em bancos de dados relacionais, a abordagem tradicional gera fragmentos que possuem um sub-conjunto de tuplas, enquanto a fragmentação vertical produz fragmentos que contêm um sub-conjunto de atributos das relações. Na fragmentação horizontal, o agrupamento de tuplas em fragmentos se dá pela análise de predicados, assim como na fragmentação vertical atributos são agrupados pela projeção das consultas. Observa-se que os mesmos critérios podem ser aplicados a outros modelos. Para o modelo XML, por exemplo, fragmentos verticais correspondem a sub-árvores de um esquema XML e fragmentos horizontais correspondem a conjuntos de documentos de uma coleção homogênea

de documentos [Figueiredo et al., 2010, Kling et al., 2010]. Os exemplos da Figura 3.2 ilustram as estratégias horizontal e vertical para BDs relacionais e XML.



(a) Fragmentação Horizontal (b) Fragmentação Vertical

Figura 3.2: Fragmentação de Dados Relacional e XML

Conjuntos de dados RDF são comumente armazenados por bancos de dados relacionais conhecidos como *Triple Stores*. Em um repositório *triple store*, cada conjunto é representado por uma tabela que contém 3 campos, cada qual representando os elementos de uma tripla RDF, isto é, sujeito, predicado e objeto. Um exemplo desta forma de armazenamento é apresentado pela Figura 3.1(a). A fragmentação horizontal e vertical RDF foi inicialmente proposta sobre esta composição. Fragmentos horizontais correspondem a um agrupamento baseado em valores de sujeitos ou objetos das triplas. No caso deste exemplo, uma chave de particionamento poderia ser tomada sobre os valores do sujeito identificador. Assim, uma tabela poderia ser criada para um conjunto de triplas de um mesmo valor de ID, ou até mesmo de um intervalo de valores. Fragmentos verticais foram introduzidos pelo projeto Jena [Wilkinson et al., 2003] através da criação de tabelas de propriedades que ficou conhecido como *property tables*. Assim como ilustrado pela Figura 3.1(b), propriedades acessadas frequentemente em conjunto são agrupadas em uma única tabela juntamente com seus respectivos sujeitos e predicados. Propriedades que não foram atribuídas a nenhum agrupamento são mantidas na estrutura de uma *property table* origi-

nal. Uma extensão deste modelo, chamada de *Property-class Tables* [Chong et al., 2005], corresponde a uma fragmentação híbrida como apresentado pela Figura 3.1(c). Neste caso, cada tabela corresponde a uma classe que agrupa triplas com os mesmos valores para o predicado *type*. Os campos destas tabelas de classe correspondem a propriedades utilizadas por triplas destas classes. Da mesma forma que ocorria nas *property tables*, triplas que não foram atribuídas a nenhuma das classes são mantidas na estrutura original.

(a) *Triple Store* RDF

Sujeito	Propriedade	Objeto
ID1	type	BookType
ID1	title	"XYZ"
ID1	author	"Fox,Joe"
ID1	copyright	"2001"
ID2	type	CDType
ID2	title	"ABC"
ID2	artist	"Orr,Tim"
ID2	copyright	"1985"
ID2	language	"French"
ID3	type	BookType
ID3	title	"MNO"
ID3	language	"English"
ID4	type	DVDType
ID4	title	"DEF"
ID5	type	CDType
ID5	title	"GHI"
ID5	copyright	"1995"
ID6	type	BookType
ID6	copyright	"2004"

(b) Fragmentação Vertical

Sujeito	type	title	copyright
ID1	BookType	"XYZ"	"2001"
ID2	CDType	"ABC"	"1985"
ID3	BookType	"MNP"	NULL
ID4	DVDType	"DEF"	NULL
ID5	CDType	"GHI"	"1995"
ID6	BookType	NULL	"2004"

Sujeito	Propriedade	Objeto
ID1	author	"Fox,Joe"
ID2	artist	"Orr,Tim"
ID2	language	"French"
ID3	language	"English"

(c) Fragmentação Híbrida

Classe: BookType

Sujeito	Title	Author	copyright
ID1	"XYZ"	"Fox,Joe"	"2001"
ID3	"MNP"	NULL	NULL
ID6	NULL	NULL	"2004"

Classe: CDType

Sujeito	Title	Author	copyright
ID2	"ABC"	"Orr,Tim"	"1985"
ID5	"GHI"	NULL	"1995"

Sujeito	Propriedade	Objeto
ID2	language	"French"
ID3	language	"English"
ID4	type	DVDType
ID4	title	"DEF"

Tabela 3.1: Fragmentação de Dados RDF [Abadi et al., 2009]

Em geral, para propor estratégias de fragmentação adequadas é necessária a análise da carga de trabalho prevista para um BD. Dependendo da quantidade de informações de carga disponíveis, é possível que exista um vasto número de possibilidades de fragmentação. Estas possibilidades, em geral, são avaliadas por uma função objetivo que reflete

as necessidades que a fragmentação deve atender. Além da função objetivo, as abordagens de fragmentação diferenciam-se pelo processo aplicado na busca por uma solução ótima. Estes processos podem ser classificados como *exaustivos* ou *heurísticos* [Zilio, 1998]. Um processo de busca exaustivo tem a vantagem de encontrar uma composição ótima com respeito a uma função objetivo. Porém, neste caso, o tempo consumido pelo processo não é determinado em tempo polinomial. A maioria dos métodos exaustivos utiliza uma solução conhecida como *what if?*, onde diversas possibilidades de fragmentação são simuladas pelo otimizador de um SGBD, produzindo os respectivos custos para assim determinar as recomendações mais adequadas. Dentre as abordagens que aplicam este método, destacam-se as ferramentas *AutoAdmin* [Agrawal et al., 2004] da Microsoft, o *DB2 Database Advisor* da IBM [Zilio et al., 2004] e o *AutoPart* [Papadomanolakis and Ailamaki, 2004]. Se comparados a processos exaustivos, os processos heurísticos reduzem o número de possibilidades avaliadas e podem (ou não) levar a soluções ótimas ou próximas ao ótimo.

Existem diversos trabalhos que propõem soluções de fragmentação e alocação de dados. A Tabela 3.2 sumariza um conjunto destes trabalhos tanto para o modelo de dados *Relacional* quanto para os modelos *XML* e *RDF*. As abordagens são inicialmente classificadas pelo processo de busca das soluções de fragmentação e/ou alocação de dados, que pode ser *Heurístico* ou *Exaustivo*. No contexto do processo de busca, algumas soluções definem um modelo de custo para classificar as soluções, como ocorre em [Zilio et al., 2004], [Papadomanolakis and Ailamaki, 2004], [Pavlo et al., 2012], [Nehme and Bruno, 2011] e em [Ozsu et al., 2013]. O método de fragmentação adotado pode ser *Horizontal*, *Vertical* ou *Híbrido*. Soluções classificadas como híbridas equivalem às abordagens que empregam ambas as estratégias horizontais e verticais de fragmentação. Juntamente com estratégias de fragmentação de dados, os trabalhos podem propor ainda soluções para a alocação de dados. Estes critérios de classificação são detalhados nas seções a seguir em conjunto com a descrição das abordagens.

Abordagem	Modelo Lógico	Modelo Físico	Processo	Modelo Custo	Fragmentação	Alocação
[Ceri and Pelagatti, 1984]	Relacional	Relacional	Heurístico		Horizontal	√
[Navathe and Ra, 1989]	Relacional	Relacional	Heurístico		Vertical	
[Zilio, 1998]	Relacional	Relacional	Heurístico		Horizontal	√
[Zilio et al., 2004]	Relacional	Relacional	Exaustivo	DB2	Horizontal	√
[Agrawal et al., 2004]	Relacional	Relacional	Exaustivo		Híbrido	√
[Papadomanolakis and Ailamaki, 2004]	Relacional	Relacional	Exaustivo	SQL Server	Híbrido	√
[Curino et al., 2010]	Relacional	Relacional	Heurístico		Híbrido	
[Nehme and Bruno, 2011]	Relacional	Relacional	Exaustivo	SQL Server	Horizontal	
[Pavlo et al., 2012]	Relacional	Relacional	Heurístico	Horticulture	Horizontal	
[Das et al., 2013]	Relacional	Chave-valor	Heurístico		Horizontal	
[Shute et al., 2013]	Relacional	Chave-valor	Heurístico		Horizontal	
[Ma and Schewe, 2003]	XML	XML	Heurístico		Híbrido	
[Gertz and Bremer, 2003]	XML	XML	Heurístico		Híbrido	
[Bonifati et al., 2004]	XML	Chave-valor	Heurístico		Híbrido	√
[Kanne and Moerkotte, 2006]	XML	XML	Heurístico		Híbrido	
[Abiteboul et al., 2008]	XML	Chave-valor	Heurístico		Horizontal	√
[Shnaiderman et al., 2008]	XML	XML	Heurístico		Híbrido	
[Bordawekar and Shmueli, 2008]	XML	XML	Heurístico		Híbrido	
[Shnaiderman and Shmueli, 2009]	XML	XML	Heurístico		Híbrido	
[Figueiredo et al., 2010]	XML	XML	Heurístico		Híbrido	
[Abadi et al., 2009]	RDF	Relacional	Heurístico		Vertical	
[Jiewen Huang, 2011]	RDF	Relacional	Heurístico		Híbrido	√
[Mulay and Kumar, 2012]	RDF	Relacional	Heurístico		Vertical	
[Hose and Schenkel, 2013]	RDF	Relacional	Heurístico		Híbrido	√
[Yang et al., 2013]	RDF	Relacional	Heurístico		Híbrido	√
[Yang and Wu, 2013]	RDF	Relacional	Heurístico		Horizontal	√
[Ozsu et al., 2013]	RDF	Grafos	Heurístico	Segmentação e Minimalidade	Híbrido	
[Zeng et al., 2013]	RDF	Chave-valor	Heurístico		Híbrido	√
[Galárraga et al., 2014]	RDF	Relacional	Exaustivo		Horizontal	√

Tabela 3.2: Abordagens de Fragmentação e Alocação de Dados

3.2.2 Fragmentação Horizontal

O princípio que classifica soluções de fragmentação como horizontais é o particionamento de unidades lógicas de dados em conjuntos homogêneos de dados. Por exemplo, no modelo relacional as unidades lógicas representadas pelas relações são particionadas de forma que cada fragmento contenha um conjunto homogêneo de dados, neste caso um sub-conjunto de tuplas da relação. Existem diversas estratégias para determinar o agrupamento de dados nestes conjuntos. Grande parte destas soluções definem sua estratégia baseando-se em chaves de particionamento, enquanto outras provêem soluções mais refinadas tendo como base a análise de predicados.

3.2.2.1 Abordagens Baseadas em Chaves de Particionamento

No modelo relacional, uma chave de particionamento de uma relação é um sub-conjunto de seus atributos cujos respectivos valores são usados para mapear cada tupla para uma das unidades de armazenamento chamadas de fragmentos [Zilio, 1998]. Os métodos *hash partitioning*, *range partitioning* e *round-robin* são os mais elementares 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.

O *DB2 Design Advisor* [Zilio et al., 2004] é uma ferramenta para determinar, de forma automática, grande parte dos aspectos do projeto físico de BDs, que incluem a escolha de índices, criação de visões materializadas, fragmentação e alocação de dados. A ferramenta explora o otimizador do DB2 para avaliar e recomendar soluções para uma dada carga de trabalho. A carga de trabalho é obtida através de ferramentas adicionais como o *Dynamic Statement Cache* e o *Query Patroller*. O *Dynamic Statement Cache* armazena um histórico de planos utilizados por consultas executadas junto com as respectivas frequências de execução. O *Query Patroller* permite ao usuário classificar as consultas, dando prioridade a algumas delas através de uma fila de prioridades. Sua estratégia de fragmentação é essencialmente baseada na escolha de chaves de particionamento dentre um vasto número de recomendações que são exaustivamente avaliadas por simulações sobre o otimizador do DB2.

A ferramenta *AutoAdmin* [Agrawal et al., 2004] possui objetivos e um processo similar ao *DB2 Database Advisor*. O processo é composto de três passos principais: (i) seleção de chaves de particionamento candidatas, (ii) *Merging* e (iii) Enumeração. No primeiro passo cada consulta é analisada para identificar atributos candidatos a chaves de particionamento sobre cada uma das tabelas utilizadas pela consulta. No passo de *Merging*, os

relacionamentos entre tabelas utilizadas são considerados para produzir novas recomendações a fim de determinar particionamentos horizontais, índices e visões materializadas adequadas. O objetivo do último passo (Enumeração) é produzir uma solução final das configurações candidatas levantadas pelos passos anteriores. Neste trabalho são aplicadas algumas heurísticas para podar o conjunto de recomendações candidatas e reduzir o tempo de resposta da ferramenta. Embora as técnicas de fragmentação aplicadas pelo *AutoAdmin* e pelo *DB2 Advisor* possam ser aplicadas no projeto de BDs distribuídos, é importante ressaltar que elas foram inicialmente propostas para ambientes centralizados. Este fato torna justificável a escolha por métodos exaustivos.

O *AutoPart* [Papadomanolakis and Ailamaki, 2004] utiliza, fundamentalmente, particionamento de dados no projeto automatizado para grandes bancos de dados científicos. O objetivo da ferramenta é evitar ao máximo redundâncias e a sobrecarga da utilização de índices. Segundo os autores, a utilização de uma boa estratégia de particionamento pode reduzir drasticamente a necessidade de indexação que tende a sobrecarregar SGBDs largamente distribuídos. O algoritmo utilizado efetua a fragmentação horizontal que é chamada de *particionamento categorizado* seguida pela fragmentação vertical de tabelas relacionais. No *particionamento categorizado* atributos que possuem uma pequena quantidade de valores capazes de identificar classes de objetos são considerados como chaves de particionamento para determinar fragmentos horizontais sobre as tabelas. Em seguida, as projeções das consultas são utilizadas para determinar a fragmentação vertical das relações. Este último ponto trata-se na verdade de uma fragmentação híbrida visto que o resultado da fragmentação horizontal já está sendo considerado.

O esquema de fragmentação inicial gerado pelo *AutoPart* deve conter fragmentos completamente atômicos, isto é, os fragmentos que são recuperados por determinadas consultas devem conter apenas valores que são requisitados por elas. Isto torna possível que um fragmento nesta fase possa conter apenas um único atributo. A partir deste ponto, o algoritmo tenta melhorar o esquema de fragmentação inicial efetuando a junção de fragmentos para assim evitar transações distribuídas. O custo de cada composição é avaliado

através de uma abordagem do tipo *what-if?* que simula o desempenho do *SQL Server* sobre o esquema de fragmentação em avaliação. Embora o processo seja essencialmente exaustivo, é aplicada uma heurística para reduzir o número de recomendações avaliadas. Esta heurística se baseia em resultados de iterações prévias para gerar fragmentos com um número de atributos maior a cada nova iteração. Embora um dos objetivos seja evitar a redundância, o *AutoPart* pode gerar fragmentos que não são completamente disjuntos pois utiliza replicação em alguns casos para evitar junções excessivas.

O problema de abordagens exaustivas é o tempo que as ferramentas levam para produzir as recomendações. Como cada expressão de consulta considerada na carga de trabalho deve ser compilada pelo otimizador, quanto maior for a carga de trabalho menos escalável será a ferramenta. No entanto, o método *what-if?* é ainda aplicado em trabalhos mais recentes como o Shinobi [Wu et al., 2011], pois corresponde a um método efetivo para identificar tendências na carga de trabalho. Acredita-se que soluções como estas são melhor aplicadas para SGBDs centralizados. Em razão de tratar-se de um problema NP-difícil [Kling et al., 2010], soluções heurísticas tornam-se mais atrativas para a fragmentação em bancos de dados distribuídos.

Na área de BDs largamente distribuídos, destaca-se a abordagem *Horticulture* [Pavlo et al., 2012] do projeto H-Store. Horticulture é uma ferramenta automática que particiona um banco de dados relacional *shared-nothing* paralelo e que tem por objetivo minimizar o número de transações distribuídas. Esta solução explora um conjunto de possíveis soluções, onde para cada tabela a ferramenta decide entre particioná-la horizontalmente e/ou replicá-la em todas as partições. Cada tupla é enviada a um determinado fragmento baseado nos valores da sua chave de particionamento usando particionamento por *hash* ou *range*. A replicação é utilizada para melhorar o desempenho de consultas sobre dados que são pouco atualizados. Holticulture aplica uma técnica baseada em *Large-Neighborhood Search* (LNS) que explora um espaço de soluções possíveis para encontrar uma solução de particionamento adequada. O LNS compara soluções em potencial com um modelo de custo que estima a qualidade de um particionamento para uma dada carga de trabalho.

Esta estimativa estabelece uma proporção entre o número de transações distribuídas e o grau de uniformidade da sua carga distribuída entre as partições, para determinar o desempenho de uma carga de trabalho com relação a uma solução em potencial.

MESA [Nehme and Bruno, 2011] é uma solução similar ao Horticulture que baseia-se em uma integração com o otimizador do *Microsoft SQL Server 2008 Parallel Data Warehouse*. Apesar de caracterizar-se como uma abordagem do tipo *what-if?*, a exaustão do processo é evitada pela heurística *branch-and-bound* na redução de soluções candidatas. Diferentemente do Horticulture, MESA é específica para cargas de trabalho OLAP.

Soluções recentes propostas pelos sistemas *Google F1* [Shute et al., 2013] e o *ElasTras* [Das et al., 2013] definem um esquema hierárquico de fragmentação para um BD relacional baseado em chaves estrangeiras entre tabelas. Neste caso, as tabelas são organizadas em uma hierarquia de tal forma que uma tabela filha deve conter uma chave estrangeira para a sua tabela pai como um prefixo de sua chave primária. Desta forma, a fragmentação horizontal pode envolver diversas tabelas, e evitar junções entre elas na execução de consultas. A diferença fundamental entre estes sistemas está relacionada ao sistema de armazenamento chave-valor adotado por eles. Enquanto *ElasTras* é constituído sobre o S3 da Amazon [Brantner et al., 2008], o Google F1 é definido sobre o *Spanner* [Corbett et al., 2012], um banco de dados distribuído em escala global, que permite múltiplas versões de dados, replicação e consistência em transações.

3.2.2.2 Abordagens Baseadas em Predicados

No modelo relacional, a fragmentação horizontal baseada na análise de predicados é classificada em *fragmentação primária* e *fragmentação derivada* [Ceri and Pelagatti, 1984]. Na *fragmentação primária*, uma relação é particionada usando apenas predicados definidos sobre ela. Já a *fragmentação derivada* corresponde ao particionamento de uma relação de acordo com o resultado de predicados definidos sobre outra relação. Na abordagem clássica definida por [Ceri and Pelagatti, 1984], a fragmentação primária de uma relação é definida pelo conjunto de predicados *minterm* sobre ela. Um predicado *minterm* corresponde

à conjunção de predicados simples. Assim, dada uma relação $R(a_1, ..., a_n)$, onde a_i é um atributo definido sobre o domínio D_i, um predicado simples p_j sobre R é definido por $p_j := a_i\ \theta\ valor$, $\theta \in \{=, \neq, >, \leqslant, <, \geqslant\}$ e $valor \in D_i$. Dado um conjunto $Pr_i = \{p_{i1}, ..., p_{im}\}$ de predicados simples de uma relação R_i, o conjunto de predicados *minterm* $M_i = \{m_{i1}, ..., m_{iz}\}$ é definido por $M_i = \{m_{ij} | m_{ij} = \bigwedge_{p_{ik}} p^*_{ik}\}$, $1 \leqslant k \leqslant m$, $1 \leqslant j \leqslant z$. Assim, cada fragmento primário de uma relação corresponde a um conjunto de tuplas que atendem a um dos seus predicados *minterm*.

Segundo [Ceri and Pelagatti, 1984], a fragmentação primária deve adicionalmente atender ao critério de completitude e minimalidade dos fragmentos. Um fragmento é dito *completo* se cada tupla tem a mesma probabilidade de ser acessada dentro do fragmento. Em outras palavras, a propriedade de completitude garante o acesso uniforme sobre as tuplas de um fragmento o que, intuitivamente, pode indicar que elas são acessadas em conjunto por uma ou mais transações da carga de trabalho. A minimalidade do fragmento é uma propriedade que estabelece que o conjunto de valores resultante de cada predicado simples em Pr_i deve contribuir para determinar um conjunto de fragmentos disjuntos de R. Na *fragmentação derivada* existe um relacionamento de junção de igualdade entre uma relação proprietária e uma relação membro. A fragmentação derivada aplica uma semi-junção entre as duas relações, sendo que a relação membro é então fragmentada de acordo com uma seleção aplicada sobre a relação proprietária. Os conceitos e algoritmos propostos em [Ceri and Pelagatti, 1984] são a base dos algoritmos de fragmentação horizontal que os sucederam [Noaman and Barker, 1999, Bellatreche and Boukhalfa, 2005, Wehrle et al., 2005].

Diversas técnicas de fragmentação para dados XML também foram propostas com base nas soluções clássicas para o modelo relacional [Kling et al., 2010]. No entanto, a avaliação de predicados neste contexto se dá pela análise de caminhos sobre a árvore de documentos XML. A técnica introduzida por [Ma and Schewe, 2003] define como os esquemas de fragmentação devem ser estruturados sobre um documento XML. Neste contexto, a fragmentação horizontal é definida pelo agrupamento de elementos de acordo

com um critério de seleção possibilitando a geração de fragmentos não-homogêneos. Os autores sugerem a aplicação do método proposto em [Ceri and Pelagatti, 1984] para identificar os critérios de seleção e efetuar o particionamento do documento. O conceito de fragmentação horizontal aplicado por este trabalho difere de outras soluções, que aplicam os critérios de seleção sobre uma coleção de documentos XML para gerar conjuntos de documentos homogêneos após o particionamento.

Em resumo, as abordagens propostas para XML baseiam-se em métodos de fragmentação clássicos para dividir um BD XML em uma coleção de fragmentos XML. Um fragmento XML é definido por uma expressão de caminho ([Gertz and Bremer, 2003], [Bonifati et al., 2004]), ou por um operador de álgebra XML ([Andrade et al., 2006]). Além disto, a fragmentação é executada sobre um único documento XML [Ma and Schewe, 2003], ou sobre uma coleção de documentos homogêneos [Andrade et al., 2006]. Existem outras abordagens propostas para XML que definem estratégias de fragmentação baseadas em restrições estruturais de esquemas, como profundidade e largura da árvore [Bonifati and Cuzzocrea, 2007], ou ainda restrições envolvendo o número de fragmentos gerados [Cuzzocrea et al., 2009]. Considera-se que estas abordagens são ortogonais ao problema de fragmentação de dados. Existem ainda soluções não empíricas como [Khan and Hoque, 2010], que baseiam-se na capacidade e desempenho de recursos disponíveis na rede. No entanto, em ambientes de alta volatilidade e *shared-nothing* como sistemas em *nuvem*, estas informações podem não estar disponíveis ou podem mudar a todo instante.

Para abordagens propostas para RDF, o conceito de fragmentação horizontal foi inicialmente associado ao modelo físico utilizado para seu armazenamento, como introduzido pela Tabela 3.1. Chaves de particionamento são tomadas sobre os valores do sujeito ou objeto das triplas RDF. No método proposto em [Yang et al., 2013], as consultas mais frequentes são avaliadas para identificar o percurso percorrido pelos padrões de acesso. O método identifica o padrão de tripla mais seletivo a partir de cada padrão de acesso, e define o sujeito da tripla como a chave de particionamento. Neste caso, uma tabela é criada para cada conjunto de triplas que correspondem a intervalos de valores para o sujeito.

A técnica de fragmentação horizontal baseada em predicados *minterm* foi recentemente proposta para o particionamento de dados RDF [Galárraga et al., 2014]. Neste trabalho, cada fragmento corresponde a um *minterm*, que posteriormente é alocado em conjunto com fragmentos relacionados por este predicado.

3.2.3 Fragmentação Vertical e Híbrida

O princípio da fragmentação vertical é a identificação de itens de dados acessados em conjunto pela carga de trabalho para definir como o esquema do BD deve ser fragmentado [Ozsu and Valduriez, 2011]. Assim, pode-se observar que a fragmentação vertical é aplicada sobre um esquema de BD enquanto a fragmentação horizontal é aplicada sobre todas as instâncias de itens de dados. Assim, o volume de dados a ser analisado na fragmentação vertical é menor que na fragmentação horizontal. No modelo relacional, uma relação é dividida em fragmentos que contêm um sub-conjunto de seus atributos. As abordagens exaustivas propostas para este modelo [Agrawal et al., 2004, Papadomanolakis and Ailamaki, 2004] simulam as possíveis soluções de agrupamento sobre o modelo de custo do otimizador do BD para determinar o melhor agrupamento. As soluções tradicionais que utilizam heurísticas são descritas em [Navathe et al., 1984] e em [Navathe and Ra, 1989]. Nelas, um algoritmo define como os atributos de uma relação devem ser agrupados baseado na afinidade entre eles. Na solução inicial, uma relação era fragmentada de acordo com estas afinidades pelo algoritmo *Bond Energy* [McCormick et al., 1972]. Na sequência uma solução com menor complexidade computacional foi proposta em [Navathe and Ra, 1989]. Nela, um grafo de afinidades é criado para relacionar os atributos de uma tabela, onde as arestas denotam o grau de afinidade entre pares de atributos. Esta afinidade é dada pela soma das frequências de transações que acessam um par de atributos em conjunto. Os fragmentos são gerados baseados na identificação de ciclos de afinidade neste grafo. A ideia é que afinidades altas devam ser encontradas entre atributos de um mesmo fragmento, e afinidades mais baixas entre atributos de fragmentos diferentes.

Existem algumas propostas que baseiam-se em grafos para determinar a fragmentação

de BDs relacionais. Em geral, estas propostas aplicam uma fragmentação híbrida pois, diferentemente do que foi proposto em [Navathe and Ra, 1989], não somente as relações entre itens do esquema do BD são consideradas, mas também a relação entre as instâncias de itens de dados. Um exemplo deste tipo de abordagem é a proposta da ferramenta *Schism* [Curino et al., 2010]. A principal ideia deste trabalho é representar o BD e sua carga de trabalho como um grafo, onde as tuplas são representadas por nós e transações por arestas conectando tuplas acessadas em conjunto em uma transação. Um algoritmo de particionamento de grafos é aplicado para encontrar partições balanceadas e que minimizam o peso das arestas de corte. Intuitivamente, o peso das arestas de corte está diretamente associado à quantidade de execuções de transações distribuídas.

No modelo XML, os fragmentos verticais de um documento XML correspondem a subárvores contidas nele. A grande maioria dos trabalhos neste contexto [Ma and Schewe, 2003, Gertz and Bremer, 2003, Andrade et al., 2006, Bonifati et al., 2004] recomendam soluções similares as propostas definidas em [Navathe et al., 1984] ou em [Navathe and Ra, 1989]. Um grupo de trabalhos baseia-se na representação das afinidades entre elementos de documentos XML [Bordawekar and Shmueli, 2004, Kanne and Moerkotte, 2006, Bordawekar and Shmueli, 2008, Shnaiderman et al., 2008, Shnaiderman and Shmueli, 2009]. Diferentemente do grafo de afinidades completo proposto por [Navathe and Ra, 1989], estes trabalhos consideram uma representação simplificada que incluem as afinidades sobre conexões pai-filho dos próprios documentos juntamente com as conexões entre elementos irmãos. A motivação desta estratégia é respeitar a estrutura dos documentos e reduzir a quantidade de conexões de afinidades para fins de avaliação. No entanto, algumas relações de afinidades são desconsideradas e são, consequentemente, perdidas na geração dos fragmentos. Outra desvantagem destas soluções e da ferramenta *Schism* está relacionada ao volume de dados a ser avaliado. Além de dificultar a representação, o volume de dados pode tornar o processo oneroso quando grandes BDs são considerados.

No modelo RDF, fragmentos verticais foram introduzidos pelo projeto Jena [Wilkinson et al., 2003] através da criação de tabelas de propriedades chamadas de *property*

tables, como ilustrado pela Tabela 3.1(b). Uma decomposição das *property tables* foi proposta pelo sistema *SW-Store* [Abadi et al., 2009]. Nesta abordagem, uma tabela é criada para cada propriedade com apenas dois campos que relacionam os respectivos sujeitos e objetos das triplas. Esta estratégia se mostrou adequada para o processamento de consultas analíticas baseadas em operações de agregação. Porém, o suporte a outros tipos de operações se torna uma tarefa cara em virtude da necessidade da decomposição de consultas [Ozsu et al., 2013].

