Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Informática

Dissertação de Mestrado

NodeWiz-R: Um Sistema P2P Relacional para a Descoberta de Recursos

José Flávio Mendes Vieira Júnior

Campina Grande, Paraíba, Brasil
Setembro - 2009

Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Informática

NodeWiz-R: Um Sistema P2P Relacional para a Descoberta de Recursos

José Flávio Mendes Vieira Júnior

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Redes de Computadores e Sistemas Distribuídos

Francisco Vilar Brasileiro
(Orientador)

Campina Grande, Paraíba, Brasil
©José Flávio Mendes Vieira Júnior, 04/09/2009

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

V658n
2009 Vieira Júnior, José Flávio Mendes.
NodeWiz-R: um sistema P2P relacional para descoberta de recursos / José Flávio Mendes Vieira Júnior . — Campina Grande, 2009.
100 f.: il.

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática.
Referências.
Orientador: Prof. Dr. Francisco Vilar Brasileiro.

1. Sistemas de Processamento Distribuído. 2. Sistema P2P. 3. Descoberta de Recursos. I. Título.

CDU – 004.75(043)

"NodeWiz-R: UM SISTEMA P2P RELACIONAL PARA A DESCOBERTA DE RECURSOS"

JOSÉ FLÁVIO MENDES VIEIRA JÚNIOR

DISSERTAÇÃO APROVADA EM 04.09.2009

FRANCISCO VILAR BRASILEIRO, PH.D
Orientador(a)

LUIS CARLOS ERPEN DE BONA, DR.
Examinador(a)

DENIO MARIZ TIMÓTEO DE SOUSA, DR.
Examinador(a)

CAMPINA GRANDE - PB

Resumo

Com a crescente popularização da Internet, surgiram grandes oportunidades para compartilhamento de recursos em larga escala. Os recursos compartilhados são os mais diversos, indo desde arquivos de conteúdo áudio-visual, documentos, grandes massas de dados produzidos por centros operacionais, até computadores e serviços disponíveis para a realização de processamento e armazenamento de informações. Um pré-requisito para o compartilhamento de recursos é a existência de um serviço que habilite os usuários a descobrirem os recursos que estão disponíveis no sistema. Em um sistema com a amplitude e a dinamicidade da Internet, é necessário que o serviço de descoberta seja distribuído e autogerenciável. Embora várias soluções para a implementação de serviços de descoberta distribuídos e autônomos tenham sido sugeridas, nenhuma delas consegue ser ao mesmo tempo expressiva (na forma como os atributos que descrevem os recursos são modelados e consultados), eficiente (em termos de latência e sobrecarga para resolver consultas) e ter alta cobertura (capacidade de obter grande parte ou todos os resultados possíveis). Neste trabalho apresentamos o NodeWiz-R, uma solução para a descoberta de recursos distribuída e autônoma que implementa uma camada relacional sobre um substrato peer-to-peer para prover simultaneamente todas essas propriedades. O NodeWiz-R implementa vários níveis de indexação que o permite localizar eficientemente as informações sobre os recursos descritos usando o modelo relacional. Para manter a carga do sistema balanceada, as tabelas são fragmentadas horizontalmente por toda a rede. A solução oferece ainda suporte à execução de consultas ricas expressas no padrão SQL. Os resultados obtidos através de simulações e experimentos com um protótipo mostram que o NodeWiz-R é mais eficiente que outras soluções, que utilizam técnicas de *flooding* para o processamento de consultas expressas no padrão SQL. O NodeWiz-R apresenta uma alta taxa de retorno de resultados com um baixo *overhead*, mesmo para consultas sobre dados que se encontram replicados na rede.

Abstract

With the ever increasing popularity of the Internet, great opportunities for large-scale resource sharing have emerged. Shared resources can be of several types, ranging from multimedia content files, documents, through massive data produced by operational research centres, until computers and services that perform processing and storage of information. The first requirement to enable resource sharing is the existence of a service that allows users to find what resources are available, and where. In a system with the Internet scale and dinamicity, it is necessary that the discovery service is distributed and autonomous. Although several solutions have been proposed for distributed and autonomous resource discovery services on the Internet, they all fail to provide at least one of the following important features: i) an expressive way to annotate resources, through its associated attributes; ii) efficient ways to index these attributes; and iii) high matching rates in the requests processed. In this work, we present NodeWiz-R, a distributed and self-managed resource discovery solution that implements a relational model atop of a peer-to-peer substrate to simultaneously provide all these three features. NodeWiz-R implements several indexing levels that enables it to find resource information, described using the relational model, in an efficient manner. In order to keep the system load balanced, the relational model tables are fragmented horizontally over the network. The solution supports expressive queries processing using the standard SQL language. Results obtained by simulations and confirmed by experiments with the system prototype, indicate that NodeWiz-R is more efficient than other flooding based solutions for distributed SQL query processing.

Agradecimentos

Inicialmente, agradeço a toda minha família pelo o apoio que foi me dado e por acreditarem sempre no meu potencial, especialmente meus pais, irmãos, meu sogro e minha sogra.

Agradeço imensamente a minha esposa Érika e minha filhinha Yasmin, por terem compactuado os momentos mais difíceis da realização desse trabalho, quando inúmeras vezes cheguei em casa tarde, cansado, estressado e não pude dar a atenção devida. Mas sempre foram compreensivas e me apoiaram para que eu pudesse levar esse trabalho adiante, principalmente nas diversas vezes que pensei em desistir. Com certeza, não há como recuperar o tempo perdido com a minha família, mas todos nós sabemos que foi por uma causa justa.

Agradeço ao meu orientador, Professor Francisco Brasileiro, que para economizar chamamos de Fubica. Ele que foi crucial no desenvolvimento desse trabalho, retirando as maiores pedras do caminho, e me mostrando a luz existente no fim do túnel.

Agradeço demais a todos os meus amigos que também compartilharam os momentos difíceis durante a realização desse trabalho e claro, alguns momentos de alegria e descontração. Aos que perguntavam, "E aí Zé Flávio, quando será a sua defesa?", sempre que me viam pelos corredores, de dia, de noite, vagando pelo LSD. Aos amigos do LSD, cuja a lista é grande, por isso não discorro pelo grande risco de esquecer alguém. Portanto, a todos aqueles que contribuíram direto e indiretamente, de qualquer forma, muito obrigado!

A todos vocês, meu muito obrigado.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A popularização da Internet tem propiciado o compartilhamento em larga escala, de recursos e conteúdo cujo o interesse é comum entre usuários. Esse compartilhamento abrange desde a infra-estrutura de comunicação, passando pelos recursos computacionais que estão em computadores pessoais e servidores, até o grande volume de dados que está armazenado de forma distribuída nesses recursos. Provedores de recursos são as entidades responsáveis pelos recursos e por torná-los acessíveis para os usuários interessados. Esses provedores, geralmente, fazem parte de diferentes domínios administrativos. Por isso, tanto os provedores como também os recursos, normalmente, encontram-se distribuídos geograficamente. Essa dispersão torna o compartilhamento desses recursos uma tarefa mais complexa do que em ambientes centralizados e controlados.

Considerando a natural distribuição dos recursos por diferentes domínios administrativos, torna-se cada vez mais necessário a disponibilidade de um conjunto de serviços que facilitem a localização desses recursos. Portanto, um serviço básico em ambientes de compartilhamento em larga escala é aquele que possibilita a descoberta dos recursos e de seus provedores [31; 9; 48]. O serviço de descoberta (*Resource Discovery*), a partir de uma descrição fornecida, retorna um conjunto com informações de provedores que detêm recursos que casam com essa descrição. O serviço implementa a abstração de um catálogo que armazena informações sobre os atributos de todos os recursos disponíveis, usando os valores desses atributos para gerar índices e tornar as consultas mais eficientes. Usuários podem, portanto, realizar buscas pelos recursos que estão disponíveis, selecionando apenas aqueles que mais se adequam às suas necessidades.

Uma característica importante de muitos dos sistemas de compartilhamento de recursos existentes é a autonomia dos participantes. Em *sistemas abertos* de compartilhamento de recursos, os provedores podem entrar e sair do sistema sem que exista um prévio anúncio e sem a existência de uma coordenação central. Essa característica tem duas implicações principais para a manutenção do catálogo: i) as informações sobre os recursos disponíveis precisam ser constantemente revalidadas (em particular, informações sobre recursos de provedores que deixaram o sistema precisam ser removidas); e ii) a responsabilidade de manutenção do catálogo deve ser compartilhada entre os provedores que estão efetivamente fazendo parte do sistema em um determinado instante de tempo. Dessa forma, é recomendável que o serviço de descoberta seja distribuído e autogerenciável, de modo a diminuir o esforço administrativo necessário para mantê-lo operacional e consistente, além de evitar que falhas não recuperáveis do mantenedor do catálogo levem o sistema inteiro ao colapso [60].

Dois exemplos concretos de sistemas abertos que necessitam de serviços de descoberta distribuídos e autogerenciáveis são o SegHidro [6] e o OurGrid [20]. O SegHidro permite o compartilhamento de dados ambientais entre pesquisadores espalhados pelo mundo. Já o OurGrid implementa uma infra-estrutura de grade computacional [28] para compartilhar a capacidade de processamento dos computadores ociosos distribuídos pela Internet. Em ambos, o serviço de descoberta apresenta-se como um elemento fundamental. Na grade computacional, localizando os processadores que estão disponíveis num determinado momento para a realização de um processamento distribuído. No sistema de compartilhamento de dados ambientais, localizando dados públicos sobre diversos aspectos do ambiente, como condições do tempo e clima, viabilizando pesquisas na área das ciências ambientais.

Sistemas de descoberta de recursos para sistemas abertos baseados em arquiteturas entre-pares ou *peer-to-peer* (*P2P*) [31; 35; 9; 48] foram propostos, aproveitando as características de descentralização, escalabilidade, autogerencimento e baixo custo oferecidas por essa arquitetura. Alguns desses sistemas são extremamente eficientes para armazenar e posteriormente localizar informações, associando os atributos referentes a um recurso a seus respectivos valores. Contudo, com o aumento da complexidade dos atributos que descrevem os recursos e dos requisitos das aplicações que desejam localizar esses recursos, surgiu a necessidade de serviços de descoberta mais expressivos, capazes de lidar com recursos que possuem um modelo semântico de representação mais complexo. Por exemplo, no caso do

provedor de dados ambientais, para cada conjunto de dados (*dataset*) o catálogo armazena informações sobre os atributos de localização geográfica e temporal, variáveis de interesse (*e.g.*, umidade, temperatura, precipitação etc), além de rótulos (*tags*) que podem ser associados tanto ao conjunto de dados como um todo, quanto a cada variável nele contido. Assim, tais recursos não podem ser descritos eficientemente por um modelo semântico simplificado, como os baseados em pares de atributo-valor, presente na maioria desses sistemas.