Um grupo de abordagens utiliza heurísticas baseadas na composição de grafos que representam o conjunto de dados RDF ou no padrão de suas consultas. O trabalho proposto por [Yang and Wu, 2013] extrai padrões de consultas RDF e, em seguida, efetua o casamento destes padrões sobre o grafo de dados para constituir os fragmentos. No entanto, fragmentos são gerados para cada casamento de todo o conjunto de consultas. Isto implica na sobreposição de dados entre fragmentos que atendem a diferentes consultas, e a consequente replicação do conjunto de dados. Uma solução similar é proposta pelo sistema *chameleon-db* [Ozsu et al., 2013]. Embora o foco do trabalho esteja centrado no particionamento de bases RDF centralizadas, a abordagem introduz uma série de problemas associados ao casamento de grafos de consultas sobre bases particionadas. A estratégia de particionamento está baseada em um modelo de custo que relaciona os aspectos de decomposição de consultas e da minimalidade dos fragmentos. Padrões de consultas são analisados para a produção de fragmentos que possam corresponder ao máximo a estrutura dos padrões, de forma que ela não necessite ser decomposta em subconsultas, ou ainda obter dados irrelevantes de fragmentos recuperados.

As abordagens propostas por [Jiewen Huang, 2011] e pelo sistema *Trinity.RDF* da Microsoft [Zeng et al., 2013], propõem soluções de particionamento baseadas em heurísticas sobre a composição estrutural de grafos RDF. O método proposto por [Jiewen Huang, 2011], estabelece o particionamento de grafos RDF e o processamento de consultas utilizando o *framework Hadoop*. O objetivo do particionamento é facilitar que as consultas possam ser totalmente paralelizadas sem que haja necessidade de troca de dados entre

partições durante a execução. O processo de particionamento é dividido em 2 fases. Na primeira fase, os vértices do grafo RDF são particionados através do particionador de grafos METIS [Karypis and Kumar, 1998a], dada um número de partições que se deseja gerar. No segundo passo, cada tripla é alocada na partição que contém os seus vértices (sujeito e objeto). Desta forma, cada partição mantém um subgrafo do grafo original. Em conjunto com a alocação, o método replica triplas entre as partições para cobrir os cortes efetuados no grafo. A replicação ocorre de acordo com o estabelecido por uma garantia chamada de *n-hops*, que corresponde à quantidade de triplas que devem ser replicadas a partir de cada vértice que está associado a um corte. Uma extensão deste trabalho é definida em [Hose and Schenkel, 2013]. A proposta desta alternativa é replicar ainda mais para permitir que a execução de consultas seja totalmente paralelizada para os casos em que uma garantia *n-hops* não é suficiente.

O particionador METIS [Karypis and Kumar, 1998a] é aplicado pelas abordagens de [Jiewen Huang, 2011] e [Hose and Schenkel, 2013] utilizando uma otimização sobre vértices de alto-grau. Vértices de alto-grau correspondem a nós do grafo que estão conectados a muitos outros nós. Neste caso, um nó é dito de alto-grau se seu grau exceder em três unidades o desvio padrão sobre o grau médio de vértices do grafo. Segundo os autores, esta otimização visa evitar problemas no particionamento uma vez que grafos bem conectados são difíceis de particionar. Assim, vértices de alto-grau devem ser ignorados durante o particionamento no METIS e na alocação das triplas. Após o particionamento e alocação, os vértices ignorados são adicionados às partições que contêm o maior número de nós originalmente adjacentes a estes vértices. O mesmo método para identificar nós de alto-grau é utilizado em *Trinity.RDF*. Entretanto, neste caso um nó de alto-grau não é ignorado e passa a ser co-alocado com nós da sua lista de adjacência. *Trinity.RDF* armazena dados como um conjunto de nós em um repositório chave-valor. Cada par mantém um identificador de um nó RDF e um conjunto de identificadores para os nós na sua lista de adjacência.

3.2.4 Alocação de Dados

Considere um conjunto de fragmentos $F = \{f_1, ..., f_n\}$ e um sistema distribuído de servidores $S = \{s_1, ..., s_m\}$ sobre o qual um conjunto de consultas é executado. O problema da alocação de dados envolve encontrar uma distribuição ótima de F sobre S [Ozsu and Valduriez, 2011]. A qualificação de uma distribuição ótima está relacionada à minimização de custos de armazenamento e acesso dos fragmentos em uma dada distribuição, bem como à maximização do desempenho de consultas da carga de trabalho. O desempenho de consultas está diretamente associado ao relacionamento entre fragmentos e ao custo de comunicação entre eles em uma determinada distribuição. Obter soluções ótimas não é computacionalmente viável para grandes dimensões do problema que envolvem grandes quantidades de fragmentos e nós [Ceri and Pelagatti, 1982]. Assim como a fragmentação, o problema de alocação vem sendo considerado com um problema de otimização associado a soluções heurísticas.

Soluções que utilizam o particionamento baseado em chaves como [Zilio, 1998], [Zilio et al., 2004], [Agrawal et al., 2004] e [Papadomanolakis and Ailamaki, 2004], alocam seus fragmentos agrupando-os por estratégias de *hash*, *range* ou *round-robin*. Este fato mostra a forte dependência da estratégia de alocação com a utilizada para fragmentar os dados. Os autores de [Bellatreche and Benkrid, 2009] apresentam algumas abordagens que tratam a fragmentação e a alocação de forma isolada, e demonstram sua ineficiência através de uma avaliação experimental em [Bellatreche et al., 2011]. Os autores defendem que deve haver interdependência entre os processos de fragmentação e alocação, uma vez que o esquema de fragmentação é a entrada para o processo de alocação e que ambos os processos visam otimizar o mesmo conjunto de consultas. Logo, o processo de alocação deve apenas adicionar a consideração das restrições impostas pelo ambiente de armazenamento e processamento.

Embora muitas abordagens de fragmentação listadas pela Tabela 3.2 não relacionam soluções de alocação, muitas consideram um fragmento como unidade de alocação [Pavlo et al., 2012, Nehme and Bruno, 2011, Figueiredo et al., 2010, Curino et al., 2010], enquanto

outras recomendam heurísticas tradicionais de alocação [Ceri and Pelagatti, 1984, Gertz and Bremer, 2003, Ma and Schewe, 2003] baseadas essencialmente nas restrições de armazenamento dos servidores da rede. Os métodos de alocação propostos em [Abiteboul et al., 2008] e [Bonifati et al., 2004] são baseados nos métodos de distribuição de redes p2p estruturadas. Em [Bonifati et al., 2004] é escolhido um identificador único para cada fragmento para a posterior alocação dos mesmos através da aplicação de uma função de espalhamento da DHT subjacente. Similarmente, em [Abiteboul et al., 2008] palavras-chave que representam os fragmentos são utilizadas sobre uma rede p2p estruturada para endereçar o conteúdo.

Adicionalmente ao seu objetivo principal, soluções para alocação de fragmentos podem ser classificadas em redundantes ou não-redundantes, balanceadas ou não-balanceadas, e estáticas ou dinâmicas [Hauglid et al., 2010]. Diversos trabalhos propõem a replicação de alguns fragmentos durante a alocação para melhoria do desempenho de consultas e balanceamento de cargas de trabalho não-uniformes. Exemplos de abordagens que utilizam a replicação para a melhoria do desempenho de consultas são os trabalhos de [Jiewen Huang, 2011] e [Hose and Schenkel, 2013]. A replicação é utilizada para o balanceamento de cargas não-uniformes em *Horticulture* [Pavlo et al., 2012]. Além disto, as soluções podem ser estáticas ou dinâmicas com relação a alterações na carga de trabalho ou na composição do ambiente de processamento.

3.3 Sumário dos Trabalhos Relacionados

Os trabalhos listados pela Tabela 3.2 oferecem soluções para o particionamento de conjuntos de dados para os modelos relacional, XML e RDF. No entanto, observa-se que em alguns casos o modelo físico aplicado difere do modelo lógico. Para grande parte das soluções para RDF, o uso de BDs relacionais como *triple stores* é uma prática comum. Na maioria dos casos, esta forma de armazenamento leva a operações de junção custosas no processamento de consultas como demonstrado por [Zeng et al., 2013] e [Ozsu

et al., 2013]. Isto ocorre porque cada tripla é armazenada como um registro da tabela, e o relacionamento entre triplas é então recuperado através de junções. Desta forma, os casamentos para padrões de triplas são gerados separadamente e depois combinados. Este procedimento gera uma quantidade expressiva de resultados intermediários, e que posteriormente são descartados durante as junções. O estudo experimental apresentado por [Zeng et al., 2013] mostra que este problema limita a escalabilidade do processamento de consultas sobre *triple stores*, mesmo sobre uma arquitetura paralela e distribuída. O sistema *chamelon-db* aponta que é possível habilitar um processamento de consulta mais eficiente quando dados são armazenados por um modelo em grafos. Esta composição física habilita a exploração de grafos durante o processamento de consultas ao invés de utilizar junções custosas. A exploração de grafos também é aplicada por *Trinity.RDF*. Neste caso, um grafo RDF é armazenado sobre um repositório chave-valor em que cada nó e sua lista de adjacência são mantidos por um par chave-valor. Assim como em *Trinity.RDF*, a escalabilidade dos repositórios chave-valor motivou sua utilização pelos sistemas *F1* [Shute et al., 2013], *ElasTras* [Das et al., 2013] e pelos trabalhos propostos por [Bonifati et al., 2004] e [Abiteboul et al., 2008].

Como esperado, constata-se o predomínio de soluções de particionamento baseadas em heurísticas. Dentre as heurísticas aplicadas, destacam-se as baseadas na composição estrutural do banco de dados e as baseadas em informações da carga de trabalho de consultas. Em geral, o particionamento baseado na composição estrutural requer analisar todo o conjunto de dados para determinar a solução. Este é o caso das soluções baseadas no particionador de grafos METIS aplicadas em [Jiewen Huang, 2011] e [Hose and Schenkel, 2013]. No entanto, grandes volume de dados podem inviabilizar a utilização destas práticas. Abordagens baseadas na análise do padrão de consultas de uma carga de trabalho se mostram mais adequadas para melhorar o desempenho do processamento ao agrupar dados que são frequentemente acessados em conjunto, como demonstrado em [Navathe and Ra, 1989], [Bordawekar and Shmueli, 2008], *chameleon-db* [Ozsu et al., 2013] e [Yang and Wu, 2013]. No entanto, grande parte destas soluções ainda assim necessita analisar todo

o conjunto de dados para determinar a estratégia de particionamento. Como discutido anteriormente, bancos de dados em nuvem costumam ser formados progressivamente, o que pode tornar inviável a aplicação destas técnicas bem como o reparticionamento do banco a todo instante.

O capítulo seguinte apresenta uma nova solução para o particionamento de dados RDF. Além da sua evidência atual como modelo para representação de dados na *Web*, a escolha deste modelo é devida a sua flexibilidade para representar uma série de outros modelos como o relacional e o XML. Informações da carga de trabalho prevista para um BD são utilizadas como base para a definição da estratégia de particionamento. A análise da carga de trabalho é efetuada sobre uma visão sumarizada do conjunto de dados a ser particionado, o que a torna mais adequada para o contexto de repositórios em nuvem uma vez que a estratégia de particionamento é definida previamente a partir de uma previsão da carga de trabalho e pode ser re-aplicada à medida que novos dados vão sendo inseridos. Um sistema que implementa esta solução sobre um repositório chave-valor é apresentado pelo Capítulo 5 e denominado *ClusterRDF*.

CAPÍTULO 4

UMA ABORDAGEM PARA O PARTICIONAMENTO DE DADOS NA NUVEM BASEADA EM RELAÇÕES DE AFINIDADES EM GRAFOS

Este capítulo apresenta uma nova solução para o particionamento de dados baseada na análise da carga de trabalho prevista para um banco de dados em nuvem. Inicialmente, o objetivo do método de particionamento proposto é definido com base no modelo RDF e um conjunto de definições preliminares. Em seguida, o capítulo descreve como a carga de trabalho é caracterizada a partir de estruturas RDF e informações de consultas previstas. A solução de particionamento é formada pelos processos de fragmentação e alocação que aplicam heurísticas baseadas na afinidade de dados de uma carga de trabalho. Na fragmentação, unidades de armazenamento são formadas a partir da análise da carga de trabalho. Na sequência, a carga de trabalho é novamente aplicada para definir a co-alocação de fragmentos relacionados. Algumas considerações sobre a aplicação do método proposto para o particionamento de dados XML são adicionalmente apresentadas. No final deste capítulo o objetivo do particionamento é relacionado a esquemas de particionamento produzidos pela solução.

4.1 Definições Preliminares

Um conjunto de dados a ser particionado é representado através de um grafo capaz de descrever as associações entre dados. Assume-se que um esquema possa ser extraído a partir do grafo de dados de forma a habilitar a representação de sua estrutura na forma de associações entre itens de dados, sendo que cada item representa um sub-conjunto de dados com estruturas similares. A caracterização da carga de trabalho se dá sobre

esta estrutura, e corresponde à base para as heurísticas aplicadas no particionamento. Esta sumarização do conjunto de dados através de estruturas ou esquemas impede a exaustão do processo de particionamento em bases de grande volume por não necessitar avaliar todo o grafo de dados para determinar a solução de particionamento. Quando estas bases são dispostas sobre nuvens computacionais, seu crescimento se dá de forma progressiva à medida que novos dados vão sendo gerados ou compartilhados. Para tanto, é importante que a estratégia de particionamento possa ser determinada previamente de forma que novos dados possam ser adequadamente particionados à medida que vão sendo submetidos ao repositório.

A definição desta solução está baseada no modelo RDF devido a sua utilização expressiva como modelo de dados para repositórios de grande porte. Além disto, RDF atua como um modelo de grafo genérico capaz de expressar outros modelos de dados como o XML. Um grafo RDF é definido como um conjunto de triplas compostas de um sujeito, propriedade e objeto (s,p,o). Com a finalidade de definir o domínio dos componentes das triplas, assume-se a existência dos conjuntos $\mathcal{U}$ e $\mathcal{L}$, onde $\mathcal{U}$ corresponde a URIs (*Uniform Resource Identifiers*) e $\mathcal{L}$ corresponde a valores literais. Assim, uma tripla (s, p, o) $\in (\mathcal{U} \times \mathcal{U} \times \{\mathcal{U} \cup \mathcal{L}\})$. Logo, um grafo RDF é definido por D como um conjunto de triplas. RDF aplica um modelo de dados no qual triplas são relacionadas na forma de um grafo direcionado. O sujeito e o objeto de uma tripla correspondem, respectivamente, à origem e ao destino de uma aresta do grafo. A Figura 4.1 apresenta um exemplo de um grafo RDF que relaciona dados de produtos as suas respectivas ofertas e vendedores. Neste exemplo, o nó *product1* atua tanto como sujeito na tripla *(product1, feature, prFeature1)*, quanto como objeto na tripla *(offer1, offer, product1)*.

A caracterização da carga de trabalho tem como entrada um conjunto de consultas sobre um grafo RDF definidas através da linguagem SPARQL. SPARQL é a linguagem recomendada pela W3C para expressar consultas sobre repositórios RDF. O núcleo da sintaxe SPARQL é baseado em um conjunto de *padrões de triplas* semelhante às triplas RDF, exceto pelo fato que sujeitos e objetos podem ser representados por variáveis. SPARQL

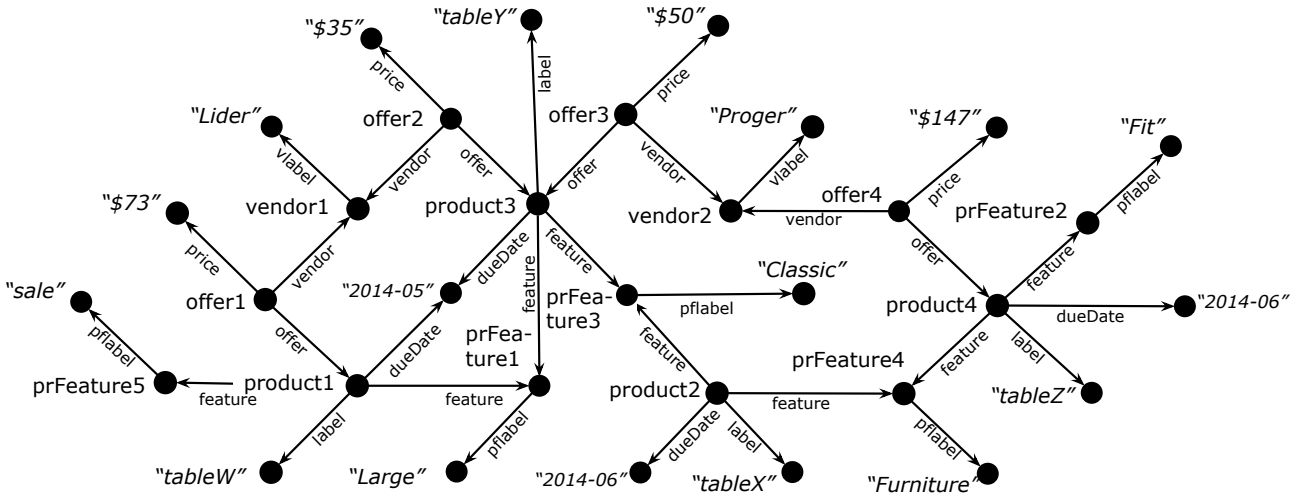


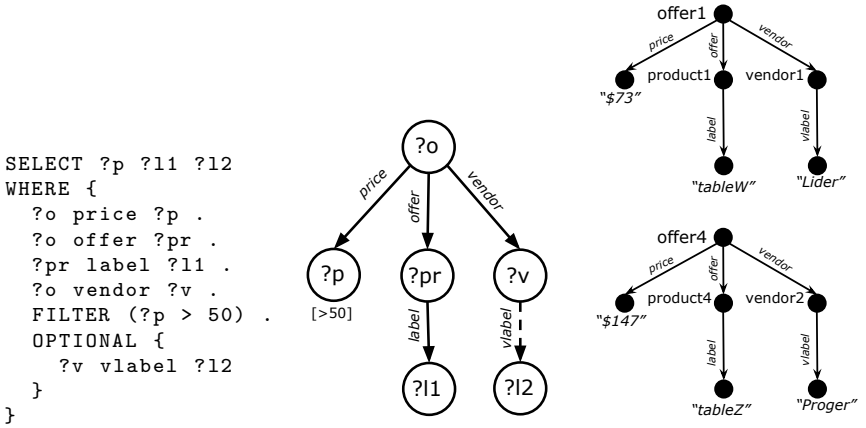
Figura 4.1: Grafo RDF [Bizer and Schultz, 2009]

é uma linguagem de consulta cujo processamento é definido como um problema de casamento de subgrafos. Neste trabalho, *padrões de grafos* são definidos para representar o núcleo da linguagem que corresponde à conjunção de triplas estabelecida pelo operador AND. Antes de introduzir a definição de *padrão de grafo*, assume-se a existência de um conjunto $\mathcal{V}$ de variáveis, que é disjunto dos conjuntos $\mathcal{U}$ e $\mathcal{L}$ anteriormente apresentados. Assim como definido pela linguagem SPARQL, variáveis em $\mathcal{V}$ são representadas pelo uso de pontos de interrogação (?).

Definição 4.1.1 *(Padrão de Grafo): Um padrão de grafo é definido por* $G = (V, E, r)$*, onde: (1)* $V \subseteq \{\mathcal{V} \cup \mathcal{U} \cup \mathcal{L}\}$*; (2)* $E \subseteq (\{\mathcal{V} \cup \mathcal{L}\} \times \mathcal{U} \times V)$*, sendo que para cada aresta* $(\hat{s},\hat{p},\hat{o}) \in E$*,* $\hat{s}$ *é a origem da aresta,* $\hat{p}$ *é o rótulo da propriedade,* $\hat{o}$ *é o alvo da aresta; e (3)* r *é uma função parcial que atribui expressões de filtro para nós variáveis em* G*. Uma expressão de filtro é expressa pela forma* $?x\ \theta\ c$*, onde* $?x \in \mathcal{V}$*,* $c \in \{\mathcal{U} \cup \mathcal{L}\}$ *e* $\theta \in \{=, >, \leqslant, <, \geqslant\}$*.*

A partir deste ponto, usa-se $V(G)$ para denotar o conjunto de nós de um padrão de grafo e $E(G)$ para o conjunto arestas. Além do fragmento conjuntivo da linguagem SPARQL, a definição de padrão de grafo incorpora o operador FILTER através da função parcial r. Adicionalmente, consultas SPARQL podem ser construídas a partir de padrões

de grafos em conjunto com os operadores OPTIONAL e UNION. Para tanto, qualquer padrão de grafo definido como G é uma consulta SPARQL e pode ser recursivamente definida da seguinte forma: se g_1 e g_2 são consultas SPARQL, então expressões (g_1 AND g_2), (g_1 UNION g_2) e (g_1 OPTIONAL g_2) são consultas SPARQL.



(a) Consulta SPARQL (b) Padrão de Grafo (c) Resultado

Figura 4.2: Exemplo de Consulta SPARQL [Bizer and Schultz, 2009]

O exemplo da Figura 4.2 apresenta uma consulta SPARQL que recupera produtos e vendedores de ofertas com preço superior a 50. A aplicação desta consulta sobre o grafo RDF da Figura 4.1 produz os sub-grafos da Figura 4.2(c). Uma representação para o padrão de grafo equivalente a esta consulta é apresentado pela Figura 4.2(b). Triplas e suas conjunções são representadas através de nós e arestas no padrão de grafo. Arestas pontilhadas denotam o relacionamento estabelecido pelo operador OPTIONAL, assim como ocorre para o padrão *(?v vlabel ?l2)*. Rótulos de nós dispostos entre colchetes remetem a filtros aplicados sobre seus valores, assim como o rótulo *[>50]* corresponde ao filtro aplicado sobre os valores de *?p*. Por razões de simplificação da notação, optou-se por representar padrões interligados pelo operador UNION através de grafos independentes. Embora o exemplo corresponda a uma formação em árvore, padrões de grafo conceitualmente assumem ciclos.

Uma carga de trabalho é dada por um conjunto de consultas SPARQL denominado Q, tal que uma consulta $q \in Q$ é definida por um padrão de grafo G. Assume-se que

o casamento do padrão q sobre D é definido em termos do método de isomorfismo de subgrafos introduzido por [Ullmann, 1976]. Os subgrafos apresentados pela Figura 4.2(c) correspondem aos casamentos do padrão de grafo da Figura 4.2(b) sobre o grafo RDF da Figura 4.1. Observe que cada subgrafo que compõe o resultado é isomórfico ao padrão de grafo da consulta. O resultado de q que representa os casamentos de seu padrão de grafo é definido por $B(q) = \{b_1, ..., b_n\}$, tal que b_i é um subgrafo do grafo RDF, isto é, $b_i \subseteq D$.

Considera-se que consultas SPARQL são processadas sobre repositórios de dados particionados, assim como introduzido pela Seção 2.2. O particionamento de um repositório é nomeado por $\mathcal{P}$ e definido por um conjunto de partições, cada qual composta por um subconjunto de triplas de um grafo RDF D a ser alocado em um dos m servidores do sistema distribuído. Formalmente, $\mathcal{P} = \{P_1, ..., P_m\}$, tal que $\bigcup_{i=1}^{m}(P_i) = D$ e $P_i \cap P_j = \{\}$ para qualquer $i \neq j$. Seja q uma consulta e $B(q)$ sua resposta, pretende-se obter $\mathcal{P}$ de forma que a quantidade de partições necessárias para recuperar dados de cada subgrafo $b \in B(q)$ seja minimizada. Assim, idealmente cada subgrafo $b \in B(q)$ deve estar completamente alocado em uma partição. Quando o número de partições exceder a 1, diz-se que b está segmentado em $1 + s$ partições. O conceito de segmentação é estendido para o conjunto $B(q)$ para determinar a quantidade total de partições a mais que são necessárias para recuperar os subgrafos, isto é, o somatório de s de cada $b \in B(q)$. A medida $\hat{P}$ define a segmentação de $B(q)$ sobre um particionamento $\mathcal{P}$ como segue:

Definição 4.1.2 *Dado um particionamento $\mathcal{P}$ de um grafo RDF D, a medida $\hat{P}$ de $\mathcal{P}$ com relação a q é definida por:*

$$\hat{P}(q, \mathcal{P}) = \Big|\{(b, P) \in (B(q) \times \mathcal{P}) | b \cap P \neq \oslash\}\Big| - \Big|B(q)\Big| \tag{4.1}$$

A relação (b, P) representa cada partição que contém triplas de um subgrafo $b \in B(q)$. Em um particionamento ideal, a quantidade total de elementos desta relação equivale

à quantidade de subgrafos de $B(q)$ de forma que será necessário acessar apenas uma partição para responder cada $b \in B(q)$. Assim, valores de $\hat{P}$ superiores a 0 indicam que há segmentação de pelo menos um dos subgrafos de $B(q)$.

Uma extensão da medida $\hat{P}$ é considerada para definir o objetivo do particionamento da presente abordagem. Antes de apresentar propriamente o objetivo, é importante destacar que a solução proposta favorece as consultas mais frequentes de uma carga de trabalho Q. Para tanto, uma função f define a frequência esperada de uma consulta $q \in Q$ em um período de tempo. O problema do particionamento é formalmente definido através do objetivo de encontrar um particionamento $\mathcal{P}$ que minimiza a seguinte equação:

$$min \sum_{q \in Q} f(q).\hat{P}(q, \mathcal{P}) \tag{4.2}$$

O problema de computar um particionamento adequado para um repositório RDF é tratado através de uma solução baseada em informações da carga de trabalho. Para se adequar à Equação 4.2, a solução dá preferência para que consultas com frequências mais expressivas encontrem seus respectivos casamentos em partições únicas. A carga de trabalho é caracterizada para identificar itens de dados acessados em conjunto por consultas em Q. O caminho percorrido por consultas e suas frequências respectivas são considerados para quantificar a afinidade entre itens de dados de uma estrutura RDF. Esta medida de afinidade é a base das heurísticas aplicadas na solução de particionamento a ser proposta. O problema de particionamento é decomposto em dois sub-problemas denominados *fragmentação* e *alocação*.

A *fragmentação* tem por objetivo gerar fragmentos cuja estrutura corresponda à estrutura das consultas mais frequentes da carga de trabalho. Neste nível são produzidos fragmentos de acordo com a afinidade entre itens de dados. Um fragmento é uma unidade de armazenamento a ser alocado pelo repositório distribuído. Na *alocação*, a medida de afinidade é estendida aos fragmentos gerados pela fase anterior para co-alocar fragmentos relacionados nos servidores do repositório. Em resumo, a *fragmentação* cria unidades de

armazenamento e a *alocação* procura alocar fragmentos relacionados em um mesmo servidor. As seções seguintes apresentam detalhes sobre a caracterização da carga de trabalho e a solução proposta para a fragmentação e alocação de dados.

4.2 Caracterização da Carga de Trabalho

Nesta seção é apresentado um método para representar informações da carga de trabalho. O fundamento deste método é identificar e medir relações de afinidade entre nós do grafo RDF. Valores de afinidade são considerados pela abordagem de particionamento como uma heurística para dar suporte à função objetivo definida em 4.2.

Denomina-se uma *Estrutura RDF* uma composição da estrutura de um grafo RDF e o tamanho esperado para suas instâncias. Embora RDF possa definir um modelo livre de esquema, em geral um grafo RDF representa ambos esquema e instância. Usualmente, repositórios RDF definem a propriedade `type` para conectar entidades a suas respectivas classes. Na Figura 4.3 uma *Estrutura RDF* é representada pela forma tracejada e denota os componentes das classes e os relacionamentos entre elas. Uma *Estrutura RDF* é um grafo cíclico não direcionado e definido como uma 6-tupla $S = (C, L, l, A, s, o)$, onde (1) C é um conjunto de nós rotulados que representam classes RDF; (2) L é um conjunto de nós rotulados que se referem a propriedades entre classes e seus respectivos valores literais no grafo RDF; (3) l atribui um tipo de dado para cada nó em L; (4) A é um conjunto de arestas não-direcionadas $(n_1,\ p,\ n_2) \in (C \times \mathcal{U} \times \{C \cup L\})$ representando associações entre nós através de uma propriedade p; (5) s é uma função que atribui o tamanho esperado para as instâncias de nós em $\{C \cup L\}$; e (6) o fornece a cardinalidade esperada das associações $(n_1,\ p,\ n_2)$; isto é, uma função que mapeia $(C \times \mathcal{U} \times \{C \cup L\})$ para um inteiro que define para cada nó $n_1 \in C$ o número esperado de ocorrências de um nó $n_2 \in (C \cup L)$ associado a ele através de uma propriedade p.

A Figura 4.3 mostra uma *Estrutura RDF* extraída de um grafo RDF. A notação utilizada representa classes RDF através de formas retangulares, literais através de eclipses

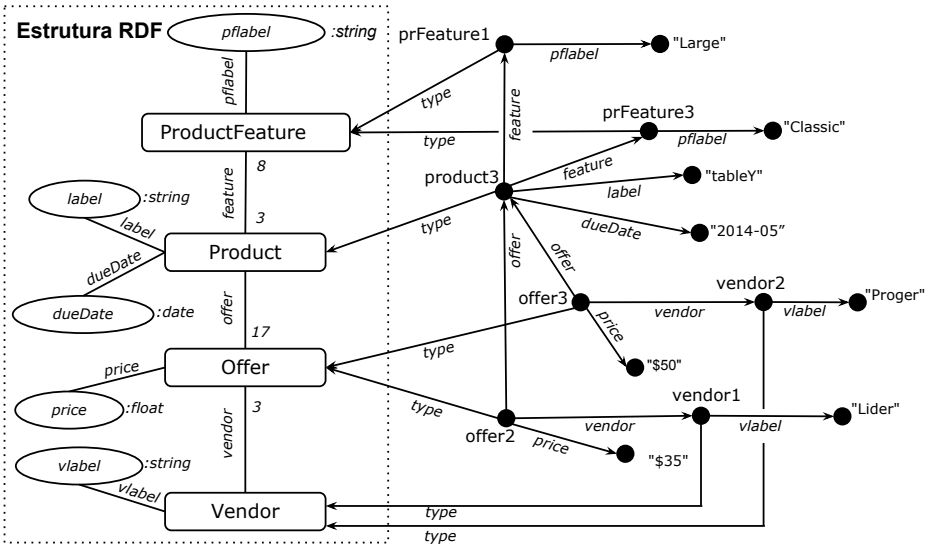


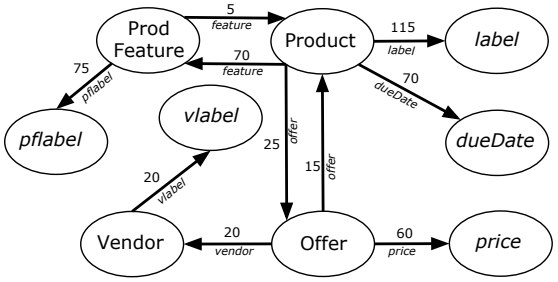
Figura 4.3: Estrutura RDF

e as propriedades como rótulos entre nós. O tamanho dos nós que assumem valores literais consiste do tamanho esperado para estes valores, enquanto para um nó do tipo classe, o tamanho é dado pela estrutura necessária para manter apontadores (URIs) para os nós imediatamente adjacentes a ele. Para fins de simplificação, omite-se o valor do tamanho dos nós e considera-se que para qualquer nó n, $s(n) = 1$. No exemplo, $o(Product, feature, ProdFeature) = 8$ porque o número médio de ocorrências de *ProdFeature* associado a uma instância de *Product* através da propriedade *feature* é 8. De forma análoga, uma instância de *ProductFeature* é relacionada a três instâncias de *Product* em média. Isto é, $o(ProdFeature, feature, Product) = 3$. Além disto, existem relacionamentos multi-valorados entre $(Product, offer, Offer)$ e $(Vendor, vendor, Offer)$. Assume-se que para as associações restantes entre quaisquer nós n_1 e n_2, $o(n_1, p, n_2) = 1$.

Dada uma representação de uma *Estrutura RDF*, a carga de trabalho é caracterizada com base no conjunto de caminhos percorridos em cada uma das consultas sobre esta estrutura. Formalmente, Q é um conjunto de consultas SPARQL representadas por padrões de grafos definidos por G, onde uma função f define a frequência esperada de consultas em Q. A Tabela 4.4(a) apresenta um conjunto de 4 consultas, suas respectivas frequências, e o conjunto de arestas do padrão de grafo dado por um conjunto de suas triplas.

Q	f	E
q_1	70	$\{(Product, label, label),$ $(Product, dueDate, dueDate),$ $(Product, feature, ProdFeature),$ $(ProdFeature, pflabel, pflabel)\}$
q_2	15	$\{(Offer, price, price),$ $(Offer, offer, Product),$ $(Product, label, label)\}$
q_3	20	$\{(Offer, price, price),$ $(Offer, vendor, Vendor),$ $(Vendor, vlabel, vlabel)\}$
q_4	25	$\{(Product, label, label),$ $(Product, offer, Offer),$ $(Offer, price, price)\}$
q_5	5	$\{(ProdFeature, feature, Product),$ $(ProdFeature, pflabel, pflabel),$ $(Product, label, label),$

(a) Matriz de Uso



(b) Grafo de Afinidade

Figura 4.4: Carga de Trabalho e Grafo de Afinidade

A afinidade de dois nós a e b relacionados por uma propriedade p é dada pela frequência na qual uma aresta (a, p, b) é acessada pelos padrões de grafo de consultas. Deste modo, uma função de afinidade $aff(a, p, b)$ toma um conjunto de consultas Q e computa a soma das frequências das consultas que acessam (a, p, b). Dado que o conjunto de arestas de uma consulta q é representado por $E(q)$, o conjunto de consultas que acessam (a, p, b) é dado por $Q_{(a,p,b)} = \{q \in Q \mid (a, p, b) \in E(q)\}$, e $aff(a, p, b) = \sum f(q),\ q \in Q_{(a,p,b)}$. Como exemplo, considere a carga de trabalho dada pela Tabela 4.4(a). A afinidade entre *Product* e *label* pela propriedade *label* consiste na soma das frequências das consultas q_1, q_2, q_4 e q_5. Assim, $aff(Product, label, label) = f(q_1) + f(q_2) + f(q_4) + f(q_5) = 115$. A função de afinidade pode ser usada para rotular suas arestas em um grafo direcionado, conforme mostra a Figura 4.4(b). A este grafo é dado o nome de *Grafo de Afinidades*, o qual é definido como uma tupla $\mathcal{A} = (N, \hat{E}, aff)$, onde N é um conjunto de nós em uma *Estrutura RDF*, tal que $N = \{C \cup L\}$; e $\hat{E}$ é um conjunto de *arestas de afinidade* que relacionam dois nós a e b por um valor de afinidade através de uma propriedade p $(aff(a, p, b))$. O conjunto de arestas de um grafo de afinidades é dado por arestas em padrões de grafo de consultas, tal que $\hat{E} = \bigcup_{q \in Q} E(q)$.

Observe que o sentido das arestas no *Grafo de Afinidades* não necessariamente corresponde ao sentido das arestas do grafo RDF. Isto ocorre porque o *Grafo de Afinidades*

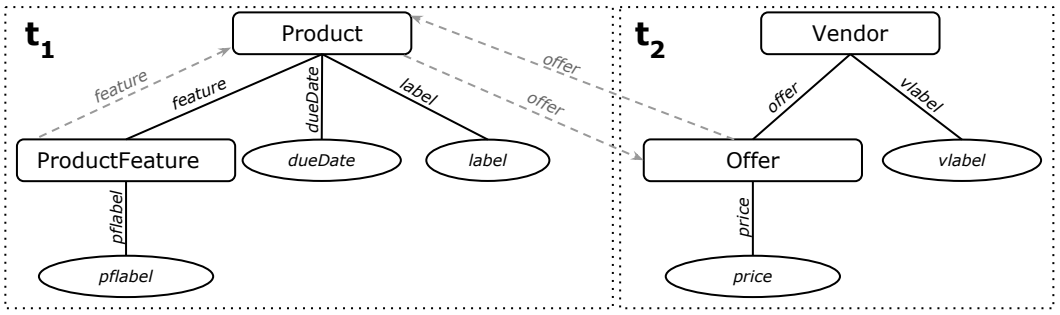


Figura 4.5: *Templates* de Fragmentação

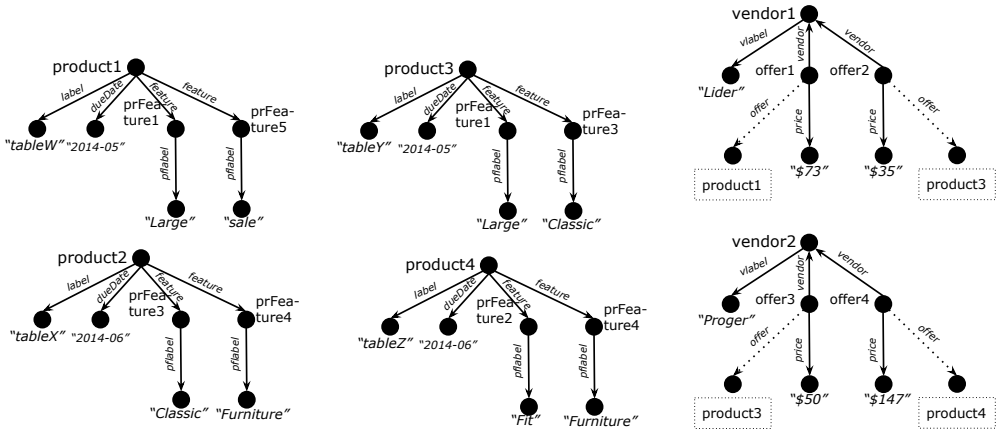
preza por representar o sentido dos casamentos efetuados pelas consultas através de um *padrão de grafo*. No exemplo, a aresta $(ProdFeature, feature, Product)$ corresponde ao sentido de navegação da consulta q_5, porém não corresponde ao sentido das arestas envolvendo nós das classes *Product* e *ProdFeature* no grafo RDF.

A seguir, arestas de afinidade são utilizadas na fragmentação de dados RDF para unir itens de dados fortemente relacionados em uma mesma unidade de armazenamento. Em seguida, esta medida de afinidade é estendida aos fragmentos para co-alocar fragmentos relacionados nos servidores de um repositório de dados distribuído.

4.3 Fragmentação RDF

Dada uma carga de trabalho prevista sobre uma *Estrutura RDF*, uma estratégia de fragmentação é computada de forma que dados RDF relacionados possam ser acomodados em unidades de armazenamento. O método de fragmentação proposto por esta seção produz *templates* de fragmentação que determinam a formação de unidades de armazenamento a partir de uma *Estrutura RDF*. Além de habilitar a fragmentação de dados já inseridos no repositório, estes *templates* permitem fragmentar dados que serão submetidos ao repositório no futuro.

Templates de fragmentação correspondem a subgrafos da *Estrutura RDF* para determinar a formação de unidades de armazenamento baseadas nas instâncias das classes desta estrutura e suas associações. A Figura 4.5 apresenta um exemplo com *templates* de fragmentação para a *Estrutura RDF* da Figura 4.3. Um *template* corresponde a uma



(a) Fragmentos de t_1 (b) Fragmentos t_2

Figura 4.6: Fragmentos de Acordo com *Templates* de Fragmentação

composição em árvore, sendo que o nó raiz de um *template* determina como se dará o aninhamento dos dados a partir de uma instância do item de dado da classe raiz. A Figura 4.6 apresenta um exemplo da produção de fragmentos a partir dos *templates* com relação ao grafo RDF da Figura 4.1. De acordo com o *template* t_1, fragmentos são gerados a partir de cada instância da classe *Product*. O mesmo ocorre para as instâncias de *Vendor* através do *template* t_2. As arestas de um grafo RDF que não foram incluídas por quaisquer dos *templates* deverão ser incluídas nos fragmentos que contêm seu nó origem juntamente com a inclusão do nó alvo para que seja mantida a navegabilidade entre os fragmentos. No exemplo, a aresta *offer* não foi incluída pois representa o corte efetuado na *Estrutura RDF*. Neste caso, os fragmentos de t_2 que contêm instâncias de nós da classe *Offer* devem incorporar a aresta faltante que tem como alvo instâncias da classe *Product*. As arestas representadas como setas pontilhadas na Figura 4.5 correspondem a arestas do grafo de afinidades que não foram utilizadas para compor estes aninhamentos, mas que serão futuramente consideradas para determinar a alocação de fragmentos.

O objetivo da fragmentação é reunir dados relacionados por altas afinidades em um mesmo fragmento. Porém, existem duas restrições que são consideradas neste processo. A primeira restrição diz respeito a redundância de dados que eventualmente poderá ser gerada na produção de fragmentos. No exemplo é possível identificar que *prFeature1*, *prFea-*

ture3 e *prFeature4* estão replicados para as diferentes instâncias de *Product*. Isto se deve ao relacionamento multi-valorado envolvendo *Product* e *ProductFeature* como definido pela *Estrutura RDF* da Figura 4.3. Assim como para qualquer outro relacionamento N:M, a redundância de dados se torna inevitável quando efetuado o aninhamento entre as instâncias das classes envolvidas. No entanto, deseja-se fornecer um controle quanto à quantidade de dados replicados gerados. Este controle é dado por uma função u que atribui um limiar assumido para cada item de dado de uma *Estrutura RDF* e que determina a quantidade máxima de dados replicados permitida para cada instância deste item. Assume-se que estes limiares são fornecidos como entrada ao processo. A atribuição adequada destes limiares pode impedir, por exemplo, anomalias de atualização sobre itens de dados que frequentemente sofrem atualizações ao atribuir valores baixos a estes limiares.

A segunda restrição considerada no processo de fragmentação é relacionada ao tamanho máximo permitido para fragmentos. Como visto na Seção 2.2, o desempenho de consultas distribuídas não é apenas afetado pela quantidade de mensagens trocadas entre servidores, mas também pelo tamanho de suas mensagens. Um tamanho adequado para estas mensagens motivou a adoção de um limiar de armazenamento como a base do método de fragmentação a ser proposto. Além deste motivo, existem outras vantagens associadas à adoção de um delimitador para o tamanho de fragmentos. O controle deste tamanho está diretamente associado a estratégias que tratam do particionamento dinâmico e da replicação de dados. Por exemplo, ao assumir um nível de granularidade fino para o tamanho de fragmentos, em muitos casos é possível tratar variações na carga de trabalho através da migração de fragmentos de um servidor para outro ao invés de assumir um procedimento custoso de re-fragmentação. Denota-se por Γ o limiar de armazenamento assumido para uma dada carga de trabalho. O objetivo é formar fragmentos que contenham a maior quantidade de nós relacionados possível sem extrapolar o limiar de armazenamento assumido. O problema da fragmentação de dados é descrito a seguir, juntamente com um método proposto para resolvê-lo.

4.3.1 O Problema da Fragmentação RDF

O problema da fragmentação RDF define a formação de *templates* de fragmentação a partir de arestas de afinidades e restrições impostas pela sua estrutura. Dada uma *Estrutura RDF* $S = (C, L, l, A, s, o)$ e um grafo de afinidade $\mathcal{A} = (N, \hat{E}, aff)$, pretende-se obter *templates* de fragmentação $T = \{t_1, ..., t_m\}$, $m \geqslant 1$ e $t_i = (N_{t_i}, E_{t_i})$ tal que $N_{t_i} \cap N_{t_j} = \{\}$ para todo $i \neq j$ e $\bigcup_{i=1}^{m}(t_i) = (N, E')$, onde $E' \subseteq \hat{E}$.

Dado que a fragmentação é baseada em um limiar de armazenamento, é necessário determinar o tamanho esperado para um *template* de fragmento $t_i \in T$. O tamanho de t_i é dado pela soma do número esperado de ocorrências dos nós multiplicados pelos seus respectivos tamanhos. A composição em árvore para *templates* de fragmento requer medir a ocorrência do nó na sua estrutura de aninhamento. A função $Occ \downarrow (n)$ mapeia cada nó em um *template* t_1 para seu número esperado de ocorrências em uma instância de t_i. A função é recursivamente definida como segue: $Occ \downarrow(n) = 1$ se n é o nó raiz de t_i, e $Occ \downarrow(n) = Occ \downarrow(n_0) \times o(n_0, p, n)$ onde n_0 é um nó pai de n pela propriedade p em t_i. Como exemplo, considere t_1 na Figura 4.5. O número de ocorrências para *ProdFeature* é definido por $Occ \downarrow(ProdFeature) = Occ \downarrow(Product) \times o(Product, ProdFeature)$. Segundo a *Estrutura RDF* da Figura 4.3, $o(Product, ProdFeature) = 8$, e $Occ \downarrow(Product) = 1$ pois *Product* é o nó raiz do t_1. Assim, $Occ \downarrow(ProdFeature) = 1 \times 8 = 8$. A mesma quantidade de ocorrências é obtida para *pflabel*, pois $Occ \downarrow(pflabel) = Occ \downarrow(ProdFeature) \times o(ProdFeature, pflable) = 8 \times 1 = 8$. Para os demais nós, o número de ocorrências obtido é 1. O tamanho de t_i é finalmente definido por $size(t_i) = \sum_{n \in t_i}(Occ \downarrow (n) \times s(n))$. Considerando que o tamanho de cada nó é definido como 1 ($s(n) = 1$), o tamanho de t_1 é dado pela soma das ocorrências dos nós, isto é, $size(t_1) = 1 + 8 + 8 + 1 + 1 = 19$.

A quantidade de dados replicados para um nó n pode ser calculada de maneira análoga ao cálculo de ocorrências de um nó na sua estrutura de aninhamento. Para tanto, assume-se uma nova função denominada $Occ \uparrow (n)$ que mapeia cada nó em um *template* t_i para seu número esperado de ocorrências replicadas em uma instância de t_i. Ao contrário da função $Occ \downarrow (n)$, a função $Occ \uparrow (n)$ considera a ocorrência de n com relação a sua ascendência,

isto é, do nó n para seu nó pai n_0. A função $Occ \uparrow (n)$ é recursivamente definida como segue: $Occ \uparrow(n) = 1$ se n é o nó raiz de t_i, caso contrário $Occ \uparrow(n) = Occ \uparrow(n_0) \times o(n, p, n_0)$. Como exemplo, considere novamente t_1 na Figura 4.5 para calcular a quantidade de dados replicados para *ProdFeature*. Segundo a *Estrutura RDF*, *ProdFeature* está associada a 3 instâncias de *Product*, isto é, $o(ProdFeature, Product) = 3$. Assim, $Occ \uparrow(ProdFeature) = Occ \uparrow(Product) \times o(ProdFeature, Product) = 1 \times 3 = 3$.

Com o objetivo de formalmente definir o problema, é introduzida a noção de um conjunto de arestas fortemente correlacionadas *sce (strongly correlated edges)* a partir de um nó específico de um grafo de afinidade, definido como segue: $sce(n_0) = \{(n_0, p, n_1) | aff(n_0, p, n_1) \geq aff(n_2, p', n_1)$ para qualquer aresta (n_2, p', n_1) que tem n_1 como alvo$\}$. Intuitivamente, *sce* determina quais arestas que partem de n_0 e que tem como alvo nós que são mais fortemente conectados a partir de n_0 do que a partir de qualquer outro nó no grafo de afinidades. Denota-se por sce^+ o fechamento transitivo da relação *sce*. De acordo com o grafo de afinidades da Figura 4.4(b), o conjunto de arestas fortemente correlacionadas a partir de *ProdFeature* é dado por $sce(ProdFeature) = \{(ProdFeature, pflabel, pflabel)\}$. Neste caso, a aresta $(ProdFeature, feature, Product)$ não faz parte do conjunto pois a aresta $(Offer, offer, Product)$ apresenta um valor de afinidade superior.

O problema de fragmentação consiste em encontrar T de forma que as seguintes condições sejam satisfeitas: (1) $size(t_i) \leq \Gamma$ para cada $t_i \in T$; (2) $Occ \uparrow (n) \leq u(n)$ para cada nó n em um *template* $t \in T$; e (3) se (s_1, p_1, o_1) e (s_2, p_2, o_2) são arestas no mesmo fragmento, então $(s_2, p_2, o_2) \in \{sce^+(s_1) \cup sce^+(o_1)\}$. A primeira condição define que todos os *templates* em T devem se ajustar a Γ e a segunda condição define que todo nó contido em um *template* deve se ajustar ao seu respectivo limiar atribuído pela função u. Por fim, a última condição estabelece a geração de fragmentos pelo agrupamento de arestas fortemente correlacionadas.

Como exemplo, considere $\Gamma = 20$, o grafo de afinidade da Figura 4.4(b), e para qualquer nó n, $u(n) = 100$. Os *templates* 4.5 satisfazem as condições porque (1) o tamanho dos *templates* não extrapola o limiar de armazenamento, isto é $size(t_1) = 19$ e $size(t_2) = 4$; (2)

a quantidade de dados replicados para *ProductFeature* e sua descendência não extrapola a 100 unidades, isto é, $Occ \uparrow (ProductFeature) = 3$ e $Occ \uparrow (pflabel) = 3$; e (3) as arestas nos *templates* apresentam valores de afinidade superiores a arestas não incluídas no *template* e que incidem em quaisquer de seus nós.

4.3.2 O Algoritmo *affFrag*

Um algoritmo de fragmentação é proposto com base em *Estruturas RDF* e cargas de trabalho. O Algoritmo *affFrag* recebe como entrada uma *Estrutura RDF* S com informações do tamanho dos nós e o número de suas ocorrências, um grafo de afinidades $\mathcal{A}$, um limiar de armazenamento Γ e um limiar de replicação u. O algoritmo produz *templates* de fragmentos baseado na análise de conjuntos fortemente relacionados se os respectivos tamanhos de seus nós estão de acordo com Γ e se o número de ocorrências replicadas não excede ao determinado pela função u.

Variáveis auxiliares são utilizadas para computar os *sce*'s: *allNodes* para manter os nós em $\mathcal{A}$ que não foram atribuídos a um fragmento até o momento; e *allEdges* para manter as arestas em $\mathcal{A}$ que até o momento não foram percorridas. Nós e arestas no fragmento que está sendo computado são inseridos nas variáveis *tNodes* e *tEdges*, respectivamente, enquanto seu tamanho é mantido em *tSize*. A variável *border* consiste de arestas que correspondem ao fechamento transitivo de *sce*.

O algoritmo processa as arestas em $\mathcal{A}$ em ordem descendente de afinidade. Dada uma aresta (n_a, p, n_b), um nó n_1 é escolhido entre n_a e n_b como a raiz do *template* de fragmento que está sendo formado. Em geral, n_a é definido como raiz por constituir o nó origem da aresta com a mais alta afinidade (Linha 6). No entanto, quando tratar-se de um relacionamento multi-valorado no sentido n_b para n_a, opta-se por tornar o n_b como raiz da árvore caso a multiplicidade do sentido oposto for mono-valorada (Linhas 7-9). Este procedimento impede que sejam geradas redundâncias desnecessárias. Considere como exemplo o *template* t_2 do exemplo da Figura 4.5. O relacionamento envolvendo as classes *Vendor* e *Offer* corresponde a um relacionamento 1:N, isto é, uma instância de *Vendor*

Algoritmo affFrag

```
   Entrada: Estrutura RDF S = (C, L, l, A, s, o), Grafo de Afinidades A = (N, Ê, aff), Γ e u
   Saída: T template de fragmentação
1  T ← {};
2  allNodes ← N;
3  allEdges ← Ê;
4  Faça
5  |  (n_a, p, n_b) ←aresta em allEdges com a mais alta afinidade;
6  |  n_1 ← n_a;
7  |  Se o(n_b, p, n_a) > 1 e o(n_a, p, n_b) == 1 então
8  |  |  n_1 ← n_b;
9  |  fim
10 |  tNodes ← {n_1};
11 |  tEdges ← {};
12 |  tSize ← s(n_1);
13 |  Occ↓(n_1) ← 1;
14 |  Occ↑(n_1) ← 1;
15 |  border ← {(n_1, p, n_b)|n_b ∈ allNodes};
16 |  allNodes ← allNodes − {n_1};
17 |  Enquanto tSize < Γ e border! = {} faça
18 |  |  (n_1, p, n_b) ← extrair aresta de border com a mais alta afinidade, onde n_1 ∈ tNodes;
19 |  |  n_b Edges ← {(n, p_b, n_b) ∈ allEdges|n ∈ allNodes};
20 |  |  Se para toda aresta e ∈ n_b Edges: aff(e) ≤ aff(n_1, p, n_b) então
21 |  |  |  Se n_b ∉ tNodes então
22 |  |  |  |  Occ↑(n_b) ← 0;
23 |  |  |  fim
24 |  |  |  Occ↓(n_b) ← Occ↓(n_1) × o(n_1, p, n_b);
25 |  |  |  Occ↑(n_b) ← Occ↑(n_b) + (Occ↑(n_1) × o(n_b, p, n_1));
26 |  |  |  Se s(n_b) × Occ↓(n_b) + tSize ≤ Γ e Occ↑(n_b) ⩽ u(n_b) então
27 |  |  |  |  Se n_b ∉ tNodes então
28 |  |  |  |  |  tNodes ← tNodes ∪ {n_b};
29 |  |  |  |  |  allNodes ← allNodes − {n_b};
30 |  |  |  |  fim
31 |  |  |  |  tEdges ← tEdges ∪ {(n_1, p, n_b)};
32 |  |  |  |  border ← border ∪ n_b Edges;
33 |  |  |  |  tSize ← tSize + s(n_b) × Occ↓(n_b);
34 |  |  |  fim
35 |  |  fim
36 |  fim
37 |  T ← T ∪ {(tNodes, tEdges)};
38 |  allEdges ← allEdges − tEdges;
39 enquanto allNodes! = {};
```

pode estar associada a N instâncias de *Offer*, porém uma instância de *Offer* está associada a apenas uma instância de *Vendor*. Por esta razão, se a opção de aninhamento fosse tornar *Offer* como raiz de t_2, uma mesma instância de *Vendor* apareceria replicada em diversas instâncias de *Offer*. Observe que assumindo *Vendor* como raiz isto não acontece.

Um novo fragmento é gerado pelo processamento de arestas (n_1, p, n_b) em *border* como segue: n_b é somente considerado para ser inserido no fragmento corrente se ele estiver relacionado com algum nó já inserido no fragmento corrente através de uma aresta com afinidade mais alta do que qualquer outra aresta ligada a n_b (Linhas 19-20). De acordo com a Linha 18, nós candidatos são processados em ordem decrescente de afinidade para

formar o *template* de fragmento corrente com nós relacionados. Ao final, todos os nós terão sido atribuídos a algum dos *templates* de fragmento. Porém, antes de inserir novos nós em $tNodes$ verifica-se se sua inclusão não excederá o limiar Γ dado o tamanho e ocorrência do nó a ser incluído, bem como verifica-se se a quantidade de ocorrências replicadas para o nó não excede o limiar apontado por u (Linhas 24-26).

Há a possibilidade de existir mais de uma aresta envolvendo dois nós em uma mesma direção. Neste caso, inicializa-se o valor de $Occ \uparrow (n_b)$ com zero (Linhas 21-23) quando a primeira aresta for inserida no *template* de fragmento corrente. À medida que as demais arestas vão sendo inseridas nesta mesma direção, considera-se as ocorrências replicadas já computadas para $Occ \uparrow (n_b)$ na Linha 25. Desta forma é possível contabilizar o número esperado de réplicas para todas as arestas envolvendo n_1 e n_b. Embora n_b possa ser processado mais de uma vez, ele é inserido apenas uma única vez no *template* (Linhas 27-30).