Além disso, de um modo geral, os sistemas P2P possuem limitações quanto ao formato de consultas, provendo suporte eficiente apenas aos tipos mais simplificados, como busca baseadas em palavras-chave (*keyword-based searches*), ou baseados em casamento exato de valores (*exact match queries*) geralmente usado por sistemas implementados sobre Tabelas *Hash* Distribuídas (DHT - do inglês, *Distributed Hash Tables*) [59; 51; 55; 53]. Em busca por palavras-chave, o usuário fornece as palavras que podem estar sendo utilizadas pelo sistema para indexar algum conteúdo. Os endereços dos conteúdos indexados por alguma das palavras-chave fornecidas são retornados e uma conexão pode ser estabelecida diretamente com o provedor do conteúdo para que ele possa acessar o recurso. Geralmente a busca é feita simplesmente enviando mensagens para todos os *peers* da rede [26]. Já nas buscas por casamento exato de valores, o valor exato (no caso de DHT, a chave) que indexa o conteúdo compartilhado deve ser conhecido. Portanto, apenas o conteúdo indexado por aquela chave é retornado.

Embora seja bastante eficiente para a localização de recursos individualmente, soluções baseadas em DHT não se mostram eficientes para realização de buscas pelas características dos recursos, já que o conteúdo é distribuído na DHT de acordo com o valor da chave gerada através de uma função de *hash*. Além disso, a proximidade dos valores dos atributos que descrevem os recursos é desconsiderada no momento da distribuição de suas informações na rede, já que na geração das chaves indexadoras essa propriedade não é considerada. Dessa forma, para que recursos que possuem um modelo de descrição mais complexo (*i.e.*, compostos por vários atributos e relacionamentos entre seus dados) sejam armazenados e consultados eficientemente, o serviço de descoberta precisa implementar um modelo de descrição expressivo, que permita combinar e correlacionar dados naturalmente, e que dê suporte a uma linguagem que permita expressar consultas complexas, envolvendo os relacionamentos existentes entre esses dados.

Uma possível solução para aumentar a expressividade do serviço de descoberta é a utilização de bases de dados relacionais para a implementação do catálogo. Os sistemas de banco de dados (SBD) possuem uma grande capacidade de armazenamento de dados com possibilidade de modelagem expressiva, possibilitando ainda uma recuperação eficiente das informações armazenadas. Os sistemas de bancos de dados distribuídos (SBDD) em particular, têm se destacado por sua arquitetura descentralizada que possibilita uma melhor distribuição da carga de trabalho, espalhando dados pela rede de forma controlada. Duas evoluções dos SBDD são os sistemas de bancos de dados federados [57] e os sistemas de multi-bancos de dados [14] que são capazes de atender a uma maior escala do que os seus antecessores. Entretanto, tais sistemas não são projetados para lidar de forma autônoma com a intermitência dos nós que armazenam as informações, o que pode causar indisponibilidade e inconsistência dos dados. Além disso, podem demandar esforços administrativos para reconfiguração do sistema, gerando um alto custo de manutenção e requerendo uma centralização da operação do sistema, o que é indesejável em um serviço de descoberta de recursos para sistemas abertos.

Os SBD P2P têm surgido como alternativa para ligar os dois mundos discutidos acima, oferecendo suporte à modelagem de dados expressiva e armazenamento eficiente, além de garantirem transparência e autogerenciamento [13]. Entretanto, tais sistemas não dispõem de um mecanismo de indexação eficiente, geralmente recorrendo a técnicas de inundação (*flooding*) para a disseminação de consultas mais elaboradas na rede, comprometendo a escalabilidade do sistema e gerando uma sobrecarga de mensagens indesejada. Outra consequência é que esses sistemas nem sempre conseguem obter uma alta *cobertura* das consultas (*i.e.*, nem todas as consultas conseguem retornar todos os casamentos possíveis), um aspecto importante para alguns sistemas que utilizam o serviço de descoberta [13]. Em sistemas baseados em técnicas de *flooding*, obter todos os resultados possíveis pode custar caro em termos de quantidade de mensagens enviadas para o processamento de uma operação, dependendo do tamanho da rede e da profundidade necessária para se alcançar todos os nós participantes.

1.1 NodeWiz-R

Ao longo desse capítulo, vimos que se faz necessário um serviço de indexação e descoberta de recursos mais expressivo, capaz de lidar bem e de forma autônoma, com recursos intermitentes e que possuem um modelo de descrição mais complexo. Vimos ainda que a arquitetura P2P confere algumas características desejáveis para um serviço de descoberta, mas que ainda possui algumas limitações, principalmente no modelo semântico para descrição dos dados e nos tipos de consultas geralmente permitidos. Contudo, é possível construir uma arquitetura para um serviço de descoberta de recursos que resolva de forma eficiente os problemas apresentados, utilizando soluções existentes. Para isso, elencamos os principais requisitos que uma solução deve atender:

- o sistema precisa ser autônomo (auto-gerenciável), sem a existência de um controle centralizado;
- deve ser uma solução naturalmente distribuída, assumindo o caráter distribuído dos provedores e dos recursos, além de não adicionar pontos centralizados de falha na arquitetura;
- deve implementar um modelo de dados expressivo, de modo a permitir relacionamentos entre as informações armazenadas e, consequentemente, consultas mais elaboradas;
- deve implementar um modo de armazenamento que facilite as operações de manutenção da base de dados, e lidar bem com a intermitência e dinamicidade dos provedores e dos recursos;
- deve implementar um mecanismo de roteamento que permita, a um baixo custo, alcançar uma alta taxa de cobertura das consultas.

O objetivo desse trabalho é propor uma solução para descoberta de recursos de forma distribuída e autônoma que visa resolver de forma eficiente o problema apresentado, satisfazendo os principais requisitos elencados. A solução chamada de NodeWiz-R, implementa uma camada relacional sobre um substrato P2P, de forma similar aos SBD P2P. Porém, ao invés de usar um sistema P2P não estruturado ou um estruturado baseado em DHT, usa-se

um sistema baseado em árvores k-dimensionais (*kd-trees*) [9] que confere ao NodeWiz-R as características que o distingue dos demais. O NodeWiz-R permite uma indexação eficiente tanto em termos de armazenamento como em termos de latência para a resolução de consultas. Ele garante ainda uma alta taxa de cobertura, a um baixo custo em termos de quantidade de informações que precisam ser trocadas para implementar o serviço (comparado com outras soluções que utilizam *flooding* para realização de consultas). O uso da abordagem de armazenamento de estado fraca (*soft-state*) garante uma facilidade de manutenção das informações no sistema. Além disso, o NodeWiz-R oferece suporte a consultas ricas expressas no padrão SQL, o que facilita em vários aspectos a utilização do sistema.

Os resultados de simulações, mostraram que o NodeWiz-R é mais eficiente que a solução PeerDB, que utiliza técnicas de *flooding* para o processamento de consultas distribuídas. O NodeWiz-R apresenta uma alta taxa de retorno de resultados com um baixo *overhead*, mesmo para consultas sobre dados que se encontram replicados na rede.

1.2 Organização da Dissertação

O restante do trabalho está organizado da seguinte forma. O Capítulo 2 apresenta o estado-da-arte, sumariando as principais soluções para a descoberta de recursos em sistemas abertos propostas na literatura e discutindo suas limitações. No Capítulo 3, é apresentada a formalização do problema e é descrito em mais detalhes o problema que se pretende atacar nesse trabalho. O Capítulo 4 propõe a solução, apresenta a arquitetura do NodeWiz-R e descreve o seu funcionamento. O Capítulo 5 descreve como foi feita a implementação do protótipo do NodeWiz-R e os detalhes mais importantes relacionados à sua implementação. O Capítulo 6 apresenta os resultados da avaliação quantitativa realizada usando um modelo simulado do sistema e os resultados de uma avaliação experimental com o protótipo desenvolvido. Finalmente, o Capítulo 7 conclui o trabalho apresentando as considerações finais e uma indicação dos trabalhos futuros que podem ser realizados.

Capítulo 2

Estado da Arte

2.1 Arquitetura Peer-to-Peer (P2P)

Arquiteturas P2P surgiram como uma alternativa à tradicional arquitetura cliente/servidor. Em forte ascensão na década de oitenta, a arquitetura cliente/servidor tinha grande vantagem competitiva ao possibilitar um menor custo de implantação em comparação ao antigo modelo centralizado baseado em *Mainframes*. No modelo centralizado, tudo é feito pelo *mainframe*, o que o torna um gargalo quando é sobrecarregado com um aumento de requisições ao sistema. Já no modelo cliente/servidor, os computadores clientes também realizam processamento. Entretanto, a figura central do servidor não deixa de existir, realizando ainda grande parte da computação, deixando o sistema dependente de seu desempenho. Além disso, esse modelo arquitetural ainda requer que intervenções administrativas sejam realizadas para manutenção da parte centralizada do sistema, como planejamento e implementação de políticas de *backup* e tolerância a falhas.

Diferente dessas, na arquitetura P2P os computadores que compõem a rede (*peers*) tem a mesma capacidade e responsabilidades, podendo atuar como cliente e servidor ao mesmo tempo [62]. Esse modelo arquitetural minimiza a carga de trabalho nos servidores e maximiza o desempenho do sistema como um todo [41]. A comunicação entre os *peers* é feita de forma direta, sem a intervenção ou presença de um mediador central. A entrada de um novo *peer* na rede se dá através de uma solicitação a outro *peer* que já faça parte do sistema e a sua saída pode ocorrer a qualquer momento, sem aviso prévio. Cada *peer* mantém informações sobre outros *peers* participantes, criando assim uma estrutura de conexão entre eles. Quando

uma requisição do usuário é enviada, o *peer* que a recebe verifica se ele é responsável por atendê-la. Caso seja, o processamento é realizado e os resultados são retornados para o requisitante. O mesmo *peer* pode ainda encaminhar a requisição para outros *peers* conhecidos. A forma como as requisições são encaminhadas pela rede depende do modelo da arquitetura implementada, que pode ser estruturado ou não estruturado.

Nas redes P2P não estruturadas, também conhecidas como redes P2P puras, não existem regras que definem como a conexão entre os *peers* é formada. Geralmente, o número de conexões é fixo e são definidas de forma aleatória, permitindo que um nó se conecte com a mesma probabilidade a qualquer outro nó da rede [62]. As requisições são encaminhadas por propagação (*flooding*) para todos os nós em que há uma conexão direta estabelecida. Para limitar o número de mensagens transmitidas, um tempo de vida (TTL, do inglês *Time-To-Live*) é associado a cada requisição, indicando o número de saltos que a requisição deve dar desde o seu ponto de partida. Outras versões dessa técnica foram propostas como *random walks* e *multiple random walks* [62]. Entretanto, tais técnica ainda têm se mostrado pouco eficiente devido à sobrecarga de mensagens geradas na rede e a baixa cobertura obtida uma vez que não há garantias de que todos os nós relevantes serão alcançados. Apesar disso, a técnica de *flooding* tem sido amplamente utilizada por várias implementações de sistemas P2P. Já nas redes P2P estruturadas, estruturas de dados são usadas para melhorar o encaminhamento das requisições pela rede, tais como tabelas *hash* distribuídas (DHT - *Distributed Hash Table*) [53; 7; 51; 55] e árvores distribuídas [23; 52; 9]. O uso dessas estruturas permite que as requisições sejam encaminhadas para um seleto número de *peers*, propiciando um aumento na qualidade do serviço como um menor tempo de resposta e uma menor quantidade de mensagens trocadas é requerida.