Como um exemplo de execução do algoritmo, considere o grafo de afinidades da Figura 4.4(b) e $\Gamma = 20$ como parâmetros de entrada do algoritmo *affFrag*. Considere ainda que para qualquer nó n em $\mathcal{A}$, $u(n) = 100$. A primeira aresta a ser processada é aquela com a mais alta afinidade envolvendo os nós *Product* e *label*. *Product* é inserido no *template* t_1 como o nó raiz. O tamanho de t_1 assume inicialmente 1, dada a asserção que todos os nós possuem tamanho 1. Uma vez que o limiar de armazenamento não foi atingido, nós continuam a ser inseridos em t_1 dentre aqueles conectados a *Product* que estão mantidos em *border*. Dentre estes, o nó com mais alta afinidade é *label*. Tal nó é inserido em t_1, uma vez que ele não está sendo conectado por qualquer outra aresta com afinidade mais alta e sua inserção não excede Γ. O mesmo comportamento se dá na inserção dos nós *dueDate*, *ProdFeature* e *pflabel* em t_1. Neste ponto, *tSize = 19* dada a ocorrência simples de *dueData* e *label* com a múltipla ocorrência de *ProdFeature* e *pflabel*. O número de réplicas geradas pelo aninhamento de *ProductFeature* e *pflabel* corresponde a 3 unidades cada, o que não impede a inserção destes nós em t_1. As próximas arestas a considerar em *border* relacionam *Product* a *Offer* e *price*. No entanto, a adição destes nós excederia o Γ

dada a múltipla ocorrência de *Product* para *Offer*. Assim, o primeiro *template* é criado com os nós *Product*, *label*, *dueDate*, *ProdFeature* e *pflabel*. Um processo similar cria o segundo *template* com *Offer*, *price*, *Vendor* e *vlabel*. O *template* de fragmentação final gerado é apresentado pela Figura 4.5.

O *template* de fragmentação define como fragmentar instâncias de uma *Estrutura RDF*, isto é, um grafo RDF. Assim, um fragmento é gerado para cada instância do nó raiz de acordo com o *template* $t_i \in T$. No exemplo, t_1 deve gerar fragmentos para cada instância da classe *Product* como demonstrado pela Figura 4.6(a). Embora o algoritmo *affFrag* efetue o corte do grafo de afinidades baseado nas relações de alta afinidade, nós localizados em fragmentos diferentes podem ainda manter uma forte relação de afinidade. Isto ocorre porque o processo de fragmentação foi projetado para satisfazer a um limiar de armazenamento que potencialmente pode cortar arestas com afinidades expressivas. Para evitar a comunicação entre servidores quando uma consulta envolve dados de diferentes fragmentos, procura-se co-alocar tais fragmentos em um mesmo servidor sempre que possível. A seção seguinte apresenta o método de alocação que define como fragmentos são co-alocados em grupos.

4.4 Alocação de Fragmentos

Dado que um fragmento corresponde a uma unidade de armazenamento, o problema da alocação envolve determinar quais fragmentos devem ser alocados em um mesmo servidor. Esta seção define o problema da alocação de fragmentos, tratado pelo Algoritmo *affAlloc*. A estratégia de alocação é definida sobre arestas não incluídas por quaisquer dos *templates* de fragmentação e que definem como grupos de fragmentos deverão ser formados e co-alocados pelo repositório de dados distribuído.

4.4.1 O Problema da Alocação de Fragmentos

Dado um conjunto de *templates* de fragmentos T e um grafo de afinidade $\mathcal{A} = (N, \hat{E}, aff)$, pretende-se obter um conjunto de *links* de alocação $U = \{e_1, ..., e_m\}$, $m \geqslant 1$, de forma que cada $e_i \in \hat{E}$ é uma aresta capaz de conectar *templates* em T através de arestas não incluídas por nenhum dos *templates*. Assim, se fragmentos extraídos a partir de dois *templates* t_1 e t_2 em T estão definidos para serem co-alocados, então existe uma aresta $(n_1, p, n_2) \in U$ tal que n_1 é um nó de t_1 e n_2 é um nó de t_2. Neste contexto, admite-se $t_1 ! = t_2$ ou $t_1 = t_2$. Desta forma é possível co-alocar fragmentos de um mesmo *template* ou a *templates* diferentes.

Dentre as arestas capazes de conectar fragmentos em T, aplica-se uma restrição relacionada à composição de aninhamento estabelecida pelos *templates*. Considere como um exemplo os fragmentos da Figura 4.6 e o grafo de afinidades da Figura 4.4(b). Observe que a aresta (*Product*, *offer*, *Offer*) poderia ser considerada para a co-alocação de nós da classe *Product* com os nós relacionados de *Offer*. Porém, ao tentar agrupar *product1* a *offer1*, além de *offer1*, uma oferta relacionada a *product3* seria incluída no agrupamento. Na sequência, a tentativa de agrupar *product3* as suas ofertas incorre em uma replicação desnecessária de *offer1* e de *offer4*, bem como das respectivas ascendências (*vendor1* e *vendor2*). Por este motivo, assume-se apenas arestas que conectam fragmentos cujo nó destino é um nó raiz de um *template*. A este conjunto de arestas restritas, dá-se o nome de *links*. A função $root(T)$ mapeia T para um conjunto dos nós que atuam como raiz de seus *templates*. Seja *TEdges* o conjunto de arestas contidas em *templates* de T, o conjunto de *links* é dado por $lEdges = \{(n_1, p, n_2) \notin TEdges | n_2 \in root(T)\}$, onde n_1 é um nó contido em quaisquer dos *templates* de fragmentação.

O problema de alocação consiste em encontrar U de forma que a seguinte condição seja satisfeita: dadas as arestas (n, p, n_b) e (n', p', n_b) em *lEdges*, $(n, p, n_b) \in U$ e $(n', p', n_b) \notin U$ se $aff(n, p, n_b) \geq aff(n', p', n_b)$. Esta condição estabelece que se existir mais de uma opção para o aninhamento de um determinado nó n_b, apenas o *link* respectivo que detém a mais alta afinidade será assumido para definir o grupo de alocação de n_b. Este tratamento evita

a geração de replicação excessiva dando preferência e exclusividade para o aninhamento através da aresta com a mais alta afinidade.

Como exemplo, considere o grafo de afinidade da Figura 4.4(b) e o agrupamento de *templates* apresentado pela Figura 4.7. Este agrupamento satisfaz a condição porque $aff(Offer, offer, Product) > aff(ProdFeature, feature, Product)$. A inclusão da aresta ($Prod\ Feature, feature, Product$) é evitada uma vez que *Product* já está sendo considerado pela inclusão da aresta ($Offer, offer, Product$). O Algoritmo *affAlloc* proposto pela seção seguinte é definido de forma que *links* com maior afinidade sejam avaliados por primeiro.

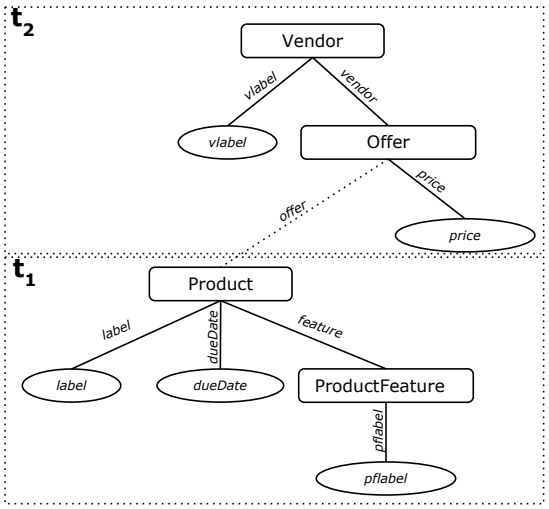


Figura 4.7: *Link* de Alocação entre *Templates* de Fragmentos

4.4.2 O Algoritmo *affAlloc*

Novamente, o grafo de afinidades é utilizado como entrada para o algoritmo de alocação. Neste caso, as arestas não incluídas nos *templates* de fragmentação (Linha 4) são consideradas para proceder à co-alocação de fragmentos. Uma vez que o nó raiz de um *template* de fragmentação é a base para o aninhamento dos demais nós que compõem o fragmento, apenas consideram-se as arestas remanescentes que tem como alvo um nó raiz de fragmento, isto é, arestas caracterizadas como *links*. O algoritmo processa estes *links* em ordem descendente de afinidade (Linha 7), o que colabora para a garantia da condição do problema de alocação introduzido pela seção anterior. Além disto, a variável auxiliar

tNodes controla a inclusão de nós que atuam como destino dos *links* (Linhas 5, 7 e 9), isto é, não permitindo que um determinado fragmento seja aninhado por mais de um *link*. Ao final da execução, grupos de alocação são determinados pelas arestas que devem ser percorridas para co-alocar fragmentos relacionados, isto é, *links* de alocação.

Algoritmo affAlloc

Entrada: *Templates* de Fragmentação T e Grafo de Afinidades $\mathcal{A} = (N, \hat{E}, aff)$
Saída: Conjunto de *Links* de alocação U

```
1  U ← {};
2  TEdges ← arestas de T;
3  rNodes ← root(T);
4  lEdges ← {(n_a, p, n_b) ∈ Ê | (n_a, p, n_b) ∉ TEdges, n_b ∈ rNodes};
5  tNodes ← {n' | (n, p, n') ∈ lEdges};
6  Enquanto lEdges! = {} faça
7  |   (n_a, p, n_b) ← extrair aresta de lEdges com a mais alta afinidade, tal que n_b ∈ tNodes;
8  |   U ← U ∪ (n_a, p, n_b);
9  |   tNodes ← tNodes − {n_b};
10 fim
```

Considere como exemplo o grafo de afinidades da Figura 4.4(b) e os *templates* e fragmentos das Figuras 4.5 e 4.6. Observa-se que as únicas associações não processadas do grafo de afinidades referem-se a (*Product*, *offer*, *Offer*), (*Offer*, *offer*, *Product*) e (*Prod Feature*, *feature*, *Product*). De acordo com o algoritmo, a associação (*Product*, *offer*, *Offer*) é descartada como *link* pois *Offer* não constitui um nó raiz de t_2. Portanto, apenas as demais arestas são caracterizadas como *links*. O *link* (*Offer*, *offer*, *Product*) é processado primeiro, devido a sua mais alta afinidade, e passa a definir a co-alocação de fragmentos de t_1 e t_2 através da aresta *offer*. Na sequência, o *link* (*ProdFeature*, *feature*, *Product*) é descartado pois *Product* já se encontra aninhado pelo link (*Offer*, *offer*, *Product*) inicialmente processado. O resultado da execução do Algoritmo *affAlloc* corresponde ao esquema da Figura 4.7.

Grupos de alocação de fragmentos definidos pelo *link* da Figura 4.7 podem ser representados pelas formas pontilhadas da Figura 4.8. Observe que neste caso em particular, o fragmento referente a *product3* se encontra replicado nos dois primeiros grupos devido a sua associação multi-valorada com *Offer*. O procedimento que define a geração de fragmentos a partir de *templates*, bem como o agrupamento destes em grupos, é definido pelo Algoritmo *affPart* na seção seguinte. Dentre outras questões, o algoritmo habilita, ou

não, a produção de fragmentos replicados como mencionado no exemplo através de um controle de replicação similar ao aplicado pelo Algoritmo *affFrag*.

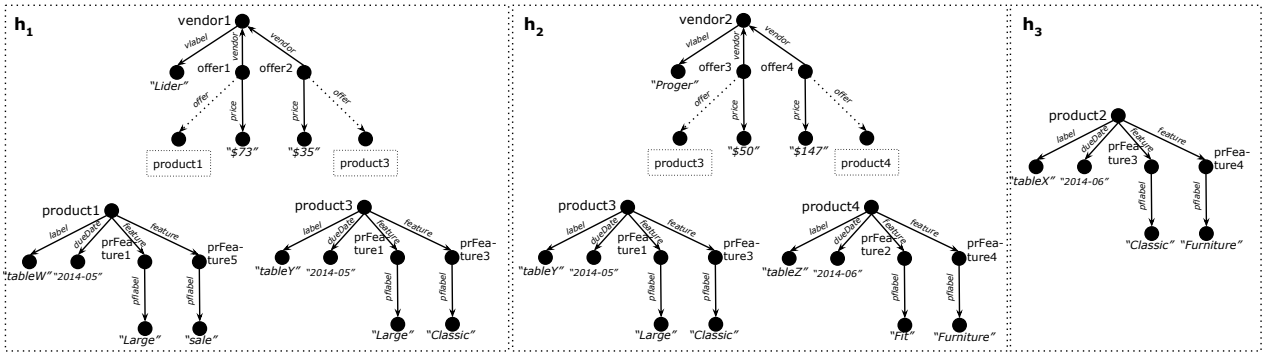


Figura 4.8: Grupos para Alocação de Fragmentos

4.5 O Algoritmo *affPart*

O Algoritmo *affPart* define como os fragmentos são extraídos de um grafo RDF a partir de *templates* de fragmentação, assim como o procedimento de co-alocação de fragmentos a partir dos *Links* de alocação. O algoritmo é composto pelo procedimento *extractFrags* destinado à extração de grupos de fragmentos a serem co-alocados. A extração de fragmentos parte de nós que correspondem à raiz de cada um dos *templates* de fragmentação (Linha 3). Observe que um nó raiz é também uma classe de uma *Estrutura RDF*, o que permite obter as diversas instâncias desta classe através das associações com a propriedade *type* no grafo RDF. Cada uma das chamadas ao procedimento *extractFrags* no algoritmo *affPart* (Linha 6) produz um conjunto de fragmentos F a ser co-alocado pelo repositório de dados distribuído. A cada um destes conjuntos de fragmentos dá-se o nome de *grupo de alocação*, sendo o conjunto destes grupos representado por H. A cada novo grupo de alocação extraído, as arestas e nós processados do grupo são eliminados do grafo (Linha 9) e do conjunto de nós do tipo raiz (Linha 10). O parâmetro de entrada φ corresponde a uma função utilizada para o controle da replicação de dados. Detalhes sobre este controle são fornecidos adiante.

O Procedimento *extractFrags* extrai os fragmentos e forma grupos de alocação através

Algoritmo affPart

Entrada: *Templates* de Fragmentação T, *Links* de Alocação U, Grafo RDF D, Γ e φ
Saída: Conjunto de grupos de alocação H

```
1  H ← {};
2  D' ← D;
3  rNodes ← ⟨n|(n, "type", r) ∈ D', r ∈ root_ord(T)⟩;
4  Enquanto rNodes! = {} faça
5  |   n ← extrair nó de rNodes;
6  |   F ← extractFrags(n, rNodes, T, U, D', Γ, φ);
7  |   H ← H ∪ {F};
8  |   (fNodes, fEdges) ←nós e arestas de F;
9  |   D' ← D' − fEdges;
10 |   rNodes ← rNodes − fNodes;
11 fim
```

de chamadas recursivas ao procedimento (Linha 32). Cada execução do laço das Linhas 7-39 corresponde à criação de um fragmento, bem como dos fragmentos de seu grupo de alocação. O procedimento inicia a partir de um nó raiz n para a criação de um fragmento que inclui nós e arestas associadas a ele pelo grafo RDF e que formam a estrutura determinada pelo seu *template* de fragmentação. A função $root_{ord}(T)$ mapeia T para uma lista de nós ordenados que atuam como raiz de seus *templates*. A ordenação é estabelecida de forma a corresponder à hierarquia estabelecida por *links* de alocação definidos por U. A ordem dos elementos de $root_{ord}(T)$ é definida da seguinte forma: se existir uma aresta $(n_1, p, r_2) \in U$ tal que r_1 é o nó raiz do *template* em que n_1 está incluído e r_2 é o nó raiz de seu próprio *template*, então r_1 precede r_2 em $root_{ord}(T)$.

A variável auxiliar *nEdges* é utilizada para manter estas arestas associadas, dentre as quais as arestas incidentes em n são mantidas por *inEdges* e as que partem de n em *outEdges*. Todos os nós contidos em associações mantidas em *outEdges* são incluídos no fragmento mesmo que não estejam definidos pelo *template*, o que permite que as associações presentes no grafo RDF possam ser mantidas e percorridas mesmo que os nós estejam em fragmentos distintos. A exclusão das arestas processadas na Linha 24 impede que o algoritmo percorra ciclos indefinidamente sobre arestas do grafo RDF.

Esta diferenciação entre arestas *inEdges* e *outEdges* é importante, especialmente, para identificar arestas cujo sentido no grafo de afinidades não corresponde ao sentido no grafo RDF (*inEdges*), mas que podem ser utilizadas para determinar a estrutura de aninhamento do fragmento de acordo com seu *template*. No caso de uma aresta corresponder a

uma associação do *template* (Linha 25), ocorre o caminhamento no grafo RDF a partir da inclusão das adjacências destes nós (Linhas 26-29). Observe que no caso de uma aresta em *inEdges*, é necessário inverter a direção (Linha 17) para que seja possível o caminhamento no grafo RDF a partir do nó corrente. Assim, a estrutura do fragmento assume a estrutura do seu respectivo *template*.

Os fragmentos da Figura 4.6 correspondem a fragmentos gerados a partir dos *templates* da Figura 4.5. Observe que embora a associação *offer* não esteja sendo considerada pelo *template* t_2, a aresta e nós adjacentes são incluídos nos fragmentos correspondentes para que seja possível recuperar a associação entre instâncias de *Offer* e *Product* dispostos em fragmentos distintos.

As variáveis auxiliares *fNodes* e *fEdges* são utilizadas para manter os nós e arestas que estão sendo incluídos no fragmento que está sendo formado. Idealmente, um fragmento deve ser gerado para cada um dos nós das classes definidas como raiz dos *templates*. No entanto, é possível que mais fragmentos sejam necessários uma vez que o tamanho dos nós e as ocorrências dos relacionamentos multi-valorados podem variar no grafo RDF. É importante lembrar que tanto o tamanho quanto as ocorrências consideradas na *Estrutura RDF* correspondem a valores médios fornecidos por uma carga de trabalho prevista. Estes valores são utilizados para contabilizar o tamanho previsto sobre um *template* de fragmento. No procedimento *extractFrags*, a função *nSize* é utilizada para determinar o tamanho real ocupado por um nó de um grafo RDF, enquanto a variável *fSize* mantém e controla o tamanho do fragmento que está sendo formado.

As associações referentes a *links* de alocação iniciam uma chamada recursiva ao procedimento para a criação do fragmento a ser co-alocado (Linha 33). Esta chamada e a respectiva criação do fragmento são evitadas caso o nó em questão (n_{target}) já tenha sido incluído por outro agrupamento e a redundância de dados extrapola o limiar u para qualquer dos nós contidos neste fragmento (Linha 32). O limiar u foi introduzido pela Seção 4.3.1 e aplicado no Algoritmo *affFrag* sobre a função $Occ \uparrow (n)$, que mapeia cada nó em um *template* para o número permitido de ocorrências replicadas em seus respectivos

Procedimento extractFrags

Entrada: Nó n, Nós raiz a serem processados *queueNodes*, *Templates* de Fragmentação T, *Links* de Alocação U, Grafo RDF D, Γ e φ
Saída: Conjunto de fragmentos F

```
1  F ← {};
2  D' ← D;
3  TEdges ← arestas de T;
4  inEdges ← {(n_b, p, n) ∈ D'};
5  outEdges ← {(n, p, n_b) ∈ D'};
6  nEdges ← inEdges ∪ outEdges;
7  Enquanto nEdges! = {} faça
8      fNodes ← {};
9      fEdges ← {};
10     fSize ← 0;
11     Enquanto nEdges! = {} e fSize ⩽ Γ faça
12         (n_a, p, n_b) ← aresta de nEdges, onde (n_a, p, n_b) ∉ fEdges;
13         class_a ← c_a, tal que (n_a, "type", c_a) ∈ D';
14         class_b ← c_b, tal que (n_b, "type", c_b) ∈ D';
15         n_target ← n_b;
16         Se (n_a, p, n_b) ∈ inEdges então
17             n_target ← n_a;
18         fim
19         Se fSize + nSize(n_b) ⩽ Γ então
20             fNodes ← fNodes ∪ {n_a, n_b};
21             fEdges ← fEdges ∪ {(n_a, p, n_b)};
22             fSize ← fSize + nSize(n_b);
23         fim
24         D' ← D' − {(n_a, p, n_b)};
25         Se (class_a, p, class_b) ∈ TEdges então
26             inEdges ← inEdges ∪ {(n', p', n_target) ∈ D'};
27             outEdges ← outEdges ∪ {(n_target, p', n') ∈ D'};
28             nEdges ← nEdges ∪ inEdges ∪ outEdges;
29             TEdges ← TEdges − (class_a, p, class_b);
30         fim
31         Se (class_a, p, class_b) ∈ U então
32             Se n_target ∈ queueNodes ou !φ(class_a, p, class_b) então
33                 F ← F ∪ extractFrags(n_target, T, U, D', Γ, φ);
34             fim
35         fim
36     fim
37     queueNodes ← queueNodes − fNodes;
38     F ← F ∪ {(fNodes, fEdges)};
39 fim
```

fragmentos. Em *extractFrags*, uma função φ é assumida para mapear o nó n_{target} para um valor lógico que indica se ele poderá ser replicado. Este mapeamento é definido pelo Procedimento *redundancyFree*. De forma análoga à verificação aplicada pelo Algoritmo *affFrag*, o Procedimento *redundancyFree* calcula a medida $Occ \uparrow(n_{target})$ e aplica o limiar de redundância sobre cada um dos nós do *template* em que n_{target} está inserido. Caso exista um nó que exceda ao seu u, o procedimento retorna o valor lógico "Sim" indicando que a redundância deve ser evitada. Caso contrário, a replicação do fragmento é autorizada no grupo de alocação corrente. Os grupos de alocação da Figura 4.8 correspondem a uma formação em que a redundância do fragmento referente a *product3* foi habilitada.

Embora esta verificação seja similar à aplicada pelo Algoritmo *affFrag*, optou-se por criar um procedimento específico dado que no Algoritmo *affFrag* ela é executada sobre *templates* em formação, o que não corresponde à necessidade do procedimento *extractFrags*.

Procedimento redundancyFree

```
   Entrada: Aresta (n0, p, n1), Templates de Fragmentação T, Estrutura RDF S = (C, L, l, A, s, o), Γ e u
   Saída: "Sim" caso livre de redundância, e "Não"caso contrário
 1 t ← template de n1;
 2 tEdges ← arestas de t;
 3 Occ ↑ (n1) ← o(n1, p, n0);
 4 Se Occ ↑ (n1) > u(n1) então
 5 |   retornar "Sim";
 6 fim
 7 border ← {(n1, p', n2) ∈ tEdges};
 8 allNodes ← {n1};
 9 Enquanto border! = {} faça
10 |   (n1, p', n2) ← extrair aresta de border;
11 |   Se n2 ∉ allNodes então
12 |   |   Occ ↑ (n2) ← 0;
13 |   fim
14 |   Occ ↑ (n2) ← Occ ↑ (n2) + (Occ ↑ (n1) × o(n2, p', n1);
15 |   Se Occ ↑ (n2) > u(n2) então
16 |   |   retornar "Sim";
17 |   fim
18 |   border ← border ∪ {(n2, p'', n3) ∈ tEdges|n3 ∉ allNodes};
19 |   allNodes ← allNodes ∪ {n2};
20 fim
21 retornar "Não";
```

Considere como exemplo de execução do algoritmo *affPart*, e de seus procedimentos, a extração dos fragmentos do grafo RDF da Figura 4.1 a partir dos *templates* de fragmentação da Figura 4.5 e dos *links* de alocação da Figura 4.7. Nós do grafo RDF que correspondem a raiz dos *templates* são obtidos pela Linha 3 do algoritmo *affPart*. Neste caso, os nós recuperados correspondem a *vendor1*, *vendor2*, *product1*, *product2*, *product3* e *product4*. Iniciando por *vendor1*, o procedimento *extractFrags* é acionado pela Linha 6. As adjacências de *vendor1* são mantidas pela variável *nEdges* na Linha 6 de *extractFrags*, que corresponde ao conjunto das arestas em que *vendor1* atua como alvo (*inEdges* - Linha 4) ou origem (*outEdges* - Linha 5). As arestas em *nEdges* e seus respectivos nós são incluídos no fragmento corrente caso seu tamanho não exceder Γ (Linhas 19-23). Assim, a aresta *vlabel* e o nó *"Lider"* são adicionados ao fragmento de *vendor1*. As adjacências de *vendor1* com instâncias de *Offer* são também adicionadas, porém, como *vendor1* atua como nó alvo, o sentido de navegação precisa ser alterado (Linha 17). Neste caso, como *offer1* e *offer2* correspondem a arestas do *template* relacionado, as adjacências de *offer1*

e *offer2* são adicionadas nas Linhas 26-28 para que sejam incluídas no fragmento. Como fazem parte do *template*, os preços das ofertas são incluídos no fragmento. As adjacências de *offer1* com *product1* e *offer2* com *product3* são também adicionadas, apesar de não definidas pelo *template*, para que se possa manter a navegabilidade entre os fragmentos. No entanto, estas adjacências correspondem a *links* de alocação de acordo com a Figura 4.7. Caso não existam fragmentos gerados para *product1* ou *product3* (Linha 32), seus respectivos fragmentos são gerados pela chamada recursiva ao procedimento, e unidos aos fragmentos gerados a partir de *vendor1* (Linha 33). Desta forma, o primeiro grupo de alocação h_1 é formado conforme apresentado pela Figura 4.8.

O mesmo procedimento é adotado para gerar o grupo de alocação h_2 a partir de *vendor2*. Neste caso, ao tentar novamente processar *product3* verifica-se que um fragmento já foi gerado para este nó. Para determinar se é possível a geração de um novo fragmento para *product3*, o valor lógico retornado pela função φ é utilizado (Linha 32 de *extractFrags*), que por sua vez foi predeterminado pelo procedimento *redundancyFree*. Este procedimento verifica se a própria estrutura de aninhamento do *link* de alocação está permitindo a replicação de *product3* (Linhas 3-6) ou se a replicação é gerada pelo aninhamento definido pelo *template* de *product3* (Linhas 7-20). Caso o limiar de replicação u esteja sendo excedido para qualquer nó no aninhamento, o procedimento retorna o valor lógico "Sim" indicando que a replicação de *product3* não será permitida. De acordo com as ocorrências definidas pelas *Estrutura RDF* da Figura 4.3, o aninhamento de uma instância da classe *Product* está associada, em média, a 17 instâncias de *Offer*. Assim, 17 réplicas de um nó da classe *Product* podem ser geradas ao aninhá-lo as suas 17 associações com instâncias de *Offer* através do *link* de alocação. Na estrutura definida a partir de *product3* e que corresponde ao *template* t_1, o aninhamento de uma instância da classe *ProdFeature* produz 3 réplicas pelo seu aninhamento com *Product*. Assim, as 17 réplicas de *product3* levariam a 51 réplicas de *ProdFeature*. Caso o limiar de replicação para *ProdFeature* tenha sido definido como $u(ProdFeature) = 10$, por exemplo, a replicação de *product3* não seria permitida para o grupo de alocação h_2.

4.6 Considerações sobre o Particionamento de dados XML

O particionamento de dados XML corresponde a um sub-conjunto da solução apresentada por este capítulo. Portanto, é possível aplicar a metodologia proposta para o modelo RDF com algumas simplificações para o modelo XML. Dentre as características diferenciadas do XML, destaca-se o modelo em árvore sobre o qual seus documentos são formados, em contra-ponto ao modelo de grafos aplicado pelo RDF. Esta composição torna desnecessárias as tratativas referentes ao controle de redundância aplicadas pelos algoritmos *affFrag* e *affPart*, dado que um determinado nó XML só pode estar associado a no máximo um nó pai. Outro tratamento desnecessário diz respeito a ciclos. Embora em um esquema XML seja possível assumir ciclos, especialmente para auto-relacionamentos de itens de dado, nos documentos XML as instâncias respectivas a ciclos de um item de dado só poderão corresponder a associações entre nós diferentes. Em resumo, a composição estrutural do XML consegue simplificar tanto a definição de *templates* de fragmentos quando da extração destes em grupos de alocação. Um fragmento XML constitui uma sub-árvore de um documento XML que representa seu conjunto de dados.