As redes P2P foram inicialmente projetadas com o objetivo de prover o compartilhamento de arquivos de forma colaborativa entre milhares de usuários. Entre os precursores dessa arquitetura estão o Napster [4], Gnutella [1] e Kazaa [3]. Nesses sistemas, cada *peer* armazena uma parte dos arquivos que estão sendo compartilhadas na rede. Mas para que os arquivos possam ser compartilhados, é preciso saber em quais *peers* eles estão armazenados. Essa é a chamada etapa de descoberta dos recursos, onde o usuário submete uma consulta ao sistema que é processada e a informação sobre os arquivos disponíveis e seus respectivos *peers* é devolvida para a aplicação cliente do usuário. A transferência de arquivos é feita

diretamente numa conexão estabelecida entre o *peer* que armazena o arquivo requisitado e o cliente. A forma como a consulta é processada, os tipos de consultas permitidas, os tipos de recursos que podem ser indexados e como eles são descritos no sistema, podem variar de sistema para sistema. Por exemplo, no caso do Napster, as consultas são processadas de forma centralizada, apenas pesquisas baseadas em palavras-chave são permitidas (*keyword-based searches*), apenas arquivos podem ser indexados e sua descrição é feita através da atribuição de rótulos (*tags*) ao identificador que é o próprio nome do arquivo.

Portanto, os primeiros sistemas P2P eram limitados de várias formas, como por exemplo a falta de suporte a buscas baseadas em conteúdo e o compartilhamento somente de arquivos. Isso motivou e ainda tem motivado os estudos nessa área visando prover um serviço de indexação mais expressivo, bem como estendendo o uso da arquitetura para compartilhamento de recursos que podem ser informações, serviços, poder computacional etc. Além disso, arquiteturas P2P são inerentemente atrativas para o desenvolvimento de sistemas distribuídos de modo geral, principalmente os que devem operar em larga escala como grades computacionais e sistemas para descoberta de recursos na Internet.

2.2 Arquiteturas Peer-to-Peer baseadas em DHT

Nos últimos anos, arquiteturas P2P baseadas em DHT se tornaram bastante populares devido a sua eficiência para processamento e roteamento de operações, além de serem altamente escaláveis, mesmo para redes altamente dinâmicas (onde a entrada e saída dos nós é freqüente). Várias DHTs para arquiteturas P2P foram propostas como Chord [59], CAN [51], Pastry [55] e Bamboo [53]. Uma DHT funciona como uma tabela *hash* convencional operando de forma distribuída numa rede. A idéia é que a cada *peer* e recursos no sistema sejam atribuídos um identificador único (ID). O ID do *peer* define o lugar que ele deverá ocupar na topologia lógica da rede. O ID associado ao recurso define deterministicamente o lugar na rede onde ele será armazenado. Cada *peer* é responsável por um subespaço de IDs bem definidos. Cada informação sobre um recurso é armazenada em um *peer* responsável pela faixa do seu ID. Uma consulta é feita pelo ID do recurso e é roteada até o *peer* que está na zona que armazena a chave requisitada [53].

DHTs conseguem indexar eficientemente os itens armazenados e fazer o balanceamento

da carga do sistema através do particionamento dos dados e roteamento de consultas entre os diversos subespaços. DHTs garantem ainda que o roteamento entre dois nós requer somente $O(log(N))$ mensagens, onde N é o número de nós da rede. Entretanto ela é limitada a um único tipo de consulta que especifica um casamento exato entre o atributo e seu valor. Além disso, uma DHT não preserva a proximidade semântica dos dados, uma vez que eles são fragmentados pela rede de acordo com as chaves que lhes são atribuídas. Isso faz com que consultas mais elaboradas incluindo múltiplos atributos e faixas de valores (*range queries*) não possam ser executadas de forma natural.

2.3 Sistemas para Descoberta de Recursos

A fim de solucionar o problema da descoberta de recursos em redes amplamente distribuídas, surgiram várias propostas de sistemas baseados em arquiteturas P2P [9; 48; 34; 12; 47; 10; 35; 55; 51; 59; 44; 45]. Mas apesar de serem altamente escaláveis, esses sistemas apresentam limitações como: (i) a ausência de um modelo semântico bem definido para a descrição dos recursos, que possibilitasse a descrição de qualquer tipo de recurso, (ii) limitada capacidade de processamento de consultas, permitindo que apenas tipos simplificados de consultas fossem submetidas e (iii) o uso de técnicas de inundação para o roteamento de consultas. Nesta seção, alguns sistemas baseados em diferentes arquiteturas são discutidos.

2.3.1 Sistemas baseados em DHT

Sistemas apoiados sobre uma arquitetura P2P estruturada baseada em DHT foram propostos. Alguns desses sistemas são descritos a seguir.

SWORD

SWORD [48] é um serviço de descoberta de recursos para sistemas distribuídos em larga escala, que vem sendo utilizado para localizar computadores no PlanetLab[1]. SWORD permite que os usuários descrevam os recursos desejados como uma topologia de grupos interconectados com características intra-grupos, inter-grupos e características individuais de cada

[1]PlanetLab: http://www.planet-lab.org/

nó. Desta forma, SWORD possibilita expressar características entre os nós, como latência e largura de banda, essencial para o escalonamento eficiente de tarefas em aplicações que demandam baixa latência. Os recursos podem ser descritos de duas formas: através de uma sintaxe XML proprietária (SWORD XML) ou Condor ClassAd [50]. Dois tipos de linguagem de consulta são suportados: *SWORD's XML Specification Language* ou EBNF [48]. SWORD utiliza DHTs para fazer a distribuição das informações sobre os recursos por várias sub-regiões da rede, como também para fazer o roteamento das consultas para os nós da sub-região que são responsáveis por aquelas informações. Cada nó da rede é responsável por um atributo e um subconjunto de valores. Isso permite que consultas por múltiplos atributos e por faixa (*multi-attribute queries*) sejam executadas. Entretanto, muitos nós são visitados quando uma consulta possui mais de um atributo, o que pode levar a uma transferência excessiva de dados sobre a rede. As consultas são enviadas para servidores que as encaminham para os *peers* responsáveis pelos atributos especificados na consulta. Essa abordagem, além de adicionar pontos centralizados de falha, também implica em problemas de indisponibilidade das informações, uma vez que a falha em um servidor tornará indisponíveis os vários *peers* gerenciados por ele. Além disso, as linguagens utilizadas para descrição e consulta apresentam baixa expressividade, pois não permitem combinar e correlacionar dados, além de serem pouco amigáveis.

PIER

Barren, Huebsch *et al.* [7; 34] apresentam PIER, um processador de consultas relacional de propósito geral baseado em uma arquitetura P2P e apoiado em uma estrutura de DHT. Ele permite a localização de recursos na sua rede, utilizando vários tipos de consultas, como *exact-match queries*, naturalmente provido pela DHT, e ainda consultas com operações relacionais (como junções). Algoritmos comuns em sistemas de bancos de dados distribuídos são utilizados em PIER para otimizar o processamento de consultas (*Symetric Hash Join* [63] e *Bloomjoin* [43]). Os dados são distribuídos pela rede usando a implementação de DHT Bamboo [53]. Para que possa indexar recursos por todos de seus atributos, PIER sobrepõe várias DHTs, uma para cada atributo. Isso permite que consultas por múltiplos atributos sejam realizadas. Entretanto, assim como no trabalho de Oppenheimer *et al.* [48], uma operação de atualização pode comprometer a escalabilidade do sistema, quando múltiplos atributos

precisam ser atualizados. Além disso, PIER faz *broadcast* para conseguir realizar alguns tipos de consultas mais elaboradas, como as que envolvem operações de junção. O uso de *broadcast* implica na geração de tráfego desnecessário na rede, além de não garantir que todos os nós sejam alcançados, o que não assegura que todos as informações disponíveis na rede sejam retornados quando requisitadas. PIER também adota uma linguagem de consulta pouco conhecida e pouco amigável (*Unnamed Flow Language - UFL* [34]), o que dificulta sua utilização.

RDF/S sobre DHTs

Sidirourgos, Heine, Liarou, *et al.* [58; 33; 39] apresentam soluções baseadas em DHT para prover um eficiente armazenamento de informações sobre esquemas RDF (*Resource Description Framework*) [16], bem como o roteamento de consultas expressivas sobre eles. RDF/S é uma linguagem de representação de informações na Web. RDF armazena informações em conjuntos de triplas, compostas por assunto (*subject*), predicado (*predicate*) e um objeto (*object*). A semântica das triplas é definida pelo modelo teórico e por regras, que definem como uma tripla pode ser gerada. RDF permite que informações se relacionem, através de um conceito de classes e entidades. Propriedades descrevem as características de uma classe ou relacionamentos entre classes [58]. Essas soluções propõem que cada tripla seja indexada pela DHT, atribuindo uma chave para cada componente da tripla. Dessa forma, é possível consultar uma tripla por assunto, predicado ou objeto, já que as consultas também são baseadas em conjuntos de triplas. O maior problema é o *overhead* de mensagens gerado pelo sistema para publicar as informações, já que uma mesma tripla é armazenada em três *peers* ao mesmo tempo. Além disso, operações de consulta também geram um *overhead* indesejado, uma vez que aquelas formadas por mais de uma tripla são avaliadas fazendo várias combinações com todas as triplas, fazendo com que várias chaves de busca na DHT sejam geradas, uma para cada combinação possível. Por isso, uma operação de consulta pode requerer que uma elevada quantidade de mensagens sejam trocadas pelo sistema, para a realização de várias buscas simultâneas, uma para cada chave gerada, a fim de satisfazer uma única operação de consulta. Em Heine *et al.* [33], resultados intermediários são encaminhados até o último *peer* que deve responder a uma consulta, gerando tráfego em consultas que requerem grandes quantidades de dados.

Outras soluções baseadas em DHT apresentadas por Bharambe, Cai, Spence, *et al.* [10; 18; 25], possuem limitações semelhantes às já apresentadas nessa seção, como (i) a necessidade de sobrepor várias DHTs para indexar cada atributo da descrição de um recurso, (ii) ineficiência para lidar com consultas contendo múltiplos atributos, (iii) não preservação da proximidade semântica dos recursos, dificultando ainda mais sua localização no sistema. Portanto, de modo geral, os sistemas baseados em DHT ainda não atendem os requisitos básicos para a indexação e posterior localização de recursos com um modelo de representação mais complexo.

2.3.2 Sistemas apoiados em Banco de Dados

Sistemas de bancos de dados tem evoluído para grandes plataformas distribuídas, tornando-se também uma solução viável para aplicações de sistemas abertos, tal como sistemas P2P [17]. Essa evolução além de ser uma tendencia, se justifica pelas limitações dos sistemas de bancos de dados distribuídos, que possuem topologia estática e requerem um dispendioso trabalho administrativo [56]. A combinação de abordagens P2P com tecnologias de bancos de dados (SBD P2P) têm emergido, explorando a eficiente infra-estrutura de busca provida por algumas soluções P2P, aliado com a expressividade que é provida pelas tecnologias de bancos de dados para a descrição de recursos e especificação de consultas [17]. Em outras palavras, os SBDs P2P estendem os sistemas P2P puros, oferecendo suporte a uma modelagem de dados mais expressiva e armazenamento eficiente, além de garantirem transparência nas operações e autogerenciamento, propiciado pelo substrato P2P sobre o qual se apoiam. Um SBD P2P é formado por uma coleção de repositórios locais autônomos que interagem entre si assim como nas tradicionais redes P2P [13]. Nos SBD P2P que se apoiam em sistemas P2P estruturados, os nós participantes mantêm informações de roteamento que minimizam o número de nós pelos quais uma consulta precisa ser encaminhada. Já aqueles que se apoiam sobre sistemas P2P não estruturados, usam técnicas de *flooding* para realizar operações de consulta na rede. Dentre alguns SBD P2P estão AmbientDB e PeerDB que são descritos a seguir.

AmbientDB

AmbientDB [12] é uma arquitetura para localização de informações e processamento de consultas complexas em uma rede P2P *Ad hoc*. As informações são armazenadas em sistemas de bancos de dados, que podem ser usados para armazenar informações sobre os recursos disponíveis na rede. Assim como em PIER, AmbientDB usa várias DHTs (Chord [59]) para indexar as informações na rede P2P e garantir a escalabilidade. Além disso, permite a execução de consultas relacionais no padrão SQL. Em AmbientDB, o esquema de descrição de dados é bem definido, implementando conceitos de tabelas locais, tabelas distribuídas e tabelas particionadas. Mas, assim como em PIER, AmbientDB não possui um mecanismo eficiente para roteamento de consultas mais elaboradas (*e.g.* consultas que requerem operações de junção), diferente das consultas por valores exatos que podem ser realizadas eficientemente através das DHTs. Assim, AmbientDB faz *broadcast* para executar algumas operações de consulta, por exemplo, as que requerem operações de junção. Além disso, por não dispor de nenhum mecanismo para balancear a carga dos nós do sistema, ele descarta algumas operações de *INSERT* quando a capacidade máxima do índice local de um *peer* é atingida. Isso faz com que um índice incompleto seja criado, reduzindo o número de informações que podem ser encontrados no sistema.

PeerDB

PeerDB [47] utiliza um SBD relacional sobre uma arquitetura P2P não estruturada, possibilitando o compartilhamento de informações que estão armazenadas nos bancos de dados locais de cada nó. Em outras palavras, PeerDB é uma rede de nós *database-enabled*. PeerDB utiliza uma técnica baseada em agentes móveis para executar as consultas remotas ao invés de *message-passing*, abordagem comumente utilizada por outros sistemas P2P para troca de mensagens de controle e dados. Ele fornece suporte a consultas no padrão SQL, que são executadas no banco de dados local de cada nó. Porém, por utilizar uma arquitetura P2P não estruturada, PeerDB não garante que uma consulta irá retornar todas as informações disponíveis que casam com determinada consulta. Além disso, o mecanismo de inundação para encaminhamento de consultas pode ter um alto custo, dependendo da profundidade que se necessita enviar as consultas para obter uma cobertura aceitável.

2.3.3 Sistemas P2P baseados em Árvores Distribuídas

A ausência de sistemas para descoberta de recursos com suporte a consultas mais complexas, como consultas por faixa de valores (*range queries*) e com múltiplos atributos, fizeram com que novas soluções fossem propostas neste sentido. Sistemas com índices de roteamento estruturados baseados em árvore distribuída foram propostos de forma a permitir que estes tipos de consultas fossem executadas de forma natural, ao contrário dos sistemas baseados em DHTs. Alguns desses sistemas são discutidos a seguir.

P-Trees

Crainiceanu *et al.* [23] apresenta P-Trees, uma estrutura de índice P2P para aplicações de descoberta de recursos baseada em árvores distribuídas. A estrutura de P-Trees é baseada em uma Árvore B+ [27] e usa a DHT Chord [59] como mecanismo de roteamento. A idéia principal é dar uma noção de estar mantendo uma árvore B+ global e consistente, no entanto cada nó mantém uma árvore B+ semi-independente. Cada nó do sistema mantém o caminho para o nó *root* mais à esquerda e os nós folha armazenam os dados da rede. As consultas são processadas recursivamente do *root* até os nós folha, escolhendo a sub-árvore que contém o valor requisitado a cada passo da busca. Periodicamente, a árvore B+ de cada *peer* é atualizada quando um *peer* ou item de dado é adicionado ou removido do sistema. P-Trees permite avaliar eficientemente consultas por faixa de valores além das consultas com casamento exato de valores. O maior problema dessa arquitetura está no enorme custo para se atualizar as tabelas de roteamento quando um novo *peer* ou item de dado é adicionado no sistema, já que cada *peer* indexa todos os dados. Além disso, P-Trees se restringe para cenários de aplicações onde cada *peer* armazena informações apenas sobre um ou poucos recursos.

NodeWiz

Basu, Brasileiro *et al.* [9; 8; 15] apresentam o NodeWiz, um sistema P2P para descoberta de recursos cujo principal objetivo é permitir que consultas por faixa de valores e com múltiplos atributos sejam executadas eficientemente em um ambiente distribuído, mantendo um único índice distribuído. Ele é implementado sobre um conjunto de nós (*peers*) que armazenam coletivamente anúncios de recursos, enviados por provedores de serviços, e respondem a

consultas de clientes. Os *peers* armazenam meta-informações que são enviadas periodicamente pelos originadores/provedores. Essas informações são formatadas em um conjunto de pares atributo-valor, que devem conter os valores atualizados dos atributos dos recursos, junto com a URL do recurso. A URL serve para determinar a localização do recurso.

O NodeWiz organiza as informações publicadas em um espaço de atributos multidimensional onde cada *peer* é responsável por um subespaço do espaço total de atributos do sistema. Para distribuir esses dados pelos *peers* da rede, o NodeWiz implementa uma árvore k-dimensional (*kd-tree*) que determina qual *peer* é responsável por determinado subespaço. Como uma *kd-tree* é basicamente uma árvore binária de pesquisa, faixas de valores podem ser encontradas eficientemente. Por exemplo, o número de saltos necessários para que uma mensagem alcance o *peer* destino e o tamanho da tabela de roteamento são uma função logarítmica do número de *peers* do sistema. Além disso, o balanceamento da carga de trabalho é feito naturalmente, pois as consultas são processadas apenas nos *peers* responsáveis pelos valores de atributos requisitados, evitando a difusão de consultas por toda a rede. A Figura 2.1 mostra um exemplo de como o espaço de atributos de consultas e anúncios são divididos entre os *peers*. As folhas da árvore representam os *peers* da rede NodeWiz.

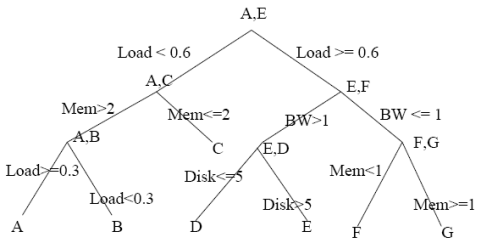


Figura 2.1: Exemplo da divisão do espaço de atributos no NodeWiz [8]

Level	Attr	Min	Max	IP Addr
0	Load	0.6	+inf	E
1	Mem	0	2	C
2	Load	0	0.3	B

Figura 2.2: Exemplo de uma tabela de rotas do NodeWiz [8]

Cada *peer* NodeWiz mantém uma tabela de roteamento que armazena informações parciais sobre a distribuição do espaço de atributos entre os *peers* do sistema. A tabela de roteamento permite que um *peer* envie mensagens (consultas ou anúncios) para outros *peers* de forma seletiva, de acordo com os atributos mantidos por cada *peer* na rede. A Figura 2.2 mostra como é organizada a tabela de rotas de um *peer* NodeWiz.

O processamento das consultas do NodeWiz é visto como um casamento (*match-making*) distribuído, onde elas são roteadas através da rede P2P até encontrar os *peers* que mantêm

os dados para a faixa de valores desejada. Por exemplo, uma consulta enviada para o *peer A*, que tem sua tabela de roteamento é apresentada na figura 2.2, seria roteada para os *peers* cuja entrada na sua tabela de roteamento combinasse com os requisitos da consulta. Os *peers* que recebem a consulta realizam uma busca na sua base de anúncios, e as URLs dos recursos que casam com os requisitos da consulta são retornadas para o cliente como resultado. A mesma consulta pode ainda ser roteada para outros *peers* apontados pela sua tabela de roteamento cujo espaço de atributos casa com os requisitos da consulta. Atualmente o NodeWiz oferece suporte à consultas por faixa de valores e com vários atributos. Entretanto, tais consultas não permite expressar qualquer relacionamento entre as informações que descrevem os recursos no sistema.

Além da divisão das informações pelo sistema através da estrutura de árvore, o NodeWiz provê um mecanismo explícito para balanceamento da carga de trabalho dos *peers*. Esse mecanismo permite que o *peer* que contém um maior número de dados e que está recebendo um maior número de consultas possa dividir essa carga de trabalho com outro *peer* menos sobrecarregado. Detalhes sobre a abordagem utilizada pelo NodeWiz para realização do balanceamento da carga do sistema são descritos em [9; 15]

Outro aspecto importante é que o NodeWiz também implementa técnicas para tolerar falhas que garantem a disponibilidade do sistema [15]. Ele mantém um mecanismo distribuído de detecção de falhas. Quando uma falha ocorre, a rede P2P detecta e um outro nó que já faz parte do sistema assume a responsabilidade sobre o espaço de atributos do nó falho. As publicações dos recursos mantidas pelos nós falhos são perdidas, mas são novamente armazenadas quando o serviço de publicação de dados republica as informações.