Consultas sobre fontes de dados XML são estabelecidas através da linguagem XQuery[1] e de expressões em XPath[2]. A recuperação de dados é efetuada através de caminhos percorridos sobre seus documentos. Diferente dos *padrões de grafo* assumidos para a linguagem SPARQL, consultas sobre dados XML obedecem a estrutura de aninhamento de seus documentos. Em RDF, mesmo assumindo um grafo direcionado, *padrões de grafos* podem ser estabelecidos no sentido inverso das arestas presentes no grafo RDF, e executados através de casamentos de seus subgrafos. O caminhamento sobre a estrutura XML permite trabalhar sobre um grafo de afinidades não-direcionado. Além de simplificar o processo de particionamento, esta condição potencialmente reduz o conjunto de possibilidades de particionamento a serem avaliadas.

O método de particionamento proposto por esta tese corresponde a uma extensão de

[1] http://www.w3.org/TR/xquery/
[2] http://www.w3.org/TR/xpath/

uma solução para o particionamento XML desenvolvida durante o programa de doutorado e descrita em [Schroeder et al., 2012]. Durante a evolução da proposta, optou-se por eleger o modelo RDF como base para uma metodologia de particionamento em virtude da ascensão expressiva do uso deste modelo, especialmente para repositórios de grande porte. Além disto, RDF atua como um modelo genérico e compatível com uma série de outros modelos, incluindo o XML. Alguns resultados experimentais sobre bases de dados XML são apresentadas pela Seção 6.

4.7 Segmentação de Consultas sobre Grupos de Alocação

Conforme definido pela Equação 4.2, o objetivo do particionamento é minimizar a segmentação dos dados que correspondem ao resultado de consultas sobre às partições do repositório. A segmentação deve ser principalmente minimizada para as consultas mais frequentes da carga de trabalho. Para tanto, encontrar um particionamento $\mathcal{P}$ que responde a este objetivo é o foco da solução apresentada. Esta seção descreve como o objetivo de particionamento é tratado por esta solução.

Em um particionamento $\mathcal{P} = \{P_1, ..., P_m\}$, cada P_i é formado por grupos de alocação. Desta forma, cada $P_i \in \mathcal{P}$ é definido por $P_i = H'$, tal que $H' \subseteq H$. Seja uma consulta q submetida ao repositório, o resultado de q que representa os casamentos de seu padrão de grafo é definido por $B(q) = \{b_1, ..., b_n\}$, tal que b_i é um subgrafo do grafo RDF, isto é, $b_i \subseteq D$. Como discutido anteriormente, cada elemento de $B(q)$ deve estar idealmente contido em uma única partição, isto é, não segmentado. A segmentação de cada $b_i \in B(q)$ é dada pela quantidade de partições adicionais que mantém suas triplas. Dado que $E(H')$ representa o conjunto de triplas de uma partição definida por H', a segmentação de cada elemento de $B(q)$ sobre um repositório particionado é dada pela função $seg(b_i)$ definida a seguir:

$$seg(b_i) = 1 - \left|\{H' \in \mathcal{P} | b_i \cap E(H') \neq \oslash\}\right| \tag{4.3}$$

Desta forma, a medida $\hat{P}$ introduzida pela Definição 4.1.2, corresponde ao somatório dos resultados produzidos pela segmentação de todos os elementos do conjunto $B(q)$. Formalmente, $\hat{P} = \sum_{b_i \in B(q)} seg(b_i)$. O processamento de uma consulta q sobre o repositório particionado é desempenhado conforme introduzido pela Seção 2.2. Neste caso, cada elemento $b_i \in B(q)$ deverá ser recuperado por *threads* independentes, que por sua vez devem idealmente acessar apenas um servidor. Isto significa que a segmentação de b_i implica em acessos adicionais a outros servidores que, como visto, devem ser minimizados.

A solução apresentada baseia-se em heurísticas de afinidade entre itens de dados com o objetivo de formar grupos de alocação cuja estrutura corresponda ao padrão de grafo das consultas mais frequentes da carga de trabalho. A estrutura de um grupo de alocação é determinada por *templates* de fragmento em conjunto com *links* de alocação responsáveis por conectar fragmentos. O particionamento RDF proposto por este capítulo foi aplicado sobre um repositório de dados distribuído conforme proposto pela Seção 2.2. Um sistema desenvolvido para compor os elementos desta arquitetura e dar suporte a diferentes estratégias de particionamento de dados RDF é proposto pelo capítulo a seguir, nomeado *ClusterRDF*.

CAPÍTULO 5

CLUSTERRDF

Um sistema foi desenvolvido para dar suporte ao método de particionamento proposto, bem como a outras abordagens, com o objetivo de permitir comparações entre diferentes soluções. *ClusterRDF* corresponde a um sistema que habilita a recuperação de dados de consultas SPARQL sobre uma base de dados RDF particionada em um repositório de dados distribuído. A arquitetura considerada para este processamento é apresentada pela Figura 5.1, e equivale à arquitetura estratificada apresentada pela Seção 2.1.2. Neste caso, o papel do SGBD é simplificado pelos componentes denominados *Coordenador* e *Executor*, responsáveis por processar requisições de consulta.

Um usuário submete sua consulta para um *Coordenador* que gera um plano de execução e o entrega para todos os servidores que mantêm os dados requisitados. Cada servidor executa o plano em paralelo sob a coordenação de um *Executor de Consulta* que obtém os resultados parciais da consulta sobre os subgrafos contidos em suas partições. Ao final, os resultados provenientes de todos os servidores são unidos e enviados como resposta ao cliente pelo *Coordenador*. Idealmente, o particionamento efetuado sobre o repositório deverá contribuir para que cada servidor acesse apenas seu repositório local, de forma que

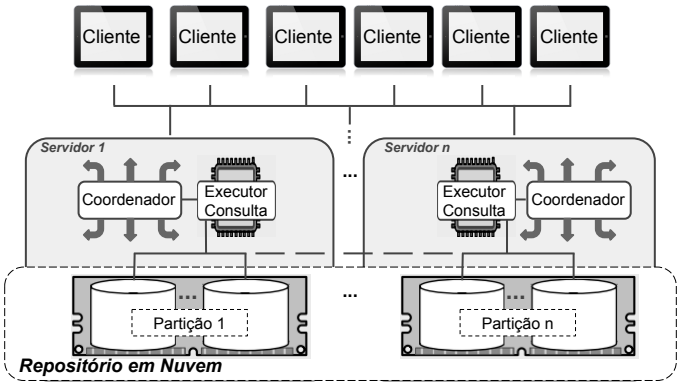


Figura 5.1: Arquitetura de Processamento de Consultas SPARQL

a execução ocorra de maneira independente entre os servidores. No entanto, é possível que um servidor obtenha dados mantidos por outros servidores caso os dados de sua partição local não sejam suficientes para produzir a resposta.

Nesta composição, o sistema *Scalaris* [Schütt et al., 2008] foi escolhido para atuar na base deste sistema, provendo um sistema de armazenamento escalável através de um modelo chave-valor e de uma rede de sobreposição par-a-par estruturada. Neste capítulo são apresentadas as decisões de projeto efetuadas durante a construção de *ClusterRDF*. Em particular, o sistema *Scalaris* é apresentado, juntamente com o método adotado para o armazenamento de bases particionadas e o procedimento de recuperação de dados.

5.1 O Repositório *Scalaris*

O sistema Scalaris [Schütt et al., 2008] define um repositório chave-valor em memória que utiliza uma rede de sobreposição P2P estruturada para alcançar escalabilidade e disponibilidade. O sistema é baseado em tabelas de espalhamento distribuídas (DHT - *Distributed Hash Table*) e provê o suporte a leituras e escritas monotônicas. Sua composição está estruturada pelas camadas de *comunicação*, *replicação*, *transação* e *aplicação*. A camada de comunicação é composta por uma rede de sobreposição estruturada que garante desempenho logarítmico em buscas através do protocolo *Chord#* [Schütt et al., 2007]. A comunicação entre o *Scalaris* e seu sistema de DHT subjacente é realizado por chamadas remotas de procedimento sobre o protocolo HTTP, enquanto a comunicação entre nós *Scalaris* usam um protocolo nativo em *Erlang*. Segundo os autores, as camadas de replicação e transação fornecem disponibilidade através de replicação simétrica e consistência em operações concorrentes de escrita através da integração da rede de sobreposição com uma variação do protocolo de consenso *Paxos Commit* [Rao et al., 2011]. Este protocolo é utilizado para implementar transações sobre múltiplas chaves e também assegurar que todas as réplicas de uma chave sejam atualizadas de forma consistente. A camada de aplicação exporta a interface do repositório construído em Erlang, disponibi-

lizando um serviço de armazenamento escalável e acessível através de APIs Java, Python e Ruby. A interface para o armazenamento de pares chave-valor consiste em operações genéricas de leitura, escrita e remoção de pares chave-valor baseadas no valor das chaves, assim como ocorre para outros sistemas baseados em DHT.

Em geral, repositórios baseados em DHT não oferecem o controle sobre a alocação dos dados, uma vez que o nó do sistema distribuído que armazena um par chave-valor é determinado pelo resultado de uma função *hash* aplicada a sua chave. Desta forma um par chave-valor pode ser armazenado ou recuperado apenas a partir de sua chave, sem que haja a necessidade de acessar outras estruturas de endereçamento para obter a localização dos dados no sistema distribuído. Este mecanismo de endereçamento de dados é um dos principais fatores responsáveis pela escalabilidade destes sistemas. Porém, o projeto físico de banco de dados espera que algum controle sobre a localidade de dados seja fornecido. Este controle é fundamental para que seja possível definir, por exemplo, estratégias para a co-alocação de dados relacionados. Alguns sistemas baseados em DHT como o *Scalaris* efetuam o armazenamento de pares pela ordem lexicográfica das chaves. Desta forma, dois pares que compartilham um mesmo prefixo de chave devem estar localizados em um mesmo servidor do sistema distribuído, ou ainda em servidores próximos. Assim, a definição apropriada para chaves constitui uma forma para a definição de grupos de alocação de pares que devem ser co-alocados no repositório.

Em resumo, a justificativa para o uso do *Scalaris* é dada por duas razões principais. Primeiramente, trata-se de um sistema capaz de conferir disponibilidade e escalabilidade para sistemas de armazenamento em nuvem. A segunda razão se refere à noção de localidade de dados fornecida pelo sistema através da alocação de pares chave-valor que mantêm a ordem lexicográfica de chaves. Até o momento de conclusão desta tese, o *Scalaris* identifica-se como único sistema de armazenamento *open-source* que reúne estas duas características. A habilidade de dar suporte à localidade de dados é essencial para permitir o agrupamento de fragmentos relacionados como proposto pelo presente trabalho. A seção a seguir define como fragmentos são co-alocados através da geração

de chaves apropriadas. Além dos diferenciais mencionados a respeito do *Scalaris*, um recurso que merece destaque é a funcionalidade de empacotamento de requisições a serem submetidas para um servidor em uma única mensagem. Através deste recurso é possível reduzir o custo de envio de mensagens a um mesmo servidor.

5.2 Armazenamento de Dados

O modelo chave-valor do repositório *Scalaris* foi considerado para armazenar fragmentos RDF. Seja F um conjunto de fragmentos tal que $f \in F$ é um subgrafo RDF extraído a partir de um *template* de fragmentação $t \in T$. O repositório de dados armazena fragmentos da seguinte forma: um par chave-valor é gerado para cada fragmento $f \in F$, tal que em cada par a chave contém um identificador único e o valor contém o fragmento RDF propriamente dito na forma de um subgrafo.

Fragmentos são co-alocados pela atribuição de um mesmo prefixo para suas chaves. É importante observar que se o repositório aloca dados em ordem lexicográfica, pares chave-valor que detêm um mesmo prefixo de chave são potencialmente armazenados em um mesmo servidor, ou então em servidores próximos. Para explorar este particionamento por intervalo de chaves do repositório subjacente, as chaves de fragmentos foram definidas de forma a se adequar a abordagem de alocação proposta. Assim, todos os fragmentos de um mesmo grupo de alocação devem possuir um prefixo comum para suas chaves. Para endereçar unicamente os dados, é introduzida uma função *id* que mapeia fragmentos para um identificador único. Seja T o conjunto de *templates* de fragmento, um conjunto de *links* de alocação que determina a relação entre fragmentos é dada por $U = \{e_1, ..., e_m\}$, tal que $m \geqslant 1$ e cada $e_i \in U$ é uma aresta direcionada que conecta dois *templates* em T, conforme definido pela Seção 4.4. A chave de uma fragmento no repositório é definida por uma função *ch* que considera o prefixo de chaves relacionadas por *links* de alocação. Seja n o nó raiz de um fragmento f, a chave de f é recursivamente definida pela função $ch(n)$ da seguinte forma: se existe uma aresta (n_0, p, n) que corresponde a um *link* de

alocação em U, então $ch(n) = ch(n_0)/id(n)$, e $ch(n) = id(n)$ caso contrário. O símbolo "/" é utilizado como operador de concatenação de *strings*. Dada a composição em árvore estabelecida pelos *links* em U, chaves de fragmentos são prefixadas pelas chaves de seus fragmentos ascendentes da árvore. Para tanto, o prefixo compartilhado entre as chaves de fragmentos implica no agrupamento destes fragmentos dado pela ordem lexicográfica.

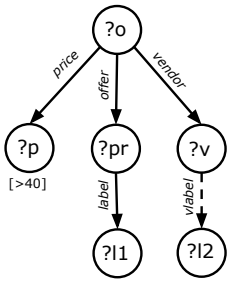
Como um exemplo, considere o *Link* de alocação de fragmentos da Figura 4.7 e os grupos de alocação da Figura 4.8. Seja $f_{vendor1}$, $f_{product1}$ e $f_{product3}$ os fragmentos referentes ao primeiro grupo de alocação (h_1), os respectivos mapeamentos para pares chave-valor são definidos a seguir. Neste caso, o prefixo para os fragmentos de $f_{product1}$ e $f_{product3}$

$$
\begin{aligned}
h_1 \texttt{ = \{ (id(} f_{vendor1} \texttt{), } & f_{vendor1} \texttt{),} \\
\texttt{(id(} f_{vendor1} \texttt{)/id(} f_{product1} \texttt{), } & f_{product1} \texttt{),} \\
\texttt{(id(} f_{vendor1} \texttt{)/id(} f_{product3} \texttt{), } & f_{product3} \texttt{)\}}
\end{aligned}
$$

5.3 Recuperação de Dados

O principal alvo de desenvolvimento de *ClusterRDF* está em dar suporte a avaliações experimentais utilizando diferentes soluções de particionamento. Especificamente, o objetivo é avaliar estas soluções em termos do custo de comunicação de mensagens trocadas entre servidores para a execução de consultas. É importante destacar que o sistema não constitui um *engine* de processamento distribuído para consultas sobre dados RDF, uma vez que questões como otimização de consultas, índices e metadados não estão sendo considerados pelo sistema.

Os componentes da arquitetura idealizada pela Figura 5.1 são implementados por componentes correspondentes no sistema desenvolvido. Uma consulta é submetida a um servidor, cujo *Coordenador* gera um *Plano de Recuperação de Dados* a ser executado, em paralelo, pelos servidores do sistema. Cada servidor executa o plano a partir de sua partição local que é acessada através da requisição de dados à DHT por um nó *Scalaris*. Quando todos os dados são obtidos, cada servidor envia seus resultados parciais



(a) Consulta

Lista de Exploração
<(?o price p) (?o offer ?pr), (?pr label ?l1), (?o vendor ?v), (?v vlabel ?l2)>

(b) Exploração

Figura 5.2: Grafo de Consulta e Lista de Exploração

ao *Coordenador* que iniciou a consulta.

Planos de Recuperação são gerados a partir de uma consulta estabelecida através de um *padrão de grafo* definido por $G = (V, E, r)$. Ao invés de junções custosas utilizadas pela maioria dos sistemas do tipo *triple store*, é assumida a *exploração em grafos* como método de recuperação de dados. Isto é, a execução se inicia pela busca dos casamentos de um nó $v \in V$. A partir de cada casamento, o grafo é explorado para encontrar os casamentos das adjacências de v, e assim prossegue até que todas as arestas em E tenham sido exploradas. Neste processo, a ordem de exploração é importante para reduzir a quantidade de resultados intermediários. Desta forma, se torna adequado definir uma sequência para a exploração de forma que os padrões de tripla mais seletivos de G tenham a preferência na exploração. Um padrão de tripla é dada por $(n_a, p, n_b) \in E$, e corresponde a uma aresta de um padrão de grafo. Assim, *ClusterRDF* assume como entrada uma consulta dada por um padrão de grafo e define o plano de recuperação através de uma lista de exploração de padrões de triplas ordenados pela seletividade dos mesmos. Neste trabalho utilizou-se o método para estimar a seletividade dos padrões definido por *Trinity.RDF* [Zeng et al., 2013]. A Figura 5.2(b) apresenta a lista de exploração para o padrão de grafo da Figura 5.2(a). Esta consulta visa recuperar nós da classe *Offer* com preços superiores a \$40, bem como seus respectivos produtos e vendedores. Neste caso, o primeiro padrão a ser explorado é (?*o*, *price*, ?*p*) em virtude da seletividade gerada pelo filtro sobre os valores de *price*.

Uma vez definida a lista de exploração, o plano de recuperação é inicialmente sub-

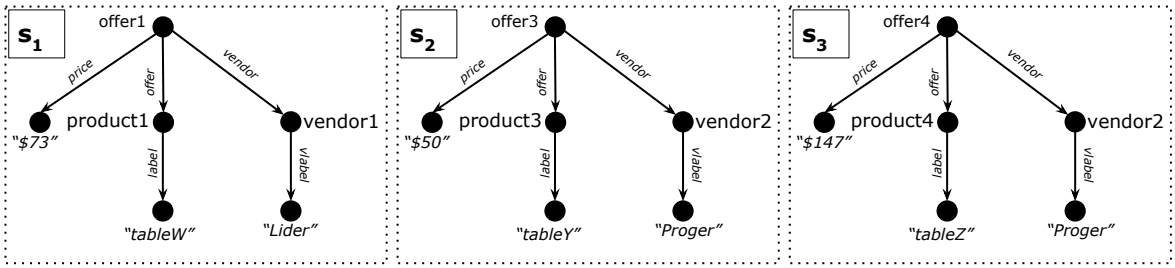


Figura 5.3: Subgrafos de Resposta

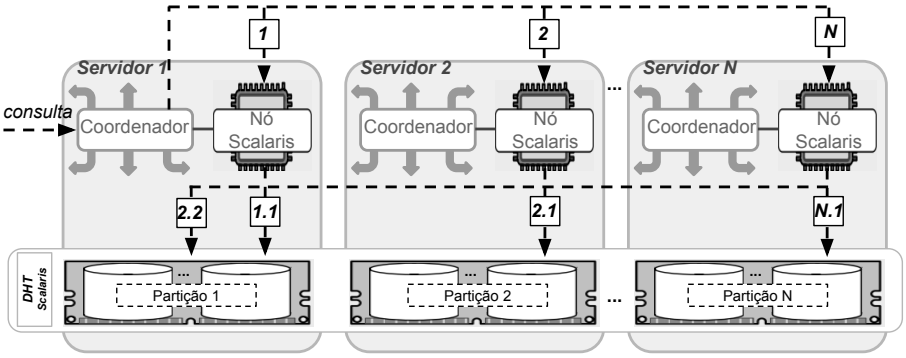


Figura 5.4: Arquitetura para o Processamento no ClusterRDF

metido a todos os servidores. Ao receber o plano, cada servidor inicia a exploração sobre os fragmentos mantidos pelo seu repositório local, e que correspondem a um subgrafo do conjunto de dados RDF global. Durante a exploração, é possível que o servidor obtenha dados mantidos por outros servidores, caso o subgrafo de sua partição local não seja suficiente para produzir a resposta iniciada pela exploração sobre ele, como mostra a Figura 5.4

Como exemplo, considere a consulta apresentada pela Figura 5.2(a), e duas estratégias de agrupamento de fragmentos H_A e H_B representadas pelas Figuras 5.5 e 5.6, respectivamente. Observe que a principal diferença entre elas está na replicação de *product3* em H_B, que não ocorre para H_A. Os subgrafos que representam a resposta da consulta são mostrados pela Figura 5.3. Para exemplificar a execução desta recuperação, assuma que o grupo de fragmentos h_1 é mantido pelo *Servidor1*, e que o grupo h_2 é mantido pelo *Servidor2*. Em primeiro lugar, considere uma execução a partir da estratégia H_A. O plano é submetido paralelamente a cada um dos servidores. Conforme as *threads* apresentadas pela Figura 5.4, a *thread* 1 corresponde à submissão da lista de exploração ao *Servidor1*,

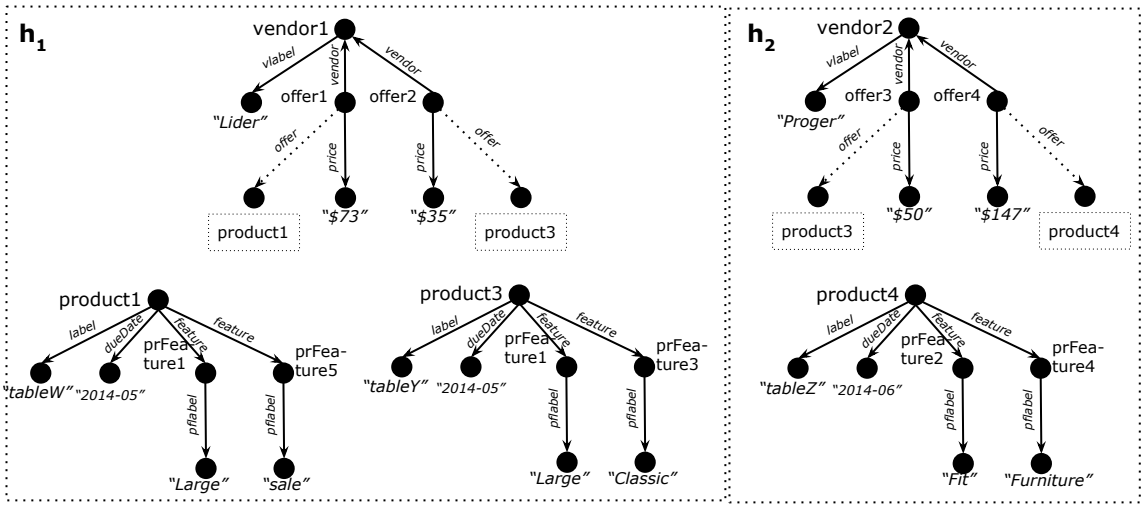


Figura 5.5: Grupos de Alocação de Fragmentos sem Replicação (H_A)

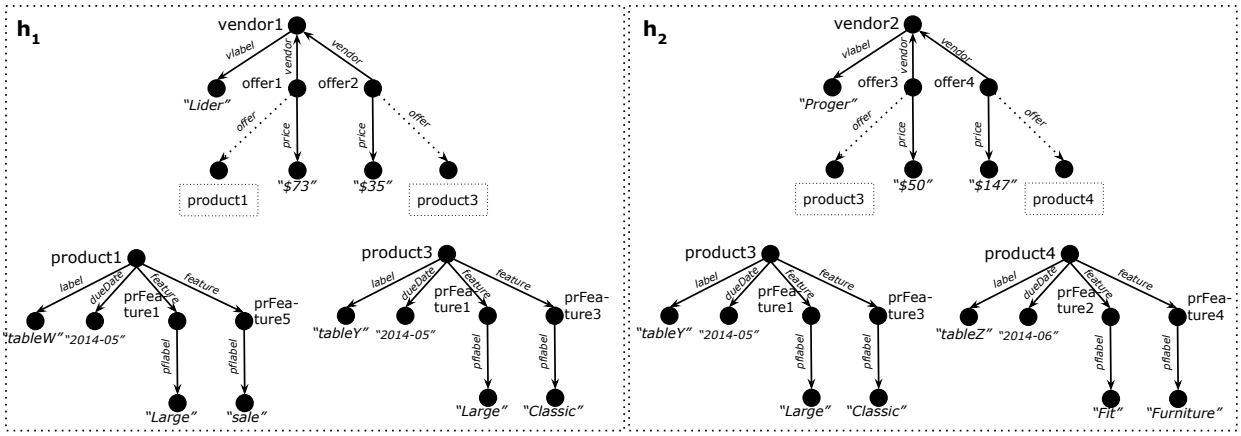


Figura 5.6: Grupos de Alocação de Fragmentos com Replicação (H_B)

enquanto a *thread* 2 submete a lista ao *Servidor2*. O *Servidor1* consegue responder completamente a exploração a partir de seu repositório local ao iniciá-la por *offer1* e *offer2* sobre o fragmento cuja raiz é *vendor1* (requisição 1.1). No caso da *thread* 2, o *Servidor2* executa a exploração a partir de seu fragmento local que contem dados de *offer3* e *offer4* (requisição 2.1). Porém, neste caso é necessário obter os dados de *product3* mantidos apenas pelo *Servidor1*, ocasionando a execução distribuída desta *thread* (requisição 2.2). Observe que se a estratégia de alocação de H_B fosse considerada, o *Servidor2* conseguiria responder completamente a exploração iniciada a partir de *offer3* e *offer4*, uma vez que *product3* se encontra replicado nele. No passo final, os casamentos de todos os servidores são coletados para produzir o resultado final.

A comparação da abordagem de particionamento proposta com abordagens relacionadas é realizada em termos do número de servidores acessados por *threads* individuais conforme apresentado no capítulo seguinte. Embora estas abordagens comparadas tenham sido construídas sobre a arquitetura de processamento genérica de *ClusterRDF*, é importante observar que os resultados reportados são válidos para determinar o custo da troca de mensagens em *engines* de processamento paralelos e distribuídos. Isto ocorre porque o modelo de processamento de consulta não tem efeito na necessidade de recuperação de todos os dados do conjunto de resultados, além disto não são considerados outros custos envolvidos na execução da consulta, como o tempo de CPU.

CAPÍTULO 6

AVALIAÇÃO EXPERIMENTAL

Dois estudos experimentais foram conduzidos para avaliar a solução de particionamento proposta em relação a outros métodos relacionados no Capítulo 3. No primeiro estudo, o experimento avalia a solução de particionamento completa comparando os resultados com métodos relacionados ao particionamento de bases RDF. No segundo estudo, avaliou-se apenas a solução de fragmentação para que fosse possível a comparação com dois métodos de fragmentação também baseados em heurísticas sobre relações de afinidades. Para adequar-se ao modelo destas soluções, optou-se por aplicar a fragmentação sobre bases de dados XML no segundo experimento.

6.1 Avaliação do Particionamento sobre bases de dados RDF

Um estudo experimental foi realizado para determinar o efeito da abordagem de particionamento proposta na recuperação de dados de consultas efetuadas sobre repositórios RDF distribuídos. O *benchmark* SPARQL chamado Berlin (BSBM) foi aplicado para avaliar e comparar o desempenho fornecido pela abordagem proposta, juntamente com as abordagens introduzidas por Huang et al [Jiewen Huang, 2011] e Trinity.RDF [Zeng et al., 2013]. Neste capítulo, a abordagem de particionamento proposta por esta tese é identificada como *ClusterRDF*.

Huang et al. aplica o particionador METIS [Karypis and Kumar, 1998a] sobre um grafo RDF e, em seguida, replica nós para sobrepor dados entre partições de acordo com uma garantia *n-hops*[1]. Esta abordagem é identificada neste estudo experimental como *METIS-2hops* visto que uma versão deste método foi implementada utilizando uma

[1]Recomenda-se a leitura da Seção 3.2.3 para maiores detalhes sobre esta garantia e sobre as abordagens comparadas.

garantia de *2-hops*. Embora o sistema *Trinity.RDF* seja focado em prover um *engine* de consulta para RDF, este sistema considera um particionamento em *hash* sobre nós RDF.

Dentre as abordagens relacionadas, *Metis-2hops* e *Trinity.RDF* foram escolhidas pois representam o estado de maturação atual de sistemas de gerenciamento de dados RDF em nuvem, conforme o levantamento apresentado por [Kaoudi and Manolescu, 2014]. Como indicado pelo Capítulo 3, ambas as soluções baseiam-se em heurísticas sobre a composição estrutural de grafos RDF. Portanto, este estudo tem como objetivo avaliar os resultados produzidos por estratégias de particionamento baseadas essencialmente na composição de grafos em comparação com os resultados produzidos pela aplicação de heurísticas de carga de trabalho, como proposto por *ClusterRDF*. Embora existam outras soluções baseadas na carga de trabalho RDF como mostra o Capítulo 3, a comparação não foi possível devido a forte dependência destas abordagens para com o modelo relacional utilizado por suas *triple stores*. As abordagens comparadas foram implementadas sobre a mesma arquitetura proposta por *ClusterRDF*. Detalhes adicionais da adequação destes sistemas sobre esta arquitetura são fornecidos a seguir.