2.3.4 Outras Arquiteturas P2P

Heimbigner [32] propõe o uso do conhecimento contido na estrutura das mensagens (seja ela de consulta ou atualização de informações), para se fazer uma distribuição eficiente das mensagens em redes P2P. Essa técnica é conhecida como distribuição de mensagens baseada em conteúdo - CBR (do inglês *Content-Based Routing*). Seu novo trabalho propõe estender significativamente a expressividade das consultas que podem ser definidas em seu sistema, sem sacrificar a eficiência. A solução apóia-se em Siena [19], uma arquitetura P2P de duas camadas, semelhante à arquitetura baseada em super-peers [64], que provê o mecanismo de

distribuição de mensagens baseada no seu conteúdo. Como as mensagens em Siena também são baseadas em pares de atributo-valor, o novo modelo estende para que triplas contendo atributo-operador-valor sejam suportados. Dessa forma, consultas contendo outros operadores, como operadores de faixa, podem ser realizadas. Não obstante, apesar de estender a capacidade do sistema para dar suporte a consultas mais expressivas, o formato das consultas ainda é bastante simplificado, não permitindo assim que consultas que expressem relacionamentos entre os dados sejam realizadas.

2.4 Conclusão

Neste capítulo, discutimos diversos sistemas para descoberta de recursos baseados em diferentes modelos arquiteturais. Em particular, sistemas baseados na arquitetura P2P, por serem mais adequados para o desenvolvimento de uma nova solução para a descoberta de recursos amplamente distribuída. Diversas soluções baseadas nessa arquitetura foram propostas.

Entretanto, localizar eficientemente os recursos que estão sendo compartilhado em redes de compartilhamento é um problema não trivial. Isso se deve às dificuldades naturais do problema como a distribuição, intermitência e heterogeneidade dos recursos, além de falta de um esquema de nomes bem definido dos atuais sistemas, para descrever qualquer tipo de recurso. Nesse ambiente, sistemas com suporte a uma semântica de descrição mais complexa e tipos de consultas mais ricas são necessários para que recursos cujo modelo de descrição é naturalmente mais complexo, sejam descritos e localizados eficientemente.

Sistemas que provêem buscas baseadas em *keywords* são insuficientes para alcançar uma alta precisão [62]. Já os sistemas atuais, que permitem buscas por faixa de valores e com múltiplos atributos (*range-queries*) são mais expressivos, entretanto, não permitem combinar e correlacionar dados. Dessa forma, tais sistemas não atendem aos requisitos para uma solução para o problema. Sistemas não-estruturados permitem que consultas mais elaboradas sejam enviadas. Entretanto, os mecanismos para propagação de consultas pela rede são ineficientes, gerando tráfego excessivo e desnecessário, comprometendo a escalabilidade do sistema. Já os sistemas estruturados, são mais atrativos neste sentido, pois possibilitam que consultas possam ser respondidas de forma eficiente, com baixa sobrecarga de mensagens na rede. Alguns ainda fazem o balanceamento da carga do sistema naturalmente e toleram

falhas. Entretanto, tais sistemas carecem ainda de um modelo de dados mais expressivo, como os utilizados pelos SBDs, além da necessidade de proverem suporte a uma linguagem de consulta mais rica.

Dentre os sistemas estudados, o NodeWiz apresenta-se como o mais atrativo no sentido de prover um sistema mais completo, fornecendo uma arquitetura autogerenciável, escalável, eficiente em termos de armazenamento e consultas, além de tolerar falhas. Mas assim como os demais sistemas propostos, o mesmo implementa um modelo de dados simplificado, baseado em uma única estrutura de dados que armazena pares atributo-valor, o que impossibilita a indexação de recursos mais complexos e que consultas mais elaboradas sejam tolerados de forma natural e eficiente. Os sistemas de descoberta apoiados em SBDs provêem uma semântica mais rica, podendo descrever recursos usando o modelo relacional. Entretanto carecem de um mecanismo de indexação mais eficiente para localizar informações em várias tabelas, distribuídas em domínios administrativos diferentes.

Atualmente, indexar recursos complexos provendo suporte a consultas elaboradas em um sistema distribuído autogerenciável tem sido considerado um dos maiores problemas de sistemas para descoberta de recursos em geral. Esse trabalho visa mitigar tais problemas propondo uma nova solução para a descoberta de recursos na Internet, combinando diversas abordagens propostas pelos sistemas aqui apresentados.

Capítulo 3

Formalização do Problema

No Capítulo 1, foi apresentado superficialmente o problema relacionado à descoberta de recursos e a necessidade de uma solução capaz de lidar com recursos que exigem um modelo de descrição mais complexo e com uma linguagem de consulta expressiva. Neste capítulo será especificado em mais detalhes o problema que está sendo resolvido, além de explicitar os requisitos que precisam ser atendidos por uma solução.

Seja $P_t = \{p_1, p_2, ..., p_n\}$ o conjunto de provedores de recursos que estão ativos em um determinado instante de tempo t. Cada provedor $p_i \in P_t$ detêm um conjunto de recursos $R_i = \{r_1, r_2, ..., r_m\}$ que ele disponibiliza para serem usados pelos clientes do sistema. Cada recurso $r_j \in R_i$ é definido por um conjunto de atributos especificando suas características mais relevantes, e relacionamentos entre esses atributos.

O serviço de descoberta é invocado pelos clientes do sistema em um instante de tempo t' e recebe como entrada uma expressão ε que contém operadores lógicos de comparação $C = \{\neq, >, <, \geq\ e\ \leq\}$, e valores para os atributos e seus relacionamentos. O resultado dessa consulta é o subconjunto $M \subseteq P_{t'}$, tal que $\forall p_i \in M, \exists r_j \in R_i$ tal que a consulta ε casa com os atributos de r_j.

Tendo em vista que o sistema é aberto, o serviço precisa considerar que os provedores de recursos são intermitentes, isto é, para $t \neq t'$ é possível ter $P_t \neq P_{t'}$. Dessa forma, é importante que a responsabilidade de implementação do serviço seja distribuída entre os provedores, evitando que a saída ou a falha de um provedor possa comprometer a disponibilidade do serviço como um todo. Essa também é uma forma de distribuir o custo de implementação do serviço corroborando para a sua sustentabilidade no longo prazo.

A intermitência dos provedores também tem o efeito de tornar as informações do catálogo desatualizadas. Isso é agravado pelo fato de que, normalmente, as informações sobre os recursos são dinâmicas; por exemplo, a quantidade de memória disponível em um processador compartilhado em uma grade computacional varia ao longo do tempo. Portanto, é preciso considerar que as informações que são publicadas no catálogo podem ficar desatualizadas mesmo sem a intermitência dos provedores.

Para atender a demanda de uma ampla faixa de aplicações, um serviço de descoberta de recursos deve ser capaz de processar consultas expressivas, por exemplo, como as que podem ser expressas através de um comando em SQL [26]. Alguns tipos de consultas foram definidos por Risson e Moors [54] e elas são listadas abaixo.

- *Key lookup*: uma das formas mais simples de consulta, onde um objeto é localizado através de uma chave ou identificador com casamento exato dos valores fornecidos na consulta. Pode ser facilmente executado em ambientes altamente escaláveis de forma eficiente (como em uma DHT);

- *Keyword lookup*: são comumente utilizadas em sistemas para busca de arquivos e consiste em disseminar consultas baseadas em palavras-chave pelos *peers* da rede. Cada *peer* que recebe a consulta deve realizar uma busca em seu índice local por recursos que ele armazena, cujos nomes casam com as palavras-chave enviadas;

- *Range queries*: permite a localização de recursos através de consultas imprecisas. Utiliza-se neste caso, operadores lógicos de faixa como $>, <, \geq\ e\ \leq$. Os recursos que possuem os valores dos atributos dentro da faixa de valores especificada na consulta serão retornados;

- *Multi-attribute queries*: são baseadas em consultas por faixa mas podem utilizar um ou mais atributos;

- *Aggregation queries*: consistem em consultas para obter um agregado de dados do sistema. Geralmente são realizadas sobre uma estrutura de árvore para combinar recursivamente os resultados de um grande número de *peers*. Exemplos das operações de agregação: SUM, AVG, MAX, MIN;

- *Join queries*: consiste em unir dados que estão espalhados por diversos nós para produzir um resultado. Na prática é mais difícil de ser implementada, pois requer troca de dados entre diversos *peers* durante o processamento da consulta.

A complexidade de uma consulta pode ser definida pelo seu tipo, mas também pela quantidade de atributos, os tipos de operadores lógicos utilizados e pela quantidade de estruturas de armazenamento de dados que são consultadas numa só consulta. Segundo Triantafillou e Pitoura [61], uma consulta simples é aquela executada sobre uma única estrutura de dados e contém apenas um único atributo. Operadores de faixa podem ser incluídos. Já consultas complexas, são aquelas que: (i) contêm múltiplos atributos, (ii) são executadas sobre várias estruturas de dados e (iii) contêm operações de junção e/ou funções especiais, como as usadas em operações de agregação citadas acima.

Portanto, é desejável que a solução contemple um subconjunto de tipos de consultas, como as apresentadas por Risson e Moors, ou dê suporte a uma linguagem mais expressiva, que permita elaborar consultas complexas, combinando e correlacionando dados por exemplo. Isso possibilitaria a utilização da solução por um universo maior de aplicações.

Os recursos em si também podem assumir complexidades diferentes. Um recurso da grade computacional OurGrid por exemplo, pode ser descrito como um simples conjunto de pares atributo-valor. Seja r_1 um recurso da grade OurGrid $\in R_i$, o mesmo pode ser descrito como $r_1 = \{CPU = 2.0, Mem = 1024, Load = 0.5\}$ referindo-se à capacidade de processamento desse recurso. Portanto, um recurso do OurGrid, pode ser descrito a partir de um modelo semântico de descrição simplificado. Já um recurso do SegHidro requer um modelo de descrição mais complexo. Cada dado é composto por um conjunto de atributos, correspondentes à cobertura temporal e espacial a que ele se refere, e um conjunto de variáveis correspondentes às medições das condições do ambiente, como temperatura, velocidade do vento etc. Entretanto, os nomes dessas variáveis podem divergir de provedor para provedor. Então, para que seja possível localizar dados com nomes de variáveis diferentes mas que possuem conteúdos equivalentes, rótulos (*tags*) são atribuídos a cada variável. Dessa forma, é notável a presença de relacionamento entre as informações (atributo-variável-rótulo). Portanto, recursos complexos não podem ser descritos de forma eficiente usando um modelo baseado em pares atributo-valor, como implementado por diversos sistemas de descoberta. É necessário um modelo semântico mais expressivo, que permita correlacionar os dados so-

bre os recursos e por conseguinte, permitir que consultas mais elaboradas sejam executadas sobre esses dados.

Finalmente, o serviço precisa ter um desempenho aceitável, medido através de duas métricas: sobrecarga e cobertura. A primeira pode ser quantificada através da quantidade de informação que os processos distribuídos que implementam o serviço precisam trocar, sendo uma indicação indireta da eficiência do serviço. Quanto menor a sobrecarga, mais eficiente é a solução. A segunda é uma medida da acurácia do sistema e é calculada como sendo a relação entre a quantidade de recursos que foram efetivamente descobertos e a quantidade de recursos que deveriam ter sido descobertos. Idealmente, a cobertura deve ser de 100%.