6.1.1 Abordagens Comparadas

METIS-2hops foi construído originalmente sobre o *triple store* RDF-3X e o *framework* de processamento *Hadoop*. Nos experimentos, foi considerado o seu método de particionamento sobre a arquitetura de processamento introduzida no Capítulo 5. Observa-se que a arquitetura proposta aplica o método de exploração de grafos ao invés da exploração baseada em junções como ocorre em RDF-3X. Por este motivo os resultados reportados neste estudo não são equivalentes aos reportados em [Jiewen Huang, 2011] e em [Zeng et al., 2013], dado que em ambos os casos o *METIS-2hops* é implantado sobre um *triple-store*. Além do método de exploração em grafos, a arquitetura considerada elimina o custo envolvido no escalonamento de processos que seria produzido ao utilizar a plataforma *Hadoop*.

Outra decisão importante relacionada à implantação do *METIS-2hops* foi tomada com

relação ao modelo de armazenamento, uma vez que esta abordagem foi originalmente projetada para *triple-stores*. Com o objetivo de prover uma comparação justa com *ClusterRDF*, decidiu-se por aplicar o mesmo limiar de armazenamento para os fragmentos de *METIS-2hops*, neste caso 12kb. Para isto, inicialmente foram geradas partições usando o *METIS* e, em seguida, alocaram-se as triplas de acordo com a garantia não-direcionada de *2-hops*. No caso do tamanho da partição exceder o limiar, os arquivos são divididos em tamanhos adequados e agrupados através da atribuição de um mesmo prefixo para as chaves DHT. Observe que este procedimento de alocação é similar ao aplicado por *ClusterRDF* sobre o *Scalaris*.

O particionador METIS foi executado utilizando a otimização de vértices de alto-grau conforme introduzido por [Jiewen Huang, 2011]. Vértices de alto-grau correspondem a nós RDF que estão conectados a muitos outros nós. Neste caso, um nó é dito de alto-grau se seu grau excede em três unidades o desvio padrão sobre o grau médio de vértices em um grafo RDF. Segundo os autores, esta otimização visa evitar problemas no particionamento uma vez que grafos bem conectados são difíceis de particionar. Assim, vértices de alto-grau devem ser ignorados durante o particionamento no METIS e na alocação de *2-hops* das triplas. Após o particionamento e alocação, os vértices ignorados são adicionados às partições que contêm o maior número de nós originalmente adjacentes a estes vértices.

O mesmo método para identificar nós de alto-grau é utilizado em *Trinity.RDF*. Neste caso, um nó de alto-grau é co-alocado com nós da sua lista de adjacência. *Trinity.RDF* armazena dados como um conjunto de nós em um repositório chave-valor. Cada par mantém um identificador de um nó RDF e um conjunto de identificadores para os nós na sua lista de adjacência. O mesmo modelo de armazenamento foi implementado para avaliar o *Trinity.RDF* neste estudo.

Os sistemas utilizados para executar *ClusterRDF*, *METIS-2hops* e *Trinity.RDF* foram desenvolvidos em Java e construídos sobre um *cluster* de instâncias EC2 da Amazon, cada qual com 15 GB de memória, 4 vCPU 8 GHz Intel Xeon e 420 GB de disco. Cada instância executou um nó *Scalaris* conectado a um sistema de DHT.

6.1.2 Configurações do Experimento

O Benchmark *Berlin SPARQL Benchmark* (BSBM) [Bizer and Schultz, 2009] foi aplicado neste estudo e construído sobre um caso de uso de *e-commerce*, onde o esquema modela os relacionamentos entre produtos, suas características, produtores, vendedores e revisões de produto. O BSBM provê uma carga de trabalho com 12 consultas e um gerador de dados para a criação de bases de tamanho arbitrário baseado no número de produtos como fator de escala. Dentre as 12 consultas, 11 delas foram utilizadas por este estudo e correspondem ao conjunto de consultas que satisfazem a definição de *padrões de grafos*. A consulta remanescente foi desprezada por utilizar uma construção que não é atendida pela definição de padrões de grafos e que corresponde ao uso de variáveis para propriedades de padrões de triplas. Diferentemente de outros *benchmarks* para RDF, o BSBM fornece um conjunto de informações adequado para a aplicação da caracterização da carga de trabalho, tais como frequências de consultas, tamanho de nós e cardinalidades sobre uma estrutura RDF.

Para um tamanho específico de banco de dados e carga de trabalho fornecida pelo BSBM, foram gerados *datasets* particionados de acordo com as abordagens de *Cluster-RDF*, *METIS-2hops* e *Trinity.RDF*. A Tabela 6.1 sumariza as estatísticas dos *datasets* utilizados neste estudo. O tamanho se refere ao tamanho dos arquivos no formato *N-triple* gerado para BSBM. Neste estudo, os limiares utilizados para delimitar a replicação de dados foram configurados de forma que cada item de dados pudesse assumir o máximo de 10 réplicas. A definição deste valor habilitou a produção de réplicas pelos algoritmos *affFrag* e *affPart* na maioria dos casos em que a replicação foi aplicável. Como esperado, *ClusterRDF* e *Metis-2hop* requerem um maior espaço de armazenamento em termos de triplas replicadas (*Overhead* de Triplas). Porém, *Metis-2hop* produziu o dobro de triplas se comparado a *ClusterRDF* definido a partir do limiar de replicação $u(n) = 10$, para qualquer nó n de um grafo RDF.

Dataset	#Produtos	#Triplas	Tamanho	*Overhead* de Triplas	
				ClusterRDF	*Metis-2hops*
BSBM_1	100	40405	10.2MB	14141	27071
BSBM_2	200	75620	19.2MB	22686	44615
BSBM_3	500	191650	48.9MB	67329	120739
BSBM_4	1000	375163	96MB	105045	213842
BSBM_5	10000	3567636	922.3GB	891909	1748141
BSBM_6	100000	35300350	68.6GB	7766077	15532154
BSBM_7	300000	100399052	27GB	20079810	40159620

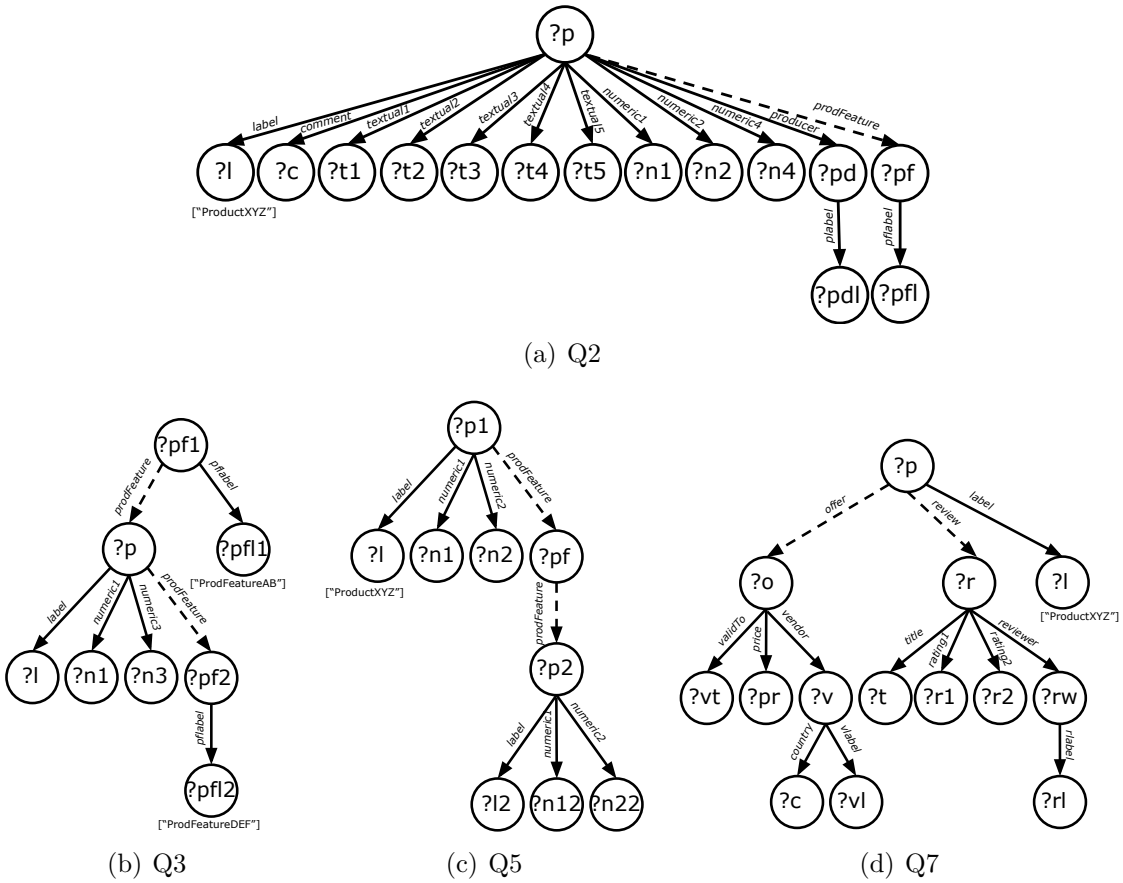
Tabela 6.1: Estatísticas de *Datasets* Utilizados na Avaliação

6.1.3 Resultados do Experimento

O objetivo dos experimentos reportados nesta seção é determinar o efeito do método de particionamento proposto sobre o desempenho do sistema, e compará-lo com as abordagens *Metis-2hops* e *Trinity.RDF*. A comparação está baseada no tempo de resposta para recuperar dados de uma consulta a partir de um repositório de dados. Os efeitos do particionamento são demonstrados através da análise de 4 consultas selecionadas. Os respectivos padrões de grafo das consultas são apresentados pela Figura 6.1.

6.1.3.1 Desempenho da Recuperação de Dados

As abordagens de particionamento foram inicialmente comparadas em um *cluster* de 8 servidores com o *dataset* BSBM_5. Os resultados são apresentados pela Figura 6.2. Os tempos obtidos em milissegundos correspondem à média de 10 execuções de cada consulta e representam o custo de recuperação de dados da consulta sobre um repositório distribuído. De acordo com o sistema de processamento proposto, cada servidor inicia uma *thread* e executa um número arbitrário de requisições locais e remotas para recuperação dos dados. As requisições remotas são identificadas como *requisições distribuídas* no decorrer do texto. Nesta composição paralela de recuperação, o número de requisições distribuídas determina os tempos de resposta reportados. Foram coletados tanto o número máximo de requisições distribuídas executadas por uma *thread*, quanto o número total de requisições distribuídas obtido pelo conjunto de *threads* executadas. A Figura 6.3 apresenta estes resultados, de forma que as colunas na base da pilha representam a quantidade máxima



(a) Q2

(b) Q3 (c) Q5 (d) Q7

Figura 6.1: Padrões de Grafo para Consultas do Experimento

de requisições distribuídas a partir de uma *thread*, enquanto a barra superior define o acumulado de requisições deste tipo desempenhadas por todas as *threads*. O número total de requisições distribuídas corresponde ao total de partições sobre as quais uma dada consulta foi segmentada. Portanto, o total de requisições distribuídas equivale à medida $\hat{P}$ que determina a qualidade do particionamento, conforme estabelece a Definição 4.1.2.

Adicionalmente, foi coletado o número total de fragmentos recuperados, e que está relacionado ao tamanho do resultado de uma consulta. A Figura 6.4 apresenta o total de fragmentos recuperados por cada abordagem a partir de cada consulta em escala logarítmica. É importante observar que, devido a fragmentação da base, uma requisição ao repositório pode encapsular um conjunto de fragmentos a ser recuperado. Este encapsulamento da requisição é possível através de um recurso fornecido pelo *Scalaris* que permite empacotar um conjunto de operações do tipo *get* e *set* para um mesmo servidor em uma

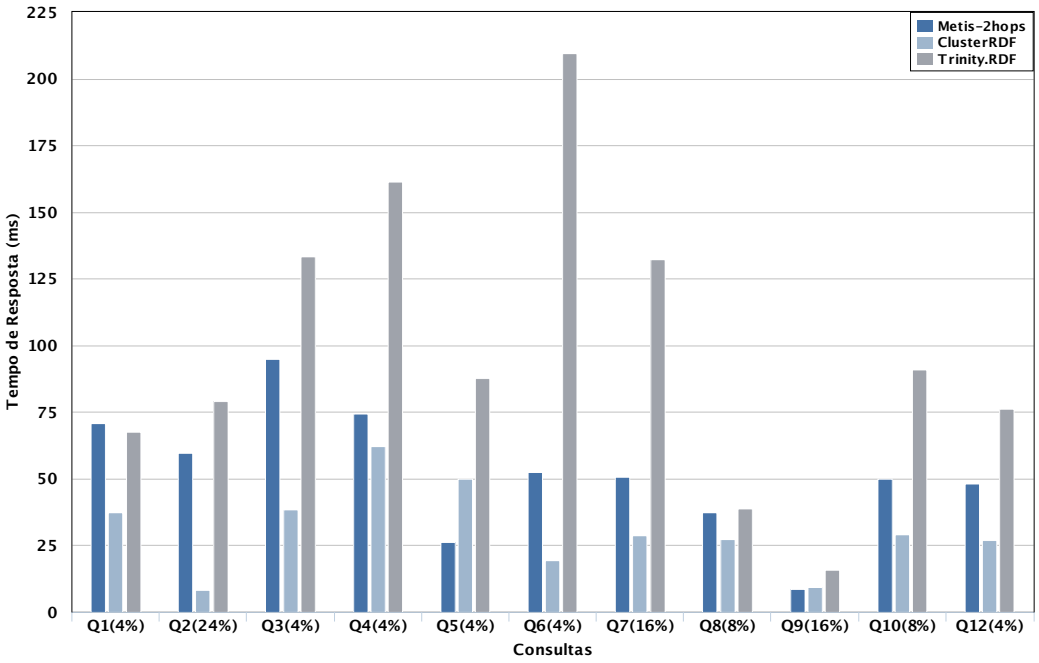


Figura 6.2: Tempo de resposta - 8 servidores e *dataset* BSBM_5

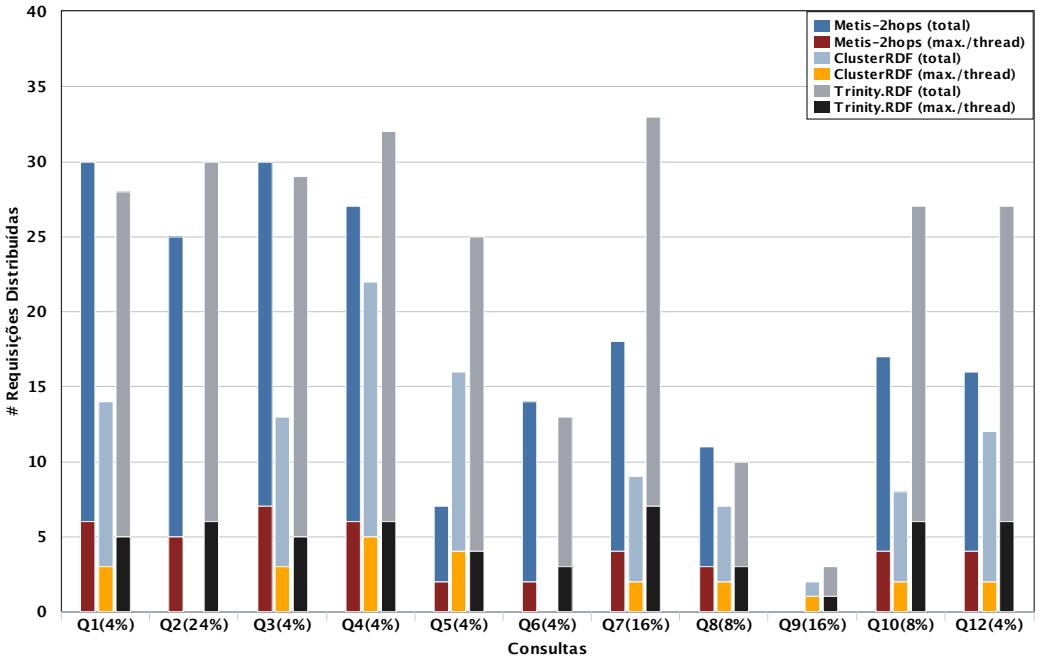


Figura 6.3: # Requisições Distribuídas - 8 servidores e *dataset* BSBM_5

única mensagem, conforme apresentado pela Seção 5.1.

Requisições entre Servidores Como esperado, existe uma correspondência direta entre o número de requisições distribuídas e o tempo de resposta. Isto é, um alto número de

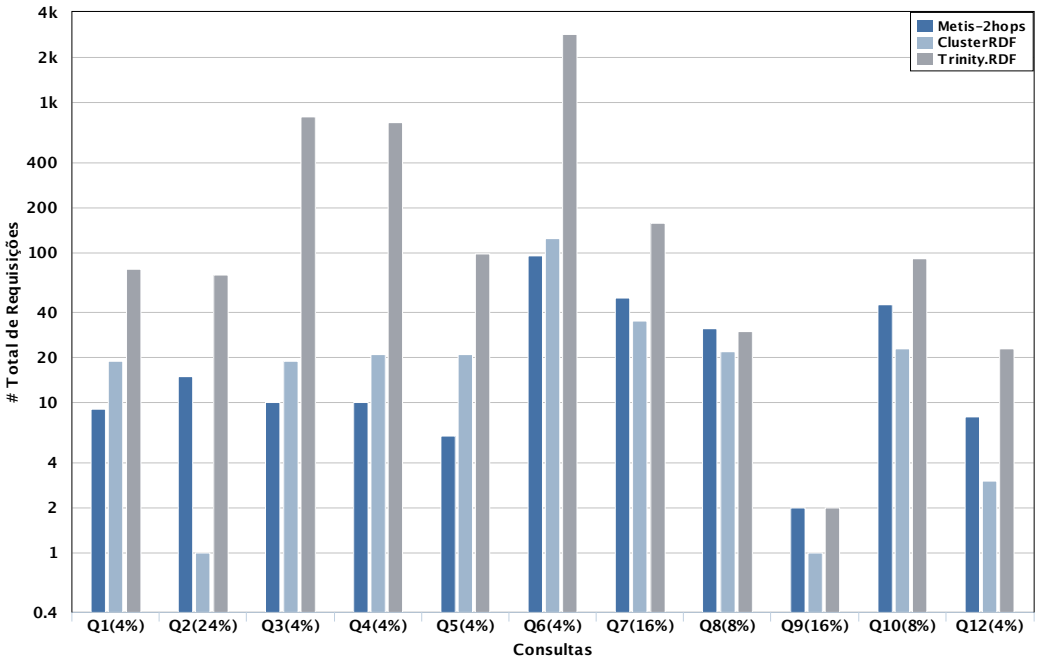


Figura 6.4: # Total de Requisições - 8 servidores e *dataset* BSBM_5

requisições distribuídas representa um alto custo para recuperar dados distribuídos entre partições distintas. Observe que uma *thread* que executa $Q1$ em *ClusterRDF* requer o acesso máximo a 4 servidores distintos, o que corresponde a um tempo de resposta de 37,27 ms. A execução da mesma consulta em *Metis-2hops* e *Trinity.RDF* quase dobra o número de requisições e tem o mesmo efeito no tempo de resposta (70,94 ms e 67,52, respectivamente).

Intuitivamente, o número de requisições entre servidores necessárias para recuperar dados de consultas mede a eficácia dos métodos de particionamento. A diferença entre os resultados dos métodos avaliados pode ser explicada pela cobertura que cada método proporciona, em termos de padrões de consulta. Pode-se verificar que uma garantia de 2-hops do *Metis-2hops* não é suficiente para cobrir todo o padrão da maioria das consultas na carga de trabalho BSBM. Por exemplo, uma garantia *2-hops* não é suficiente para recuperar os dados de *vendor* e *reviewer* do padrão de grafo de $Q7$ na Figura 6.1(d).

Observa-se que *Metis-2hops* não garante uma cobertura de *2-hops* em alguns casos para os quais é aplicada a otimização de vértices de alto-grau. Considere um caso particular

para a consulta $Q2$ na Figura 6.1(a), onde uma instância de produto p a ser consultada por $?p$ é identificada como um vértice de alto-grau. Observe que, se p é removida do grafo RDF, também se removem as associações de nós literais relacionados com p. Como consequência, esses nós podem ser alocados entre partições arbitrárias. Embora p seja adicionada à partição que contém o maior número de nodos relacionados, no passo final, não é possível evitar a distribuição completa dos nós literais. Isso explica o número elevado de requisições entre servidores para $Q2$ no *Metis-2hops*.

Trinity.RDF fornece uma cobertura simples na maioria dos casos devido à estratégia de particionamento baseada apenas em nós de alto-grau. Nós de alto-grau são co-alocados com nós de sua lista de adjacência. Para os demais tipos de nó, uma distribuição aleatória é aplicada. Assim, a menos que os nós a serem consultados sejam nós de alto-grau, não há qualquer garantia que as partições de *Trinity-RDF* possam cobrir os padrões de consulta. Isso explica porque o *Trinity.RDF* apresenta os piores resultados dentre as três soluções.

ClusterRDF fornece uma cobertura completa para as consultas $Q2$ e $Q6$, uma vez que as requisições são atendidas pelas partições locais das *threads*. Este fato é reportado pela ausência de requisições distribuídas na Figura 6.3 para *ClusterRDF* nestas consultas. Para as demais consultas, *ClusterRDF* não consegue evitar requisições distribuídas. No entanto, ele reduz o número de servidores acessados quando comparado com as outras duas alternativas. Os resultados apresentados na Figura 6.2 mostram que *ClusterRDF* supera *Metis-2hops* e *Trinity.RDF* para a maioria das consultas, com exceção de *Q5* e *Q9* em *Metis-2hops*. Isto porque *ClusterRDF* atribui dados para os agrupamentos de acordo com o padrão de acesso das consultas mais frequentes da carga de trabalho. Observe que, o *mix* de consultas BSBM é dado em termos de percentuais ao longo dos rótulos da consulta na Figura 6.2. Dado que *ClusterRDF* favorece a cobertura das consultas mais frequentes, o desempenho de consultas menos frequentes, como *Q5* pode ser comprometida.

Observe que se o número de requisições distribuídas apresentado pela Figura 6.3 equivale a medida $\hat{P}$ da qualidade do particionamento, o somatório da Equação 4.2 produz a quantidade média de requisições distribuídas para o *mix* de consultas dada pelo acumu-

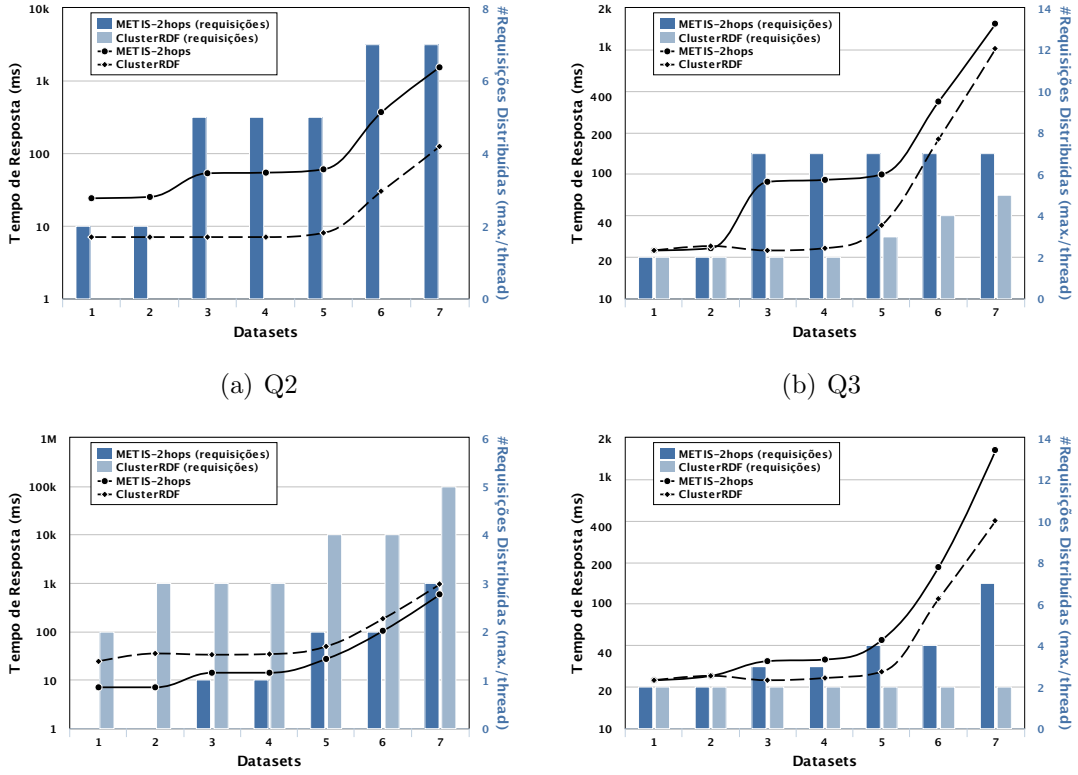
lado das requisições sobre a porcentagem da frequência das consultas. Isto é, *ClusterRDF* executará em média 6,04 requisições distribuídas, enquanto *Metis-2hops* e *Trinity.RDF* executarão 15,84 e 22,08 requisições, respectivamente. Estes resultados evidenciam a qualidade do particionamento gerado por *ClusterRDF* e a cobertura efetiva para as consultas mais frequentes da carga de trabalho.

Total de requisições O tamanho dos resultados das consultas é reportado pela quantidade de requisições totais na Figura 6.4. Esta medida representa a quantidade total de fragmentos (unidades de armazenamento) recuperada. Como apresentado, o *Scalaris* fornece uma funcionalidade para empacotar conjuntos de requisições destinadas a um mesmo servidor em uma única mensagem para minimizar o custo da troca de mensagens. Neste estudo, observou-se que o custo desses pacotes de mensagem pode ser ignorado quando a quantidade de requisições é de até 10 requisições por servidor. Este não é o caso de *Q6*, que recupera 125 fragmentos de *ClusterRDF* alocados em apenas um servidor. Neste caso, apesar do acesso local, o custo para recuperar os 125 fragmentos do repositório torna o tempo de execução de $Q6$ superior aos tempos reportados para a execução de $Q9$ que envolve 2 acessos distribuídos. Em *Trinity.RDF*, esse valor é ainda maior para todas as consultas. Este fato está relacionado ao modelo de armazenamento de granularidade fina aplicado por *Trinity.RDF*, em que cada unidade de armazenamento (par chave-valor) mantém apenas um nó. Por exemplo, apesar de *Metis-2hops* e *Trinity.RDF* apresentarem uma quantidade de requisições distribuídas equivalente, o tempo de resposta de *Trinity.RDF* é expressivamente superior. Esta diferença pode ser explicada pelos totais de fragmentos acessados. Enquanto, *Metis-2hops* acessa 95 fragmentos, *Trinity.RDF* acessa 2000 fragmentos aproximadamente.

Além das requisições distribuídas e requisições no total, o tempo de resposta é também afetado pela estrutura da consulta, o tamanho do conjunto de dados e o número de servidores no sistema distribuído. Os efeitos destas variantes são discutidos a seguir em relação às abordagens mais competitivas: *ClusterRDF* e *Metis-2hops*.

6.1.3.2 Escalabilidade

Para avaliar a escalabilidade de *ClusterRDF* e *Metis-2hops*, foram realizados experimentos variando-se o tamanho dos *datasets* e a quantidade de servidores.



(a) Q2 (b) Q3

(c) Q5 (d) Q7

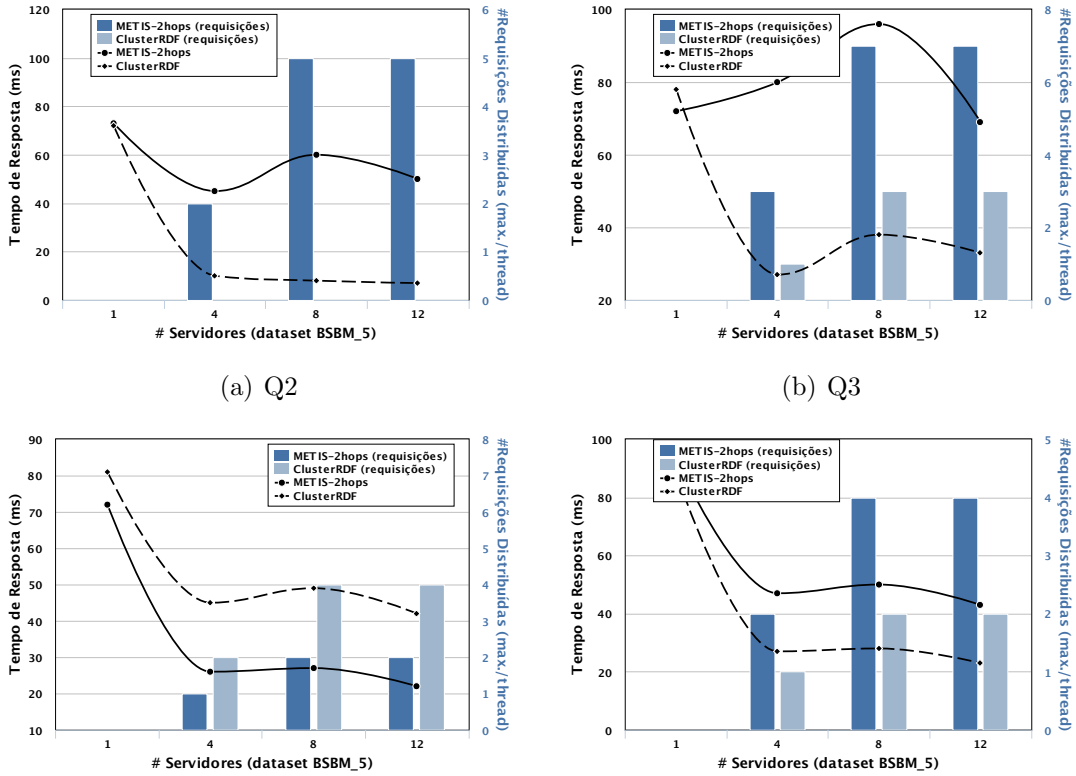
Figura 6.5: Escalabilidade de Dados

Escalabilidade de dados: Para avaliar a escalabilidade das soluções quanto a variações no tamanho dos *datasets*, um *cluster* de 8 servidores e 7 *datasets* (BSBM_1 a BSBM_7) de tamanhos crescentes foram utilizados. Os resultados são apresentados na Figura 6.5. A escala logarítmica à esquerda reporta os tempos de resposta, enquanto a escala à direita apresenta o número máximo de requisições distribuídas desempenhado por cada *thread*. Em geral, os resultados destas consultas aumentam à medida que o tamanho do conjunto de dados também aumenta. Este aumento no tamanho leva a um maior número de requisições distribuídas na maioria dos casos. Isto pode ser explicado pelo aumento dos

graus de nós RDF que eleva o número de fragmentos e, consequentemente, a necessidade do repositório efetuar o balanceamento da carga entre os servidores. No entanto, isso só acontece quando todo o conjunto de itens de dados da consulta não está agrupado. *ClusterRDF* mantém o mesmo número de requisições distribuídas na consulta *Q7* e a requisição local em *Q2* para todos os tamanhos. Por exemplo, em *Q2*, todos os dados necessários estão agrupados, e o aumento do conjunto de dados não é suficiente para sobrecarregar a capacidade do servidor. É importante lembrar que uma outra razão que explica o aumento de requisições distribuídas para $Q2$, $Q3$ e $Q7$ em *Metis-2hops* está relacionada com o problema da otimização nos vértices de alto-grau.

Metis-2hops supera *ClusterRDF* em $Q5$. Esta consulta representa um padrão de consulta recursiva entre *ProdFeature* e *Product*, cujas partições produzidas por *ClusterRDF* não dão cobertura. Apesar de uma garantia de *2-hops* poder representar tal associação, não é suficiente para cobrir o padrão completo de $Q5$. Outra consulta recursiva entre *ProdFeature* e *Product* é apresentada em $Q3$ na Figura 6.1(b). Neste caso, um resultado semelhante é relatado por *ClusterRDF* e *Metis-2hops* nos *datasets BSBM_1* e *BSBM_2*. Em *ClusterRDF*, um número reduzido de nós de produtos está associado a um *ProdFeature* com *label* "*ProdFeatureABC*". Este fato reduz o número de requisições distribuídas para explorar a associação recursiva. Em *BSBM_3*, *Metis-2hops* produz um número maior de requisições por causa da otimização para os nós de alto-grau.

O exemplo introduzido pela Figura 4.8 no Capítulo 4 apresenta alguns elementos do particionamento efetuado por *ClusterRDF* sobre o esquema do BSBM. Assim como no exemplo, *ClusterRDF* deu preferência à co-alocação de nós da classe *Product* com nós de *Offer* associados, ao invés de estabelecer a co-alocação através do *link* recursivo entre *Product* e *ProdFeature*. Assim, embora esta estratégia de alocação não corresponda a mais adequada para as consultas $Q3$ e $Q5$, ela favorece a consulta $Q7$ pela cobertura do *link* entre *Product* e *Offer*. Observe que $Q7$ apresenta uma frequência superior (16%) às frequências de $Q3$(4%) e $Q5$(4%), o que justifica a preferência de alocação efetuada.



(a) Q2 (b) Q3

(c) Q5 (d) Q7

Figura 6.6: Escalabilidade de Servidores

Escalabilidade de servidores: Para avaliar a escalabilidade das soluções quanto a variações na quantidade de servidores, optou-se por utilizar o *dataset* BSB_5. Os resultados são apresentados pela Figura 6.6. Em geral, o aumento do número de servidores proporciona os benefícios do processamento paralelo, reduzindo a carga dos servidores e os tempos de resposta. No entanto, este aumento também pode levar a uma maior distribuição de dados entre os servidores quando os itens de dados da consulta não estão definidos para serem co-alocados. O pior resultado relacionado a este efeito é observado em $Q3$ em um *cluster* de 8 servidores para *METIS-2hops*, onde cada *thread* necessita acessar todos os servidores. Acredita-se que o elevado número de requisições que estão sendo executadas por cada *thread* em paralelo eleva a competição por recursos e afeta o desempenho do sistema. Note que o efeito do processamento paralelo somente consegue reduzir o tempo de resposta quando a capacidade do sistema é aumentada para 12 servidores e o número de requisições distribuídas permanece estável.

Os resultados reportados por este estudo demonstram que a solução de particionamento proposta pode melhorar o desempenho da recuperação de dados RDF, comparado a abordagens relacionadas. Os experimentos mostram que *ClusterRDF* melhora o tempo de resposta referente às consultas mais frequentes em até 87%, comparado a *METIS-2hops*. Além disto, as recuperações sobre *ClusterRDF* podem executar até 10 vezes mais rápido do que sobre o particionamento em *hash* efetuado por *Trinity.RDF*. O estudo demonstra ainda que o uso de heurísticas baseadas em informações da carga de trabalho produz particionamentos que oferecem uma maior coberta aos padrões de consultas, se comparado a abordagens como *Metis-2hops* e *Trinity.RDF* que baseiam-se apenas na composição de grafos RDF.

A seção seguinte apresenta um estudo que compara abordagens de fragmentação de dados XML baseadas em heurísticas sobre grafos de afinidade, assim como proposto por este trabalho. Embora a alocação de dados não é contemplada por estas soluções, a comparação a ser apresentada evidencia o efeito positivo da estratégia de formação dos fragmentos do algoritmo *affFrag* na recuperação de dados XML sobre a mesma arquitetura de processamento adotada por *ClusterRDF*.

6.2 Avaliação da Fragmentação sobre bases de dados XML

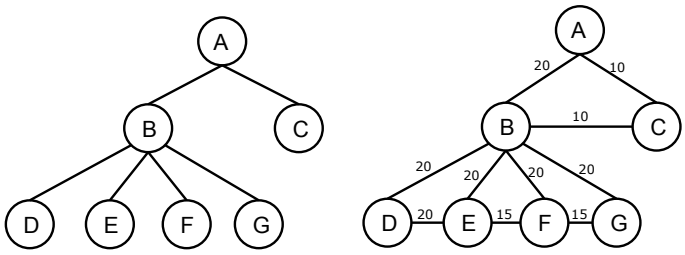
Este estudo foi realizado para determinar os efeitos da abordagem de fragmentação por afinidades proposta na recuperação de dados de consultas sobre repositórios distribuídos. Para tanto, esquemas de fragmentação produzidos pelo algoritmo *affFrag* foram comparados com abordagens de fragmentação também baseadas em heurísticas de afinidades. Duas soluções foram selecionadas para esta avaliação: o algoritmo *MakePartition* proposto em [Navathe and Ra, 1989], e uma solução para fragmentação XML denominada *XS Partition* [Bordawekar and Shmueli, 2008]. Para que fosse possível a comparação entre as 3 abordagens, adotou-se o modelo XML como modelo alvo deste estudo.

A motivação para a comparação com o algoritmo *MakePartition* é devida ao fato

de que este algoritmo introduziu a ideia de definir a fragmentação de banco de dados através de grafos de afinidade. Neste algoritmo, a fragmentação vertical de um banco de dados relacional é definida a partir de ciclos de afinidade detectados em um grafo de afinidades. Embora a intuição de *MakePartition* seja similar à adotada pelo algoritmo de fragmentação proposto por este trabalho, eles diferem na formação de fragmentos. Ciclos de afinidade em *MakePartition* relacionam itens de dados relacionados por valores de afinidade próximos, enquanto em *affFrag* é considerada a relação do conjunto de nós fortemente relacionados a um nó juntamente com um limiar de armazenamento para tentar reunir a maior quantidade possível de itens relacionados. Neste experimento, optou-se por utilizar o algoritmo *MakePartition* para definir a fragmentação de esquemas XML, definindo sub-árvores do esquema como *templates* de fragmentação. Apesar de *MakePartition* ter sido proposto para o modelo relacional, é possível aplicá-lo sobre outros modelos devido ao seu raciocínio estar baseado em um grafo de afinidades.

A comparação com o método *XS Partition* [Bordawekar and Shmueli, 2008] se deve à utilização de heurísticas baseadas na afinidade entre elementos XML. Esta abordagem considera afinidades sobre arestas pai-filho e arestas que relacionam nós irmãos sobre a estrutura de documentos XML. Esta composição é definida pelos autores como *sibling graphs*. Assim como ocorre em *affFrag*, o *XS Partition* assume tamanhos para seus nós e um limiar para a formação de fragmentos. Porém, uma análise alternativa das afinidades é aplicada neste trabalho. A árvore XML é analisada de maneira *bottom-up* para formar fragmentos a partir de conjuntos de arestas entre nós irmãos e seus respectivos nós pai. Considere um exemplo em que pretende-se gerar fragmentos a partir da árvore XML e seu *sibling graph* nas Figuras 6.7(a) e 6.7(b), respectivamente. O algoritmo inicia pela formação do conjunto nós-irmão e seu pai envolvendo os nós D, E e B, e adiciona-os ao fragmento. No passo seguinte, o algoritmo estabelece que o fragmento só poderá ser expandido através de arestas entre irmãos ou por arestas filho-pai. No caso deste exemplo, as possibilidades de extensão são dadas pelas arestas entre E e F, ou B e A. Como a afinidade entre $B - A$ é superior a de $E - F$, adiciona-se a aresta $B - A$ e assim

sucessivamente. Apesar da aresta $B - F$ apresentar uma afinidade superior a $B - A$, ela não é tratada como uma possibilidade de extensão. O algoritmo assume que devido à baixa afinidade dos irmãos $E - F$, a aresta $B - E$ é acessada de forma independente de seus irmãos. Isto também impede que o nó G seja incluído no fragmento. A cada passo, o algoritmo avalia o peso dos nós com relação a um limiar que define o tamanho máximo para os fragmentos. Apesar do critério baseado em um limiar para os fragmentos ser o mesmo, algumas relações de afinidade são desconsideradas como apresentado no exemplo. O objetivo desta avaliação é identificar o impacto causado pela disposição de dados fortemente relacionados em fragmentos diferentes. Maiores detalhes sobre as abordagens comparadas são descritos pelo Capítulo 3.



(a) Árvore XML (b) *Sibling graph*

Figura 6.7: Exemplo de Aplicação do *XS Partition*

6.2.1 Configurações do Experimento

Para efetuar este estudo experimental, utilizou-se o *benchmark XBench*[2]. *XBench* constitui uma família de *benchmarks* para XML, dos quais escolheu-se um específico para aplicações que envolvem múltiplos documentos. O *benchmark* em questão provê um esquema XML que modela parte da biblioteca digital da *Springer*, uma carga de trabalho com 19 consultas e um gerador de base de dados que pode ser configurado para gerar um conjunto de documentos contendo um número de artigos especificado pelo usuário. Dentre as 19 consultas definidas pelo *benchmark*, 11 foram aplicadas pelo experimento dado que as 8 consultas restantes (3, 4, 5, 7, 8, 10, 11 e 18) visam a avaliação de proces-

[2]http://se.uwaterloo.ca/ ddbms/projects/xbench/

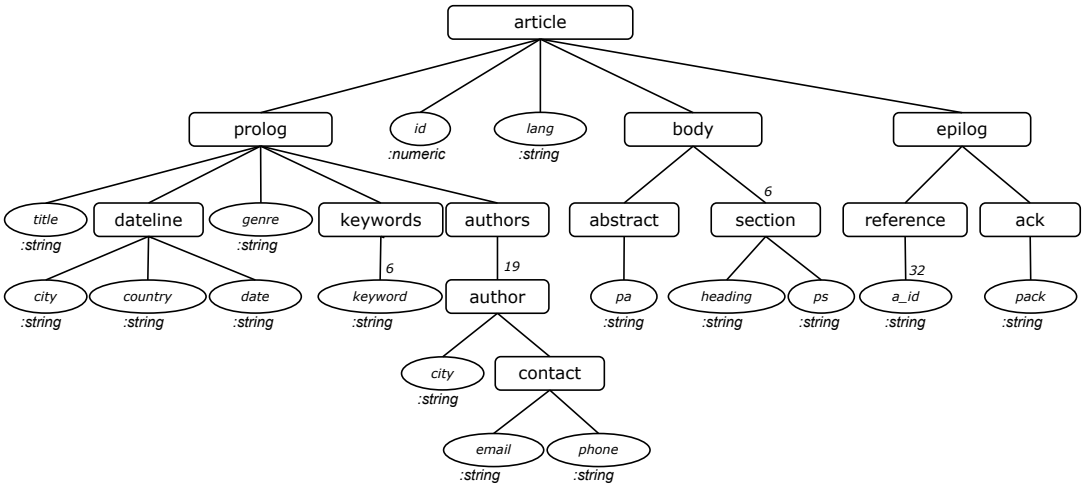


Figura 6.8: Esquema XML XBench

sadores *XQuery* que não é o foco deste estudo. A estrutura XML provida é apresentada pela Figura 6.8. Diferente do modelo RDF, as arestas da estrutura XML não são identificadas por propriedades e as ocorrências de sub-elementos são dadas apenas no sentido de aninhamento devido a sua composição em árvore.

Dada a carga de trabalho e o tamanho dos nós provido pelo *XBench*, os *templates* de fragmentação foram definidos de acordo com os algoritmos *xAFFrag*, *MakePartition* e *XS*. Um limiar de 12kb foi assumido para o tamanho de fragmentos produzidos a partir da fragmentação proposta por este trabalho. A partir de cada esquema de fragmentação, um banco de dados é gerado para avaliar o desempenho da recuperação de dados de cada uma das consultas para cada abordagem de fragmentação. Para este experimento foi utilizada uma versão simplificada do sistema *ClusterRDF* sobre o repositório *Scalaris* para o processamento de requisições geradas por consultas *XQuery*. Além da exploração ocorrer sobre árvores XML, a simplificação também está relacionada ao processamento das recuperações. Ao invés de iniciar um processo de recuperação paralelo sobre os servidores, optou-se por estabelecer o processamento apenas a partir do nó do sistema ao qual é submetido à consulta. As razões para esta decisão envolvem dois principais fatores. O primeiro deles diz respeito à distribuição aleatória dos fragmentos sobre a DHT, uma vez que a alocação não foi considerada neste estudo. Portanto, o conceito das partições

mantidas por cada servidor é desconsiderado. Em segundo lugar, o conjunto de consultas definido pelo *XBench* recupera uma porção pequena de dados, e que portanto não justificam uma execução paralela.

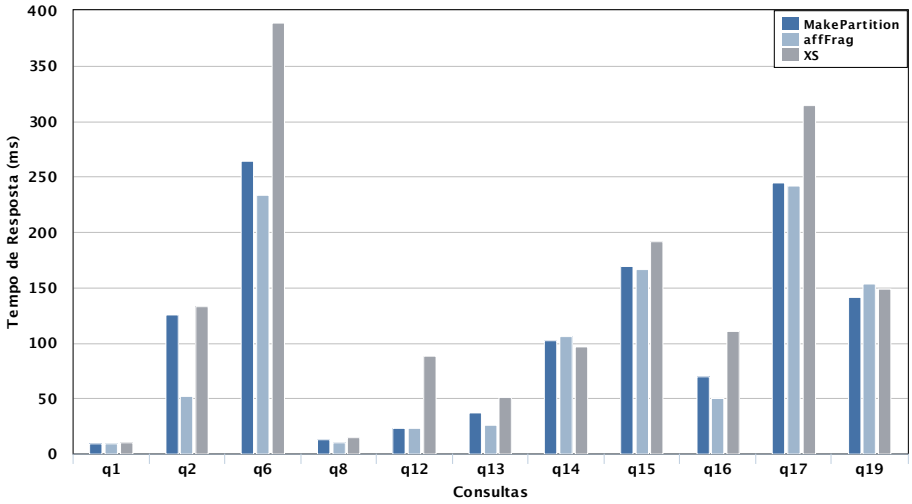
Os experimentos foram executados em instâncias *EC2* da *Amazon*. Para este fim, 8 instâncias do tipo *large* foram alocadas, cada qual com 7.5 GB de memória e 4 unidades de processamento sobre uma plataforma de 64-bits. Cada uma destas instâncias executou um nó *Scalaris* conforme disposto pela arquitetura descrita anteriormente na Figura 5.4.

6.2.2 Resultados Experimentais

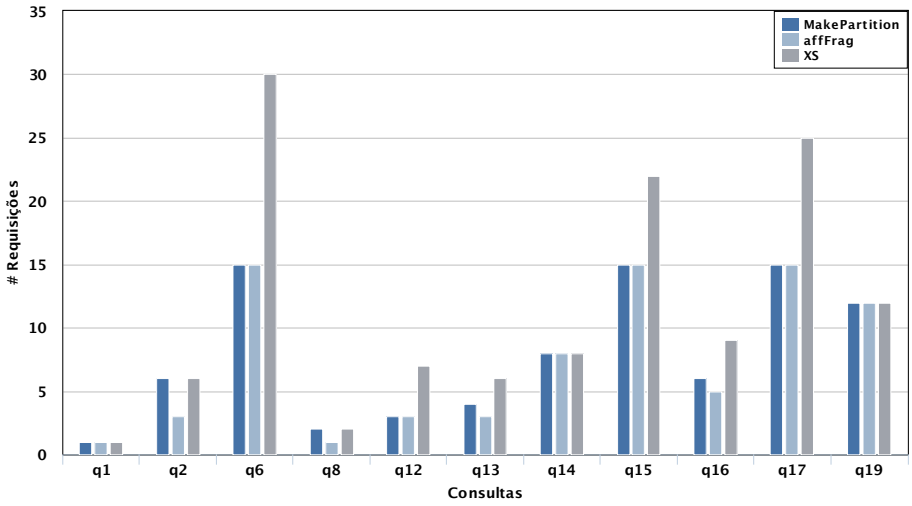
Os resultados reportados por esta seção mostram o efeito da abordagem proposta sobre o desempenho do sistema, e compara-os com os resultados obtidos pela aplicação dos algoritmos *MakePartition* e *XS Partition*. A comparação é baseada em 2 métricas: o tempo de resposta da recuperação de dados em milissegundos e o número de requisições ao repositório efetuadas por cada recuperação.

Para mensurar o tempo de resposta, um conjunto de 26 documentos XML, totalizando 100MB, foi gerado e armazenado sobre 8 nós *Scalaris*. O tempo de resposta médio gerado por 10 execuções da recuperação de dados de cada consulta é apresentado pela Figura 6.9(a). O número de requisições efetuadas ao repositório foi igualmente coletado, o qual corresponde ao total de fragmentos requeridos para execução da recuperação de dados de uma consulta do *benchmark*. A Figura 6.9(b) apresenta esta quantidade para 10 execuções. É importante destacar novamente a ausência de uma estratégia de alocação e a consequente distribuição aleatória dos fragmentos no repositório. Portanto, se uma consulta necessitar recuperar dados de mais de um fragmento, é possível que estes fragmentos estejam completamente distribuídos pelo repositório. Neste caso, como o foco da avaliação está em avaliar a formação de unidades de armazenamento representadas pelos fragmentos, optou-se por avaliar os tempos de resposta obtidos a partir da quantidade de requisições necessárias para efetuar a recuperação dos dados de uma consulta. Desta forma, a quantidade de requisições efetuadas equivale à quantidade de fragmentos

recuperados. Conforme esperado, os resultados reportados pelos 2 gráficos das Figuras 6.9(a) e 6.9(b) evidenciam uma correspondência direta entre o número de requisições ao repositório e o tempo de resposta. Observe que para executar q_6 sobre uma base de dados gerada a partir de *affFrag*, 15 fragmentos no repositório são acessados, que corresponde a uma execução em 233.47 ms. A execução da mesma consulta sobre um banco de dados gerado a partir de *XS* dobra o número de requisições e apresenta o mesmo efeito no tempo de resposta (588.63 ms).



(a) Desempenho - Recuperação de Dados



(b) Número de Requisições

Figura 6.9: Recuperação de Dados de Consultas XBench - 8 servidores e BD de 100MB

A diferença entre os resultados reportados por *affFrag* e *MakePartition* pode ser expli-

cada pelas diferenças nos critérios de geração de fragmentos. Enquanto *affFrag* baseia-se na formação do conjunto de nós fortemente relacionados e em um limiar de armazenamento, *MakePartition* cria fragmentos baseado em ciclos no grafo que relacionam itens com valores de afinidade próximos. Esta abordagem pode levar à separação de dados relacionados entre fragmentos distintos, o que aconteceu neste experimento com relação a alguns *templates* de fragmentação e consultas. Assim, com um maior número de fragmentos que relacionam estes dados, o número de requisições para processar uma consulta tende a ser maior, o que reflete nos resultados das Figuras 6.9(a) e 6.9(b). Da mesma forma que os ciclos de afinidades podem gerar fragmentos pequenos, eles também podem gerar fragmentos que detêm uma grande quantidade de dados visto que o tamanho destes não está sendo controlado. Apesar de esta situação não ter sido identificada neste experimento, grandes fragmentos aumentam o tamanho das mensagens transferidas e, por consequência, aumentam a latência de requisições ao repositório de dados.

O número de fragmentos gerados por *XS* e por *affFrag* é similar. No entanto, *XS* apresenta o pior desempenho dentre as 3 soluções. Neste caso, a formação de fragmentos é determinada por uma análise de afinidades que ignora algumas relações de afinidade como exemplificado pela Figura 6.7. Uma vez que *XS* não considera a afinidade entre todos os nós envolvidos em uma consulta, é possível que nós fortemente relacionados sejam colocados em fragmentos distintos. Consequentemente, é necessário um número de requisições maior para recuperá-los.

Os experimentos mostram que a abordagem proposta para formação de fragmentos baseada em afinidades é efetiva para melhorar o desempenho da recuperação de dados sobre um repositório de dados distribuído, comparado a abordagens correlatas. Os resultados evidenciam que o algoritmo *affFrag* pode melhorar o desempenho do sistema em 22%, se comparado à estratégia tradicional de *MakePartition*, e 55%, se comparado ao algoritmo *XS*. O algoritmo *affFrag* é capaz de relacionar itens de dados de uma estrutura de BD, gerando um esquema de fragmentação que por sua vez determina como o BD deverá ser fragmentado. Os efeitos da fragmentação atrelada à alocação de dados são avaliados

pela seção anterior e demonstram a efetividade da solução completa apresentada por este trabalho.

Um importante diferencial da solução proposta diz respeito a sua adequação a sistemas em nuvem. Como visto no Capítulo 2, um BD em nuvem está sujeito a sua volatilidade, em especial à admissão de novos dados a rede de forma assíncrona. Em geral, soluções de particionamento são propostas sobre grafos de dados, e podem exigir que um novo particionamento seja realizado à medida que novos dados são introduzidos ao grafo. A solução proposta permite definir previamente a estratégia de particionamento baseando-se em uma sumarização do grafo de dados e de uma carga de trabalho prevista. À medida que novos dados são introduzidos, novos fragmentos são gerados a partir de *templates* e co-alocados através de *links* de alocação pré-definidos. Além deste diferencial, a estratégia aqui apresentada não requer a avaliação de todo o grafo de dados. Como visto, ambientes em nuvem estão sujeitos a um grande volume de dados, o que pode levar a exaustão dos processos de particionamento estabelecidos por abordagens correlatas.

CAPÍTULO 7

CONCLUSÃO

Esta tese apresentou uma abordagem para o particionamento de dados baseada em informações da carga de trabalho e restrições de armazenamento e replicação de dados. O problema do particionamento de dados foi dividido em dois sub-problemas: fragmentação e alocação. Na fragmentação, a solução proposta analisa as informações da carga de trabalho disponíveis para identificar itens de dados que devem ser mantidos em conjunto e os atribui a um mesmo fragmento de acordo com um limiar de armazenamento. Na alocação, são consideradas as afinidades que relacionam dados em fragmentos distintos para definir como os fragmentos devem ser agrupados. A solução está focada em prover um particionamento adequado que cobre o padrão de acesso das consultas mais frequentes de uma carga de trabalho. O método utiliza a replicação de dados para aumentar a cobertura de consultas quando necessário. Porém, a quantidade de réplicas para cada item de dado é controlada por um limiar fornecido como entrada ao processo. Como resultado, é possível maximizar a execução paralela de consultas capazes de executar localmente e minimizar o número de consultas envolvendo dados distribuídos por diversos servidores de um repositório em nuvem. O método proposto pode ser aplicado por diversos modelos de dados, devido à flexibilidade do modelo RDF sobre o qual a solução foi estabelecida. Este trabalho contribui para o contexto de bancos de dados largamente distribuídos, nos quais os custos de comunicação devem ser reduzidos para prover um serviço escalável.

Os estudos experimentais realizados demonstram que a solução proposta para o particionamento de dados é capaz de melhorar o desempenho da recuperação de dados de consultas. Os resultados obtidos indicam que o método proposto melhora em até 87% o tempo de resposta referente às consultas mais frequentes de uma carga de trabalho, se comparado a métodos que são baseados em heurísticas sobre a composição estrutural de

grafos RDF. Além disto, a estratégia de formação de fragmentos se mostrou até 55% mais efetiva sobre o desempenho de consultas XML do que outras soluções igualmente baseadas em heurísticas de afinidade sobre grafos.

Um importante diferencial da solução está em definir a estratégia de particionamento a partir da estrutura de um banco de dados. Esta metodologia confere duas vantagens fundamentais para o contexto de repositórios em nuvem. Em primeiro lugar, a visão sumarizada do conjunto de dados provida pela estrutura do BD descarta a necessidade de se avaliar todo o conjunto de dados. Diante do grande volume de dados que pode ser assumido por sistemas em nuvem, esta abordagem se mostra adequada para evitar a exaustão do processo de particionamento. Em segundo lugar, é possível definir a estratégia de particionamento antes da formação completa do conjunto de dados. Este procedimento adere à formação progressiva de BDs em nuvem, dada pela admissão de novos dados ao sistema de forma assíncrona. Desta forma, os *templates* de fragmentação e alocação podem ser aplicados sobre novas porções de dados, ao invés de acionar um procedimento de re-particionamento sempre que se caracterizar um novo estado de formação do BD.