3.0.1 Modelo Relacional

O modelo relacional que é utilizado no NodeWiz-R, foi proposto em 1970 por Edgar Frank Codd [22]. Ele é formalmente baseado na teoria matemática das relações e consiste em um método declarativo para a especificação de dados e consultas. Na visão relacional, os dados são vistos como um subconjunto do produto Cartesiano de um conjunto de domínios correspondentes a um conjunto de atributos (também chamado de "campos"), contendo "tuplas" (registros) que podem ser manipulados de acordo com certos padrões algébricos de transformação de relações [11]. Esse subconjunto representa uma relação (no sentido matemático), que é visto sobre a representação de tabela. Uma relação consiste em um cabeçalho e um corpo. Um cabeçalho é um conjunto de atributos. Um corpo de uma relação é um conjunto de n-tuplas. Uma tupla é um conjunto não-ordenado de valores de atributos que define uma ocorrência de um fato do mundo real ou de um relacionamento entre fatos. Um atributo é um par ordenado formado pelo nome do atributo e seu tipo (domínio). Toda informação é representada por valores em relações [22; 11].

Um relacionamento é uma associação entre entidades (relações) distintas. Essa associação é feita através da definição de chaves. Toda relação tem no mínimo um campo chave. Esse campo é chamado de chave primária e serve para identificar uma tupla unicamente numa relação. As chaves estrangeiras são um conjunto de campos em uma relação que é usado para fazer referência a uma tupla em outra relação. Ela deve corresponder à

chave primária da segunda relação. A existência de uma chave em uma relação pertencente a outra relação, configura o relacionamento entre essas relações.

O esquema de uma relação R, que se escreve $R(A_1, A_2, ..., A_n)$, é um conjunto de atributos $R = A_1, A_2, ..., A_n$, em que cada atributo A_i tem o nome do papel desempenhado por um domínio D no esquema R. O esquema de uma relação serve para descrever a relação em que R é o nome da relação e $A_1, A_2, ..., A_n$ são os nomes dos seus respectivos atributos [22; 11].

O modelo relacional possibilita a criação de um modelo lógico consistente da informação a ser armazenada. Este modelo lógico pode ser refinado através de um processo de normalização. Um banco de dados construído puramente baseado no modelo relacional estará portanto inteiramente normalizado. O plano de acesso, outras implementações e detalhes de operação são tratados pelo sistema gerenciador de banco de dados (SGBD) [22; 11].

Capítulo 4

NodeWiz-R: Um sistema P2P relacional para a descoberta de recursos

Este capítulo apresenta o NodeWiz-R, uma solução para a descoberta de recursos que equaciona os problemas discutidos no Capítulo 3. Para deixar mais evidente quais são as reais contribuições da solução, os requisitos a serem preenchidos pelo NodeWiz-R são descritos a seguir:

1. o sistema precisa ser autônomo (auto-gerenciável), sem a existência de um controle centralizado;

2. deve ser uma solução naturalmente distribuída, assumindo o caráter distribuído dos provedores e dos recursos, além de não adicionar pontos centralizados de falha na arquitetura;

3. deve implementar um modelo de dados expressivo, de modo a permitir relacionamentos entre as informações armazenadas e, conseqüentemente, consultas mais elaboradas;

4. deve implementar um modo de armazenamento que facilite as operações de manutenção da base de dados, e lidar bem com a intermitência e dinamicidade dos provedores e dos recursos;

5. deve implementar um mecanismo de roteamento que permita, a um baixo custo, alcançar uma alta taxa de cobertura das consultas.

O NodeWiz-R é uma extensão do sistema para descoberta de recursos NodeWiz (NW). Portanto, alguns desses requisitos já são preenchidos pelo NW e outros apenas pelo NodeWiz-R. O NodeWiz-R segue a mesma estruturação dos SBDs P2P, com uma camada relacional que atende ao requisito 3, de dispor de um modelo de descrição mais expressivo, permitindo que recursos complexos sejam descritos de forma natural, além de possibilitar que consultas ricas sejam resolvidas. Utilizando uma camada relacional, o NodeWiz-R também possibilita que consultas no padrão SQL sejam realizadas.

Uma camada P2P subjacente atende aos requisitos 1 e 2. Essa camada é implementada pelo NW. Portanto, esses requisitos já são plenamente atendidos pelo NW, sendo essas características herdadas pelo NodeWiz-R. Para atender o requisito 5, o NodeWiz-R implementa três níveis de indexação que serão descritos posteriormente, que permite localizar eficientemente até mesmo informações distribuídas e copiadas em diversos *peers* da rede. É importante saber que, diferentemente dos SBD P2P encontrados na literatura, a camada P2P implementada pelo NW utiliza um modelo estruturado baseado em árvores k-dimensionais que são capazes de rotear eficientemente consultas complexas que envolvam múltiplos atributos e faixas de valores.

Finalmente, para lidar com a intermitência e dinamicidade dos recursos e provedores requeridos no requisito 4, o NodeWiz-R utiliza uma abordagem de armazenamento de estado fraca (*soft-state*), ou armazenamento transiente, um principio chave de projeto para sistemas que operam em larga escala [21], como os sistemas P2P. A abordagem consiste no armazenamento temporário das informações sobre os recursos, por um período de tempo determinado pelo publicador. Essa abordagem também é utilizada pelo NW.

4.1 Arquitetura

Um sistema NodeWiz-R é composto por um conjunto de nós autônomos que juntamente provêem informações sobre os recursos que estão sendo compartilhados. Cada nó mantém um banco de dados leve embarcado (do inglês *embedded lightweight database*) para armazenar a parte das informações do catálogo que está sob a sua responsabilidade. O banco de dados permite que as informações sobre os recursos possam ser modeladas usando o modelo relacional, além de possibilitar uma recuperação eficiente dessas informações localmente.

Por ser baseada em uma arquitetura P2P estruturada, cada nó também mantém informações sobre um subconjunto de nós da rede que são responsáveis por outra parte das informações do catálogo. Dessa forma, operações de atualização e de consulta podem ser roteadas de forma eficiente para os nós pertinentes através da camada P2P. Cada nó do NodeWiz-R poderá ter várias tabelas de roteamento, também chamados de índices, para rotear consultas referentes a diferentes tabelas definidas no modelo relacional. Três tipos de índices distribuídos são utilizados: **índice primário**, **índices secundários** e **tabelas-índice**. Mais detalhes sobre o mecanismo de indexação distribuída do NodeWiz-R serão apresentados nas seções seguintes.

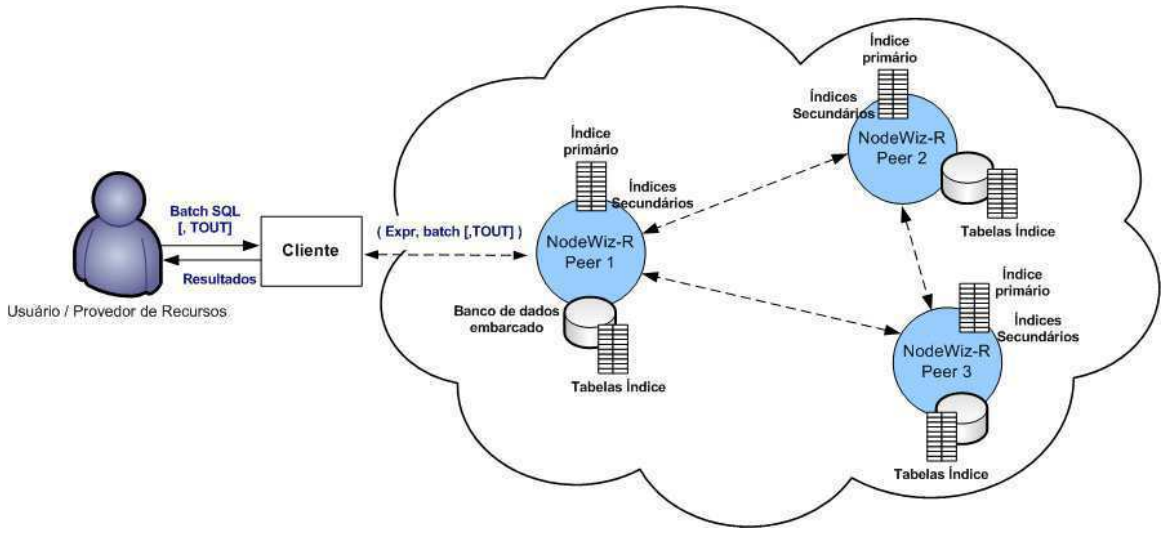


Figura 4.1: Arquitetura do NodeWiz-R em alto nível

A Figura 4.1 é uma representação gráfica em alto nível da arquitetura do NodeWiz-R. *Sites* provedores de recursos e clientes devem utilizar uma aplicação cliente para interagir com o catálogo P2P. A aplicação cliente tem por função publicar as informações sobre os recursos e enviar as requisições dos usuários para o catálogo P2P. As operações, sejam elas de atualização ou consulta, são enviadas pelo usuário em um *batch* SQL. Junto, o usuário fornece uma indicação de tempo, que chamamos de $TOUT$ e vem do termo *time out*, para determinar por quanto tempo as informações serão mantidas no catálogo. Uma expressão é gerada pela aplicação cliente e enviada junto com o *batch* SQL. O *peer* que recebe a operação verifica se ele é responsável por executá-la ou se existe algum outro *peer* que é responsável pela informação a que se refere o *batch* SQL. Essa checagem é feita comparando a expressão

com as informações mantidas pelos índices do *peer* que recebeu a operação. A operação é roteada para o *peer* responsável por armazenar as informações referentes à operação e o *batch* SQL é finalmente executado. Caso a operação seja de consulta, os resultados são retornados diretamente para a aplicação cliente, que as devolve para o usuário. Portanto, uma operação é roteada assim como em outros sistemas P2P estruturados. Entretanto, a forma como as requisições são roteadas e os tipos de índices difere o NodeWiz-R dos demais sistemas, como veremos a seguir.

4.1.1 Camada Relacional

Uma aplicação que vá usar o NodeWiz-R para descobrir a localização dos recursos compartilhados em um sistema aberto precisa primeiramente definir e tornar público o esquema do catálogo que será usado para armazenar os atributos dos recursos. Portanto, para se publicar e localizar recursos no NodeWiz-R, assume-se que o esquema de dados é conhecido, assim como nos SBDD tradicionais e outras soluções P2P como AmbientDB e PIER. O esquema deve ser definido através do modelo relacional [22], que foi escolhido para ser o modelo de descrição de dados pelas seguintes razões: (i) é um modelo claro e conciso, que não encoraja extravagâncias de *design*; (ii) é um modelo simples, que permite facilmente armazenar e recuperar fatos e informações relevantes; (iii) ao mesmo tempo em que é simples, possui uma gramática complexa para a definição de esquemas expressivos; (iv) permite gravar e relacionar grandes quantidades de dados; (v) permite a definição de múltiplos níveis de relacionamentos entre os dados; (vi) é de fácil comunicação com usuário e programadores de aplicações; (vii) é amplamente utilizado por outras aplicações, o que facilita a comunicação entre sistemas; (viii) possui uma linguagem expressiva (SQL) para a definição de consultas. Portanto, o modelo relacional agrega ao NodeWiz-R uma estrutura rica e complexa para definição de esquemas de dados e consultas.