Os trabalhos futuros incluem uma série de problemas considerados relacionados à proposta desta tese, e que são agrupados pelos seguintes itens:

1. **Replicação de Dados:** A replicação de dados foi utilizada na solução proposta para fornecer uma maior cobertura de consultas. Adicionalmente, pretende-se aplicá-la para os seguintes contextos e problemas:

   — *Distribuição não-uniforme da carga de trabalho*: O acesso intenso a determinadas porções do BD é um problema característico de diversos tipos de aplicação. Além disto, o conjunto de dados mais frequentemente acessado pode se alterar periodicamente. Neste contexto, pretende-se utilizar a replicação como uma estratégia de balanceamento da carga sobre estes dados.

   — *Banco de Dados multi-inquilino*: O compartilhamento de recursos de gerenciamento de dados por diversas aplicações acarreta na execução de diferentes

cargas de trabalho sobre o mesmo SGBD. A replicação é considerada neste contexto para permitir diferentes esquemas de particionamento e acomodar cada carga de trabalho.

— *Re-particionamento*: Alterações na carga de trabalho podem levar à definição de um novo esquema de particionamento e a consequente migração de dados. Para que não seja necessário interromper o funcionamento das aplicações, pode ser necessário manter diferentes versões do BD referentes ao particionamento vigente e ao novo particionamento envolvido no processo.

2. **Processamento de Consultas:** Embora o sistema *ClusterRDF* tenha sido implementado para dar suporte a uma arquitetura de processamento de recuperação de dados, se faz necessário o desenvolvimento de um *engine* para o processamento de consultas sobre repositórios RDF particionados em nuvem. Neste contexto, destaca-se o desenvolvimento dos itens seguintes:

   — *Metadados*: A gestão de metadados deve ser estabelecida de forma escalável, bem como prover estruturas para a definição de índices e endereçamento de fragmentos alocados no repositório de dados distribuído.

   — *Otimizações*: Esquemas de particionamento podem exigir a rescrita de consultas visando um melhor desempenho. Pretende-se explorar este problema em conjunto com otimizações aplicadas à exploração de grafos, conforme introduzido por *ClusterRDF*.

3. **Controle da Localidade de Fragmentos:** Como definido por *ClusterRDF*, grupos de fragmentos foram co-alocados através da ordem lexicográfica de chaves mantida pelo repositório chave-valor utilizado. Apesar disto, não há garantias da co-alocação de pares com chaves que assumem valores próximos na ordem. Diante deste problema, considera-se a composição de um repositório que permita tal controle sobre um sistema de arquivos distribuído.

PUBLICAÇÕES REALIZADAS NO DOUTORADO

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

1. R. R. M. Penteado, R. Schroeder, D. Hoss, J. Nande, R. M. Maeda, W. O. Couto and C. S. Hara. Um Estudo sobre Bancos de Dados em Grafos Nativos. *Escola Regional de Banco de Dados (ERBD)*, Abr. 2014.

2. R. Schroeder e C. S. Hara. Clustering RDF Data by Affinity Relations. *International Conference on Management of Data (SIGMOD)*, submetido em Dez. 2013.

3. R. Schroeder, R. R. M. Penteado and C. S. Hara. Partitioning RDF Exploiting Workload Information. *International Conference on World Wide Web Companion (WWW)*, pp. 213-214, Mai. 2013.

4. R. Schroeder and C. S. Hara. Towards Full-fledged XML Fragmentation for Transactional Distributed Databases. *Workshop de Teses e Dissertações em Banco de Dados*, Out. 2012.

5. R. Schroeder and C. S. Hara. Affinity-based XML Fragmentation. *International Workshop on the Web and Databases (WebDB: co-located with SIGMOD)*, pp. 61-66, Mai. 2012

6. D. E. M. Arnaut, R. Schroeder and C. S. Hara. Phoenix: A Relational Storage Component for the Cloud. *International Conference on Cloud Computing (IEEE Cloud)*, pp. 684-691, Jul. 2011

REFERÊNCIAS

[Abadi et al., 2009] Abadi, D. J., Marcus, A., Madden, S. R., and Hollenbach, K. (2009). SW-Store: A Vertically Partitioned DBMS for Semantic Web Data Management. *The VLDB Journal*, 18(2):385–406.

[Abiteboul et al., 2008] Abiteboul, S., Manolescu, I., Polyzotis, N., Preda, N., and Sun, C. (2008). XML processing in DHT networks. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 606–615.

[Abou-Rjeili and Karypis, 2006] Abou-Rjeili, A. and Karypis, G. (2006). Multilevel algorithms for partitioning power-law graphs. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, pages 124–124.

[Agrawal et al., 2013] Agrawal, D., Das, S., and Abbadi, E. A. (2013). *Data Management in the Cloud: Challenges and Opportunities*. Morgan & Claypool.

[Agrawal et al., 2010] Agrawal, D., El Abbadi, A., Antony, S., and Das, S. (2010). Data management challenges in cloud computing infrastructures. In *Proceedings of the 6th International Conference on Databases in Networked Information Systems*, pages 1–10.

[Agrawal et al., 2009] Agrawal, R., Ailamaki, A., Bernstein, P. A., Brewer, E. A., Carey, M. J., Chaudhuri, S., Doan, A., Florescu, D., Franklin, M. J., Garcia-Molina, H., Gehrke, J., Gruenwald, L., Haas, L. M., Halevy, A. Y., Hellerstein, J. M., Ioannidis, Y. E., Korth, H. F., Kossmann, D., Madden, S., Magoulas, R., Ooi, B. C., O'Reilly, T., Ramakrishnan, R., Sarawagi, S., Stonebraker, M., Szalay, A. S., and Weikum, G. (2009). The Claremont Report on Database Research. *Communications of the ACM*, 52(6):56–65.

[Agrawal et al., 2004] Agrawal, S., Narasayya, V., and Yang, B. (2004). Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings*

of the ACM SIGMOD International Conference on Management of Data, pages 359–370.

[Amazon, 2014] Amazon (2014). Amazon Web Services. http://aws.amazon.com/. Acessado em Maio de 2014.

[Andrade et al., 2006] Andrade, A., Ruberg, G., Baião, F. A., Braganholo, V. P., and Mattoso, M. (2006). Efficiently Processing XML Queries over Fragmented Repositories with PartiX. In *International Conference on Extending Database Technology - Workshops*, pages 150–163.

[Angles and Gutierrez, 2008] Angles, R. and Gutierrez, C. (2008). Survey of Graph Database Models. *ACM Computing Surveys*, 40(1):1:1–1:39.

[Arnaut et al., 2011] Arnaut, D. E. M., Schroeder, R., and Hara, C. S. (2011). Phoenix - A Relational Storage Component for the Cloud. In *Proceedings of the International Conference on Cloud Computing*, pages 684 – 691.

[AWS, 2014] AWS, A. (2014). SimpleDB. http://aws.amazon.com/simpledb/. Acessado em Maio de 2014.

[Aykanat et al., 2008] Aykanat, C., Cambazoglu, B. B., and Uçar, B. (2008). Multi-level direct k-way hypergraph partitioning with multiple constraints and fixed vertices. *Journal of Parallel and Distributed Computing*, 68(5):609–625.

[Baker et al., 2011] Baker, J., Bond, C., Corbett, J., Furman, J. J., Khorlin, A., Larson, J., Leon, J.-M., Li, Y., Lloyd, A., and Yushprakh, V. (2011). Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proceedings of the Biennial Conference on Innovative Data Systems Research*, pages 223–234.

[Bellatreche and Benkrid, 2009] Bellatreche, L. and Benkrid, S. (2009). A Joint Design Approach of Partitioning and Allocation in Parallel Data Warehouses. In *Proceedings of the 11th International Conference on Data Warehousing and Knowledge Discovery*, DaWaK '09, pages 99–110. Springer-Verlag.

[Bellatreche et al., 2011] Bellatreche, L., Benkrid, S., Ghazal, A., Crolotte, A., and Cuzzocrea, A. (2011). Verification of partitioning and allocation techniques on teradata DBMS. In *Proceedings of the 11th international conference on Algorithms and architectures for parallel processing - Volume Part I*, pages 158–169. Springer-Verlag.

[Bellatreche and Boukhalfa, 2005] Bellatreche, L. and Boukhalfa, K. (2005). An evolutionary approach to schema partitioning selection in a data warehouse. In *Proceedings of the International Conference on Data Warehousing and Knowledge Discovery*, DaWaK'05, pages 115–125.

[Bizer and Schultz, 2009] Bizer, C. and Schultz, A. (2009). The Berlin SPARQL Benchmark. *International Journal on Semantic Web and Information Systems*, 5(2):1–24.

[Bonifati and Cuzzocrea, 2007] Bonifati, A. and Cuzzocrea, A. (2007). Efficient fragmentation of large xml documents. In *Proceedings of the international conference on Database and Expert Systems Applications*, pages 539–550. Springer-Verlag.

[Bonifati et al., 2004] Bonifati, A., Matrangolo, U., Cuzzocrea, A., and Jain, M. (2004). Xpath lookup queries in p2p networks. In *Proceedings of the 6th annual ACM international workshop on Web information and data management*, WIDM '04.

[Bordawekar and Shmueli, 2004] Bordawekar, R. and Shmueli, O. (2004). Flexible workload-aware clustering of xml documents. In *Database and XML Technologies*, Lecture Notes in Computer Science, pages 346–357. Springer Berlin / Heidelberg.

[Bordawekar and Shmueli, 2008] Bordawekar, R. and Shmueli, O. (2008). An algorithm for partitioning trees augmented with sibling edges. *Information Processing Letters*, 108(3):136–142.

[Brantner et al., 2008] Brantner, M., Florescu, D., Graf, D., Kossmann, D., and Kraska, T. (2008). Building a database on s3. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 251–264. ACM.

[Brocheler et al., 2010] Brocheler, M., Pugliese, A., and Subrahmanian, V. S. (2010). Cosi: Cloud oriented subgraph identification in massive social networks. In *Proceedings of the 2010 International Conference on Advances in Social Networks Analysis and Mining*, ASONAM '10, pages 248–255. IEEE Computer Society.

[Bunch et al., 2010] Bunch, C., Chohan, N., Krintz, C., Chohan, J., Kupferman, J., Lakhina, P., Li, Y., and Nomura, Y. (2010). An Evaluation of Distributed Datastores Using the AppScale Cloud Platform. In *IEEE Cloud10: International Conference on Cloud Computing*, pages 305–312.

[Buyya et al., 2008] Buyya, R., Yeo, C. S., and Venugopal, S. (2008). Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities. In *Proceedings of the 2008 10th IEEE International Conference on High Performance Computing and Communications*, pages 5–13.

[Catalyurek et al., 2009] Catalyurek, U. V., Boman, E. G., Devine, K. D., Bozdağ, D., Heaphy, R. T., and Riesen, L. A. (2009). A Repartitioning Hypergraph Model for Dynamic Load Balancing. *Journal of Parallel and Distributed Computing*, 69(8):711–724.

[Cattell, 2011] Cattell, R. (2011). Scalable sql and nosql data stores. *SIGMOD Record*, 39(4):12–27.

[Çatalyürek and Aykanat, 1999] Çatalyürek, Ü. and Aykanat, C. (1999). PaToH: Partitioning Tool for Hypergraphs. Technical report, Department of Computer Engineering, Bilkent University.

[Ceri and Pelagatti, 1982] Ceri, S. and Pelagatti, G. (1982). Allocation of operations in distributed database access. *IEEE Transactions on Computers*, 31(2):119–129.

[Ceri and Pelagatti, 1984] Ceri, S. and Pelagatti, G. (1984). *Distributed databases principles and systems*. McGraw-Hill, Inc.

[Chang et al., 2008] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. (2008). Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4.

[Chong et al., 2005] Chong, E. I., Das, S., Eadon, G., and Srinivasan, J. (2005). An efficient sql-based rdf querying scheme. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 1216–1227. VLDB Endowment.

[Clark et al., 2005] Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I., and Warfield, A. (2005). Live migration of virtual machines. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 273–286. USENIX Association.

[Cong et al., 2007] Cong, G., Fan, W., and Kementsietsidis, A. (2007). Distributed query evaluation with performance guarantees. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, page 509. ACM Press.

[Cooper et al., 2008] Cooper, B. F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.-A., Puz, N., Weaver, D., and Yerneni, R. (2008). PNUTS: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288.

[Corbett et al., 2012] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., and Woodford, D. (2012). Spanner: Google's Globally-Distributed Database. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pages 1–14.

[Curino et al., 2010] Curino, C., Jones, E., Zhang, Y., and Madden, S. (2010). Schism: a workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment*, 3:48–57.

[Curino et al., 2011] Curino, C., Jones, E. P. C., Popa, R. A., Malviya, N., Wu, E., Madden, S., Balakrishnan, H., and Zeldovich, N. (2011). Relational Cloud: A Database-as-a-Service for the Cloud. In *Proceedings of the Biennial Conference on Innovative Data Systems Research*, pages 235–240.

[Cuzzocrea et al., 2009] Cuzzocrea, A., Darmont, J., and Mahboubi, H. (2009). Fragmenting very large xml data warehouses via kmeans clustering algorithm. *International Journal on Business Intelligence and Data Mining*, 4:301–328.

[Dahlhaus et al., 1994] Dahlhaus, E., Johnson, D. S., Papadimitriou, C. H., Seymour, P. D., and Yannakakis, M. (1994). The Complexity of Multiterminal Cuts. *SIAM Journal on Computing*, 23:864–894.

[Das et al., 2013] Das, S., Agrawal, D., and El Abbadi, A. (2013). ElasTraS: An elastic, scalable, and self-managing transactional database for the cloud. *ACM Transactions on Database Systems*, 38(1):5.

[DeCandia et al., 2007] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. (2007). Dynamo: amazon's highly available key-value store. *SIGOPS Operating Systems Review*, 41(6):205–220.

[Egger, 2009] Egger, D. (2009). SQL in the Cloud. Master's thesis, Swiss Federal Institute of Technology Zurich (ETH).

[Fan, 2012] Fan, W. (2012). Graph pattern matching revised for social network analysis. *Proceedings of the 15th International Conference on Database Theory - ICDT '12*, page 8.

[Fiduccia and Mattheyses, 1982] Fiduccia, C. M. and Mattheyses, R. M. (1982). A linear-time heuristic for improving network partitions. In *Proceedings of the 19th Design Automation Conference*, DAC '82, pages 175–181. IEEE Press.

[Figueiredo et al., 2010] Figueiredo, G., Braganholo, V. P., and Mattoso, M. (2010). Processing queries over distributed xml databases. *Journal of Information and Data Management*, 3(1):455–470.

[Ford and Fulkerson, 1956] Ford, L. R. and Fulkerson, D. R. (1956). Maximal Flow through a Network. *Canadian Journal of Mathematics*, 8:399–404.

[Franz Inc, 2013] Franz Inc (2013). AllegroGraph 4.11. http://www.franz.com/agraph/allegrograph/.

[Galárraga et al., 2014] Galárraga, L., Hose, K., and Schenkel, R. (2014). Partout: A distributed engine for efficient rdf processing. In *Proceedings of the Companion Publication of the 23rd International Conference on World Wide Web Companion*, WWW Companion '14, pages 267–268. International World Wide Web Conferences Steering Committee.

[Galbiati, 2011] Galbiati, G. (2011). Approximating minimum cut with bounded size. In *Proceedings of the 5th international conference on Network optimization*, INOC'11, pages 210–215. Springer-Verlag.

[Gertz and Bremer, 2003] Gertz, M. and Bremer, J. M. (2003). Distributed xml repositories: Top-down design and transparent query processing. Technical report, Departement of Computer Science,University of California, Davis, CA, USA.

[Google, 2012] Google (2012). Google app engine. http://cloud.google.com/appengine/.

[Guttmann-Beck and Hassin, 2000] Guttmann-Beck, N. and Hassin, R. (2000). Approximation algorithms for minimum k-cut. *Algorithmica*, 27(2):198–207.

[Hauck and Borriello, 1995] Hauck, S. and Borriello, G. (1995). An evaluation of bipartitioning techniques. In *Proceedings of the 16th Conference on Advanced Research in VLSI (ARVLSI'95)*, ARVLSI '95, pages 383–. IEEE Computer Society.

[Hauglid et al., 2010] Hauglid, J. O., Ryeng, N. H., and Nørvåg, K. (2010). Dyfram: dynamic fragmentation and replica management in distributed database systems. *Distrib. Parallel Databases*, 28(2-3):157–185.

[Hose and Schenkel, 2013] Hose, K. and Schenkel, R. (2013). Warp: Workload-aware replication and partitioning for rdf. In *Data Engineering Workshops (ICDEW), 2013 IEEE 29th International Conference on*, pages 1–6.

[Jiewen Huang, 2011] Jiewen Huang, D. J. A. (2011). Scalable SPARQL Querying of Large RDF Graphs. *VLDB Endowment*, 4(11):1123–1134.

[Kanne and Moerkotte, 2006] Kanne, C.-C. and Moerkotte, G. (2006). A linear time algorithm for optimal tree sibling partitioning and approximation algorithms in natix. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06, pages 91–102. VLDB Endowment.

[Kaoudi and Manolescu, 2014] Kaoudi, Z. and Manolescu, I. (2014). Cloud-based rdf data management. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 725–729.

[Karypis and Kumar, 1998a] Karypis, G. and Kumar, V. (1998a). Multilevel Algorithms for Multi-Constraint Graph Partitioning. Technical Report 98-019, University of Minnesota, Department of Computer Science.

[Karypis and Kumar, 1998b] Karypis, G. and Kumar, V. (1998b). Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129.

[Karypis and Kumar, 1999] Karypis, G. and Kumar, V. (1999). Multilevel k-way hypergraph partitioning. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, DAC '99, pages 343–348. ACM.

[Kernighan and Lin, 1970] Kernighan, B. W. and Lin, S. (1970). An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell system technical journal*, 49(1):291–307.

[Khan and Hoque, 2010] Khan, S. I. and Hoque, A. S. M. L. (2010). A New Technique for Database Fragmentation in Distributed Systems. *International Journal of Computer Applications*, 5(9):20–24.

[Kling et al., 2010] Kling, P., Özsu, M. T., and Daudjee, K. (2010). Distributed xml query processing: Fragmentation, localization and pruning. Technical report, University of Waterloo.

[Klophaus, 2010] Klophaus, R. (2010). Riak core: building distributed applications without shared state. In *ACM SIGPLAN Commercial Users of Functional Programming*, CUFP '10, pages 14:1–14:1, New York, NY, USA. ACM.

[Kossmann, 2000] Kossmann, D. (2000). The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4):422–469.

[Kossmann et al., 2010] Kossmann, D., Kraska, T., and Loesing, S. (2010). An evaluation of alternative architectures for transaction processing in the cloud. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 579–590.

[Lakshman and Malik, 2009] Lakshman, A. and Malik, P. (2009). Cassandra: structured storage system on a p2p network. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, PODC '09, page 5. ACM.

[Laney, 2001] Laney, D. (2001). 3D Data Management: Controlling Data Volume, Velocity and Variety. Technical report, Gartner Inc.

[Laney, 2012] Laney, D. (2012). The Importance of 'Big Data': A Definition. Technical report, Gartner Inc.

[Ma and Schewe, 2003] Ma, H. and Schewe, K.-D. (2003). Fragmentation of xml documents. In *Simpósio Brasileiro de Banco de Dados*, pages 200–214.

[McCormick et al., 1972] McCormick, W., Schweitzer, P., and White, T. (1972). Problem decomposition and data reorganization by a clustering technique. *Journal of the Operational Research Society*, 20:993–1009.

[Mulay and Kumar, 2012] Mulay, K. and Kumar, P. S. (2012). Spovc: A scalable rdf store using horizontal partitioning and column oriented dbms. In *Proceedings of the 4th International Workshop on Semantic Web Information Management*, SWIM '12, pages 8:1–8:8. ACM.

[Navathe et al., 1984] Navathe, S., Ceri, S., Wiederhold, G., and Dou, J. (1984). Vertical partitioning of algorithms for database design. *ACM Transactions on Database Systems*, 9:680–710.

[Navathe and Ra, 1989] Navathe, S. and Ra, M. (1989). Vertical partitioning for database design: a graphical algorithm. *ACM SIGMOD International Conference on Management of Data*, 18:440–450.

[Nehme and Bruno, 2011] Nehme, R. and Bruno, N. (2011). Automated partitioning design in parallel database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1137–1148.

[Neumann and Weikum, 2009] Neumann, T. and Weikum, G. (2009). The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19(1):91–113.

[Noaman and Barker, 1999] Noaman, A. Y. and Barker, K. (1999). A horizontal fragmentation algorithm for the fact relation in a distributed data warehouse. In *Proceedings of the eighth international conference on Information and knowledge management*, CIKM '99, pages 154–161. ACM.

[Ozsu et al., 2013] Ozsu, M. T., Daudjee, K., and Hartig, O. (2013). Chameleon-db: A Workload-Aware Robust RDF Data Management System. Technical Report iv, University of Waterloo.

[Ozsu and Valduriez, 2011] Ozsu, T. and Valduriez, P. (2011). *Principles of Distributed Database Systems*. Third edition. Prentice-Hall.

[Papadimitriou and Steiglitz, 1998] Papadimitriou, C. H. and Steiglitz, K. (1998). *Combinatorial Optimization: Algorithms and Complexity*. Dover.

[Papadomanolakis and Ailamaki, 2004] Papadomanolakis, S. and Ailamaki, A. (2004). Autopart: Automating schema design for large scientific databases using data partitioning. In *Proceedings of the 16th International Conference on Scientific and Statistical Database Management*, SSDBM '04, pages 383–. IEEE Computer Society.

[Pavlo et al., 2012] Pavlo, A., Curino, C., and Zdonik, S. B. (2012). Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 61–72.

[Pujol et al., 2010] Pujol, J. M., Erramilli, V., Siganos, G., Yang, X., Laoutaris, N., Chhabra, P., and Rodriguez, P. (2010). The little engine(s) that could: scaling online social networks. In *Proceedings of the ACM SIGCOMM 2010 conference*, SIGCOMM '10, pages 375–386. ACM.

[Rao et al., 2011] Rao, J., Shekita, E. J., and Tata, S. (2011). Using Paxos to Build a Scalable, Consistent, and Highly Available Datastore. *VLDB Endowment*, 4(4):243–254.

[Rowstron and Druschel, 2001] Rowstron, A. I. T. and Druschel, P. (2001). Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, pages 329–350.

[Sacca and Wiederhold, 1985] Sacca, D. and Wiederhold, G. (1985). Database partitioning in a cluster of processors. *ACM Transactions on Database Systems*, 10(1):29–56.

[Saran and Vazirani, 1995] Saran, H. and Vazirani, V. V. (1995). Finding k cuts within twice the optimal. *SIAM Journal on Computing*, 24(1):101–108.

[Schroeder et al., 2012] Schroeder, R., Mello, R. S., and Hara, C. S. (2012). Affinity-based XML Fragmentation. In *15th International Workshop on the Web and Databases (WebDB 2012)*.

[Schütt et al., 2007] Schütt, T., Schintke, F., and Reinefeld, A. (2007). A Structured Overlay for Multi-dimensional Range Queries. In *Proceedings of the International Euro-Par Conference on Parallel Processing*, pages 503–513.

[Schütt et al., 2008] Schütt, T., Schintke, F., and Reinefeld, A. (2008). Scalaris: reliable transactional p2p key/value store. In *Proceedings of the 7th ACM SIGPLAN workshop on ERLANG*, ERLANG '08, pages 41–48. ACM.

[Sengupta, 2008] Sengupta, S. (2008). Sql data services: A lap around. In *Microsoft Professional Developers Conference*.

[Shnaiderman and Shmueli, 2009] Shnaiderman, L. and Shmueli, O. (2009). ipixsar: incremental clustering of indexed xml data. In *Proceedings of the International Conference on Extending Database Technology - Workshops*, pages 74–84. ACM.

[Shnaiderman et al., 2008] Shnaiderman, L., Shmueli, O., and Bordawekar, R. (2008). Pixsar: incremental reclustering of augmented xml trees. In *Proceedings of the 10th ACM workshop on Web information and data management*, WIDM '08, pages 17–24. ACM.

[Shute et al., 2013] Shute, J., Whipkey, C., Menestrina, D., Vingralek, R., Samwel, B., Handy, B., Rollins, E., Oancea, M., Littlefield, K., Ellner, S., Cieslewicz, J., Rae, I., Stancescu, T., and Apte, H. (2013). F1: A Distributed SQL Database That Scales. *Proceedings of the VLDB Endowment*, 6(11).

[Son and Kim, 2004] Son, J. H. and Kim, M. H. (2004). An adaptable vertical partitioning method in distributed systems. *Journal of Systems and Software*, 73(3):551–561.

[Stoica et al., 2003] Stoica, I., Morris, R., Liben-Nowell, D., Karger, D., Kaashoek, M., Dabek, F., and Balakrishnan, H. (2003). Chord: a scalable peer-to-peer lookup protocol for internet applications. *ACM Transactions on Networking*, 11(1):17 – 32.

[Stonebraker, 1986] Stonebraker, M. (1986). The Case for Shared Nothing. *Database Engineering*, 9:4–9.

[Ullmann, 1976] Ullmann, J. R. (1976). An Algorithm for Subgraph Isomorphism. *Journal of the ACM*, 23(1):31–42.

[Valduriez and Pacitti, 2004] Valduriez, P. and Pacitti, E. (2004). Data management in large-scale p2p systems. In *6th International Conference VECPAR*, pages 104–118.

[Voldemort, 2012] Voldemort, P. (2012). Project voldemort: A distributed database. http://project-voldemort.com/.

[Wehrle et al., 2005] Wehrle, P., Miquel, M., and Tchounikine, A. (2005). A model for distributing and querying a data warehouse on a computing grid. In *Proceedings of the 11th International Conference on Parallel and Distributed Systems - Volume 01*, pages 203–209.

[Wilkinson et al., 2003] Wilkinson, K., Sayers, C., Kuno, H., Reynolds, D., and Database, J. (2003). Efficient rdf storage and retrieval in jena2. In *EXPLOITING HYPERLINKS 349*, pages 35–43.

[Wu et al., 2011] Wu, E., Curino, C., and Madden, S. (2011). No bits left behind. In *5th Biennial Conference on Innovative Data Systems Research*, pages 187–190.

[Yang and Wu, 2013] Yang, M. and Wu, G. (2013). A workload-based partitioning scheme for parallel rdf data processing. In *Semantic Web and Web Science*, Springer Proceedings in Complexity, pages 311–324.

[Yang et al., 2012] Yang, S., Yan, X., Zong, B., and Khan, A. (2012). Towards effective partition management for large graphs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 517–528. ACM.

[Yang et al., 2013] Yang, T., Chen, J., Wang, X., Chen, Y., and Du, X. (2013). Efficient sparql query evaluation via automatic data partitioning. In Meng, W., Feng, L., Bressan, S., Winiwarter, W., and Song, W., editors, *Database Systems for Advanced Applications*, volume 7826 of *Lecture Notes in Computer Science*, pages 244–258. Springer Berlin Heidelberg.

[Zeng et al., 2013] Zeng, K., Yang, J., Wang, H., Shao, B., and Wang, Z. (2013). A distributed graph engine for web scale rdf data. *VLDB Endowment*, 6(4):265–276.

[Zilio, 1998] Zilio, D. C. (1998). *Physical Database Design Decision Algorithms and Concurrent Reorganization for Parallel Database Systems.* PhD thesis, University of Toronto.

[Zilio et al., 2004] Zilio, D. C., Rao, J., Lightstone, S., Lohman, G., Storm, A., Garcia-Arellano, C., and Fadden, S. (2004). Db2 design advisor: integrated automatic physical database design. In *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, pages 1087–1097. VLDB Endowment.

[Zuse Institute Berlin, 2012] Zuse Institute Berlin (2012). Scalaris - distributed transactional key-value store. http://code.google.com/p/scalaris/.