Além das tabelas que irão compor o esquema de descrição dos recursos, deve-se especificar no esquema qual é a **tabela principal** e quais são as respectivas **tabelas secundárias**. A tabela principal é aquela que normalmente mantém os principais atributos que descrevem os recursos que estão sendo compartilhados. Já as tabelas secundárias, são aquelas que mantém informações adicionais sobre os recursos, que possuem algum relacionamento com a tabela principal, direta ou indiretamente (através de tabelas de ligação) e que se deseja

submeter consultas excepcionalmente por seus atributos. A definição da tabela principal de um esquema é obrigatória, uma vez que o índice primário, um importante indexador do sistema, baseia-se nas informações armazenadas na tabela principal para fazer o roteamento das operações. Além disso, ela é de fundamental importância na divisão do espaço de atributos do sistema, servindo como base para o particionamento horizontal das tabelas do esquema, como explicaremos mais adiante. Definir quais são as tabelas secundárias é uma tarefa opcional, caso se deseje criar índices distribuídos para indexar essas tabelas dentro do sistema, a fim de otimizar as operações sobre elas. Portanto, uma tabela é elegível para ser uma tabela secundária quando ela pode receber consultas que contém exclusivamente seus atributos. Caso alguma tabela não seja utilizada diretamente para consultas, então não se faz necessário defini-la como tabela secundária, evitando assim que um índice desnecessário seja criado e mantido pelo sistema. Tabelas de ligação, que são criadas para fazer o relacionamento entre duas ou mais tabelas do esquema por exemplo, normalmente não precisam ser indexadas, já que não é comum a realização de consultas utilizando somente campos dessas tabelas.

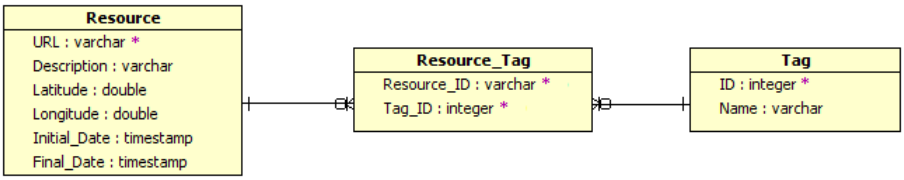


Figura 4.2: Diagrama Entidade-Relacionamento (DER) simplificado do esquema da aplicação de compartilhamento de dados ambientais do projeto SegHidro

As figuras 4.2 e 4.3 representam o modelo Entidade-Relacionamento (ER) do esquema simplificado da aplicação de compartilhamento de dados ambientais do SegHidro e o esquema descrito no padrão de definição de esquemas do NodeWiz-R. A tabela *RESOURCE* possui os principais atributos que descrevem o recurso e, portanto, é considerada a tabela principal do esquema. A tabela *TAG* é considerada uma tabela secundária, pois ela está relacionada diretamente com a tabela principal e, potencialmente, usuários podem gerar consultas contendo apenas campos dessa tabela. Já a tabela *RESOURCE_TAG* é uma tabela de ligação. Note que no final do esquema existem duas declarações, `PRIMARY TABLE` e `SECONDARY TABLE`, que determinam quais são as tabelas principal e secundárias respecti-

```
SCHEMA DEFINITION SEGHIDRO (
   CREATE TABLE RESOURCE (
        URL VARCHAR2 NOT NULL,
        Description VARCHAR2,
        Latitude Double Precision,
        Longitude Double Precision,
        Initial_Date Timestamp,
        Final_Date Timestamp,
      CONSTRAINT c1 PRIMARY KEY (URL)   );

   CREATE TABLE TAG  (
      ID NUMERIC,
      TAG_NAME VARCHAR2,
      CONSTRAINT c2 PRIMARY KEY (ID)   );

   CREATE TABLE RESOURCE_TAG  (
      RESOURCE_ID VARCHAR2,
      TAG_ID NUMERIC,
      CONSTRAINT c3 FOREIGN KEY (RESOURCE_ID) REFERENCES RESOURCE(URL),
      CONSTRAINT c4 FOREIGN KEY (TAG_ID) REFERENCES TAG (ID)   );

   PRIMARY TABLE RESOURCE,
   SECONDARY TABLE TAGS
);
```

Figura 4.3: Esquema da aplicação de compartilhamento de dados ambientais do projeto SegHidro modelado para o NodeWiz-R

vamente. A definição do esquema segue o padrão SQL, contendo algumas adaptações como no cabeçalho da definição do esquema e as declarações de tabelas principal e secundárias. O Apêndice A traz a gramática completa para definição de esquemas no NodeWiz-R.

4.1.2 Camada Peer-to-peer

A camada P2P é implementada usando uma extensão do sistema NW. O NW organiza as informações publicadas em um espaço de atributos multidimensional onde cada nó é responsável por um subespaço do espaço total de atributos do sistema (vide Figura 2.1, página 16). Sua estrutura de distribuição e indexação de informações é baseada em uma estrutura de árvore k-dimensional (*i.e.,* que é basicamente uma árvore binária de pesquisa) que determina qual nó é responsável por determinado subespaço. Dessa forma, as informações publicadas pelos provedores de recursos são distribuídas na rede de acordo com os valores de seus atributos, bem como o roteamento de consultas é feito eficientemente com base no seu conteúdo. A organização lógica do NW baseada em árvore k-dimensional lhe confere ainda um mecanismo natural para o balanceamento da carga do sistema, pois além da divisão das responsabilidades no armazenamento das informações, as consultas são roteadas e processadas apenas nos nós que potencialmente armazenam os valores de atributos requisitados, evitando assim a difusão de consultas por toda a rede.

O NodeWiz-R tira proveito das principais funcionalidades do substrato P2P NW, como

o seu mecanismo de roteamento baseado em uma estrutura de árvore k-dimensional e o balanceamento da carga do sistema, com mecanismos implícitos e explícitos para eliminação de *hot spots*. Entretanto, o atual modelo de descrição de recursos e de indexação do NW não possibilita uma indexação eficiente para recursos descritos a partir do modelo relacional adotado pelo NodeWiz-R. Isso porque, como apontado no Capítulo 2, o modelo de armazenamento do NW é bastante simplificado, baseado em uma única estrutura de dados para armazenamento, contendo pares de atributo-valor. Por isso, seu índice (tabela de roteamento) indexa apenas conteúdos dessa estrutura, e não possibilita a indexação eficiente de várias tabelas de um esquema relacional.

Para solucionar esse problema, o NodeWiz-R propõe uma extensão ao modelo de indexação do NW, criando três níveis de indexação que são apresentados a seguir.

Tipos de Índices

O NodeWiz-R implementa três tipos de índices que possibilitam a indexação eficiente das informações armazenadas nas diversas tabelas de um esquema relacional definido para o catálogo P2P. Até mesmo informações com cópias parciais na rede e que precisam ser localizadas em sua totalidade para que todos os recursos disponíveis sejam descobertos, já que estas informações podem referir-se à recursos diferentes. Os índices e suas respectivas finalidades dentro do sistema serão descritos a seguir. O seu funcionamento e como eles são criados serão apresentados na seção seguinte.

Os três tipos de índices são: o índice primário, índices secundários e tabelas-índice. Esses índices são criados e gerenciados automaticamente pelo sistema e têm por função rotear as operações para os nós adequados na rede, de acordo com as tabelas referenciadas em uma operação.

O índice primário mantém informações sobre que faixa de valores e de quais atributos um determinado nó da rede é responsável. Particularmente, o índice primário refere-se aos atributos da tabela principal. Como a tabela principal contém os principais atributos dos recursos publicados, então considera-se que a maior parte das operações irá se referir a esta tabela, o que fará com que o índice primário seja o mais utilizado para rotear as operações. Portanto, o índice primário é considerado o índice *default* do sistema, sendo sua existência obrigatória, assim como a tabela principal do esquema. O índice primário possui a mesma

estrutura de uma tabela de roteamento do NW (como exibido na Figura 2.2, página 16). Como no NodeWiz-R cada *peer* também é responsável por uma parte bem definida do espaço total de atributos do sistema, usar a tabela principal como base para o índice primário, faz com que a tabela principal seja particionada horizontalmente, como será explicado mais adiante na seção 4.2. A figura 4.7 exemplifica o índice primário referente a tabela *Resource*, que é a tabela principal do esquema, mostrada na figura 4.4.

O segundo tipo de índice é o secundário, que é criado para cada tabela secundária que se deseja indexar. Um índice secundário realiza o roteamento de operações exclusivamente por atributos de tabelas secundárias. Mas, diferente do índice primário, que irá rotear uma operação para o *peer* responsável pelo armazenamento das informações referidas numa operação, o índice secundário irá rotear uma operação referente a uma tabela secundária, para o *peer* que mantém a tabela-índice (o terceiro índice do sistema) contendo as meta-informações sobre todos os *peers* que mantém os dados referidos na operação. A tabela-índice é um nível de indireção adicionado ao sistema para permitir a indexação de informações que estão duplicadas na rede e que em um dado momento precisam ser localizadas em sua totalidade. Essa duplicação ocorre em dados de tabelas secundárias, devido ao relacionamento existente entre uma informação de tabela secundária com informações de diferentes recursos armazenados na tabela principal. Portanto, a replicação ocorre quando uma informação de tabela secundária é compartilhada por vários recursos distintos. Particularmente o índice secundário é utilizado apenas para o roteamento de uma operação que contenha, exclusivamente, campos de tabelas secundárias. O índice secundário também possui a mesma estrutura de uma tabela de roteamento do NW.

Assim como os índices secundários, as tabelas-índice são criadas para cada tabela secundária que se deseja indexar no sistema. Mas, diferente dos outros índices, que são armazenados em estruturas de dados na memória principal do computador no qual o *peer* reside, as tabelas-índice são armazenadas no próprio banco de dados de cada *peer*. Isso porque uma tabela-índice pode armazenar uma quantidade de informação substancial, um subconjunto das informações armazenadas na tabela secundária de todo o sistema. Ela possui basicamente a mesma estrutura da tabela secundária a qual ela se refere e mais um campo apontador para o *peer* que mantém aquela informação armazenada, como ilustrado na figura 4.7, na página 36 (tabela-índice referente a tabela secundária **Tag** representada na figura 4.5). A idéia da

tabela índice é servir como ponto de distribuição das operações, selecionando unicamente os *peers* que mantém as informações referidas em uma operação, redirecionando-as apenas para os *peers* que efetivamente as armazenam. A tabela-índice é a alternativa do NodeWiz-R ao uso de técnicas de inundação que geralmente são utilizadas por outras soluções para processar consultas mais complexas, de difícil roteamento.

O conceito de índices secundários e tabelas-índice foram propostos uma vez que as informações das tabelas secundárias podem com frequência ser copiadas em vários *peers*. Dessa forma, quando, por exemplo, é enviada uma consulta referente a uma tabela secundária que contém valores distribuídos por diversos *peers*, a tabela-índice fará o roteamento de forma eficiente somente para os *peers* responsáveis pelas informações requisitadas, sem que saltos adicionais na rede sejam necessários. Portanto, os índices possuem funcionamentos diferentes e são utilizados em ocasiões distintas: em operações que envolvam a tabela principal, apenas o índice primário será utilizado no roteamento; já em operações que contêm exclusivamente atributos das tabelas secundárias, o roteamento será feito pelos índices secundários e tabelas-índice.

O uso de vários níveis de indexação no NodeWiz-R, apesar de permitir uma indexação praticamente completa das informações do sistema, eleva o custo de manutenção das informações internas do sistemas, uma vez que mais meta-informações precisam ser mantidas por cada *peer*. Este é o dilema existente entre as arquiteturas P2P estruturadas e não-estruturadas. Mas neste caso, esse custo é justificado pelo fato de permitir que consultas por diversas tabelas do sistema sejam roteadas eficientemente pela rede, sem o uso de técnicas de inundação (como será descrito em mais detalhes na seção 4.2.7).

4.2 Funcionamento do sistema

No NodeWiz-R, assim como no NW, cada *peer* é responsável por um subconjunto do espaço de atributos total do sistema. A atribuição das responsabilidades sobre que informações cada *peer* deve armazenar, é feita a medida que novos *peers* entram no sistema. Quando um novo *peer* deseja ingressar na rede, ele contata algum *peer* conhecido. O sistema então detectará qual é o *peer* mais sobrecarregado da rede, para que seja feita o balanceamento de sua carga de trabalho. Essa informação é reunida pelo algoritmo distribuído **Top-K** [9], que ordena

os nós baseado num indicador da carga de trabalho (*workload*) mantido por cada *peer*. Esse indicador é incrementado a cada consulta ou atualização recebida. Dessa forma, o *peer* que contém mais dados e que está recebendo um maior número de consultas é escolhido para dividir sua carga de trabalho e seu subespaço de atributos com o novo *peer*.

Após ter sido identificado o *peer* mais sobrecarregado da rede, o algoritmo de divisão (*Splitting Algorithm*) é executado para dividir o subespaço de atributos, utilizando um dos atributos da tabela principal. O algoritmo determinará qual atributo e valor de corte garantirão uma boa divisão da carga de trabalho e garantirá que o crescimento do sistema seja feito de forma balanceada (os detalhes da implementação dos algoritmos *Top-K* e *Splitting* são descritos na seção 5.1.2 do capítulo 5). É também nesse momento que ocorre o particionamento (ou fragmentação) horizontal das tabelas do esquema.

A fragmentação horizontal de uma tabela ocorre através da movimentação de tuplas para uma localização diferente (banco de dados em um nó diferente) baseado em um predicado, formado por um atributo e um valor base [49]. No NodeWiz-R a escolha do predicado é feita com base nos atributos da tabela principal. Quando escolhemos um atributo da tabela principal para compor o predicado que será usado como base do particionamento, garantimos que as informações sobre um determinado recurso estarão disponíveis em um só *peer* da rede. Isso porque uma tupla da tabela principal é única, ou seja, refere-se unicamente a um recurso. Ao contrário dos dados armazenados nas tabelas secundárias, que podem não ser exclusivos, ou seja, estar relacionados com diversos dados de outras tabelas. Portanto, ao se usar dados da tabela principal como base para a movimentação das informações do sistema entre os *peers*, garantimos que as informações referentes a um determinado recurso estará disponível apenas em um único *peer* da rede. Os dados relacionados serão movimentados ou copiados juntamente, e por isso podem aparecer replicados na rede. Isso ocorre pois as informações das tabelas secundárias podem estar relacionadas com várias tuplas da tabela principal. Portanto, quando uma informação relaciona-se com outras n informações da tabela principal, ela é copiada juntamente com a tupla que está sendo movida e é armazenada no mesmo *peer*. O esquema utilizado no *peer* que dividiu o seu subespaço com o novo *peer*, é copiado e instanciado no banco de dados do novo *peer*, e os novos dados são portanto armazenados. É importante notar que são movidas para o devido *peer* apenas as informações sobre os recursos que casam com o predicado definido. Essa divisão é melhor exemplificada

a seguir.

As figuras 4.4, 4.5 e 4.6 ilustram a divisão das informações do *peer* A com o *peer* B. Por simplificação, o esquema do exemplo é composto apenas por três tabelas. A tabela *RESOURCE* é a tabela principal do esquema. A tabela *TAG* é uma tabela secundária. As tabelas *RESOURCE* e *TAG* possuem um relacionamento do tipo N x N. Portanto, um recurso R_1 em *RESOURCE* pode estar relacionado com zero ou mais tuplas da tabela *TAG*, assim como uma *tag* T_1 em *TAG* pode estar relacionada com vários recursos em *RESOURCE*. A tabela *RESOURCE_TAG* é a tabela de ligação que faz o relacionamento entre as duas tabelas. Na figura 4.4, pode-se observar na tabela *RESOURCE_TAG* que a *tag* "$Wind$", que possui $ID = 0$ está associada a dois recursos distintos da tabela *RESOURCE*, cujos ID's são 0 e 1 respectivamente. Considera-se que as informações estão inicialmente armazenados no *peer* A no instante de tempo T_1 e que ele é o único *peer* do sistema.

Quando o *peer* B entra no sistema, ele vai se juntar a rede através do *peer* A, pois ele é o único participante. Assumindo que o atributo escolhido para fazer a divisão do espaço de atributos é o atributo $Att1$ da tabela principal e o valor base escolhido é 15, ao fazer o particionamento, os recursos cujo o atributo $Att1$ tem valor maior do que 15 devem migrar para o *peer* B. Como a *tag* "$Wind$" está associada ao recurso cujo o valor de $Att1$ é igual a 16, então a tupla referente a essa *tag* é **copiada** e a tupla na tabela *RESOURCE_TAG*, que faz a ligação da *tag* com a tupla na tabela principal, **migra** para o *peer* B (figura 4.6). Portanto, a *tag* "$Wind$" passa a existir em dois *peers* diferentes no instante de tempo T_2, nos *peers* A e B respectivamente, já que a mesma *tag* está associada a dois recursos distintos cujas as suas informações estão armazenados em dois *peers* diferentes no instante T_2. A figura 4.5 mostra quais informações permanecem armazenadas no *peer* A após o instante de tempo T_2.

Resource

ID	Att1	Att2
0	15	35
1	16	42

Tag

ID	Tag Name
0	Wind
1	Rain

Resource_Tag

Resource_ID	Tag_ID
0	0
0	1
1	0

Figura 4.4: Informações armazenadas no *peer* A no instante de tempo T_1

Resource		
ID	**Att1**	**Att2**
0	15	35

Tag	
ID	**Tag Name**
0	Wind
1	Rain

Resource_Tag	
Resource_ID	**Tag_ID**
0	0
0	1

Figura 4.5: Informações armazenadas no *peer* A no instante de tempo T_2, após particionamento

Resource		
ID	**Att1**	**Att2**
1	16	42

Tag	
ID	**Tag Name**
0	Wind

Resource_Tag	
Resource_ID	**Tag_ID**
1	0

Figura 4.6: Informações armazenadas no *peer* B no instante de tempo T_2, após particionamento do *peer* A

Índice Primário

Level	Attr	Min	Max	Peer Address
0	Att1	15	+inf	*PeerB*

Índice Secundário

Level	Attr	Min	Max	Peer Address
0	Tag Name	Rain	+inf	*PeerB*

Tabela Índice

ID	Tag Name
1	Rain

Peer
PeerA

Figura 4.7: Índices do *peer* A no instante de tempo T_2, após divisão do seu espaço de atributos

Índice Primário

Level	Attr	Min	Max	Peer Address
0	Att1	-inf	15	*PeerA*

Índice Secundário

Level	Attr	Min	Max	Peer Address
0	Tag Name	-inf	Rain	*PeerA*

Tabela Índice

ID	Tag Name
0	Wind

Peer
PeerA
PeerB

Figura 4.8: Índices do *peer* B no instante de tempo T_2, após receber as informações do *peer* A

Após o particionamento, o novo *peer* integrante recebe, além da definição do esquema e parte dos dados, uma cópia dos índices primários e secundários do *peer* com o qual dividiu o seu espaço de atributos e ambos adicionam uma entrada no final de seus índices apontando um para o outro. Dessa forma, as operações que são enviadas para um desses *peers* podem ser encaminhadas para seus *peers* antecessores e sucessores. As figuras 4.7 e 4.8 mostram como seriam os índices dos *peers* A e B após a divisão do espaço de atributos do *peer* A. Como inicialmente o *peer* A era o único do sistema, seus índices permaneciam vazios no tempo T_1, até a entrada do novo *peer*. No tempo T_2, após o particionamento das informações e cópia dos índices para o *peer* B, as informações foram adicionadas nos índices de ambos os *peers*, completando assim o procedimento de divisão dos subespaços entre os dois *peers*.

As tabelas-índice entretanto, recebem um tratamento diferente dos índices primário e secundários. Como as tabelas-índice são armazenadas no próprio banco de dados local do *peer*, ela também é fragmentada juntamente com os dados sobre os recursos que estão lá armazenados. No caso da divisão do *peer* A com o *peer* B, a tabela-índice referente a tabela secundária TAG no *peer* A é particionada. A divisão da tabela-índice ocorre como descrito a seguir.

O sistema verifica primeiramente quais são as tabelas-índice do esquema através de um descritor de esquema (*schema-descriptor*). O algoritmo de *Splitting* é executado sobre o histograma mantido para cada tabela-índice existente e determina quais predicados serão utilizados como base para o particionamento de cada tabela-índice, assim como é feito com a tabela principal. Dentre os campos da tabela-índice, apenas o campo apontador não pode ser utilizado para compor o predicado. Escolhido o predicado e valor base para o particionamento, as informações contidas na tabela-índice são **movidas** para o novo *peer*, de acordo com o predicado definido. As informações sobre o predicado que determinou a divisão da tabela índice são portanto adicionadas no final do *índice secundário* correspondente à tabela-índice, em ambos os *peers*, assim como ocorre com o índice primário. Portanto, a tabela-índice é particionada e o índice secundário mantém a informação sobre quais subespaços de informações sobre as tabelas secundárias do esquema, as tabelas-índice de alguns *peers* mantêm.

Exemplificando com a divisão da tabela-índice referente à tabela secundária TAG do exemplo anterior. No instante T_1 a tabela-índice referente à tabela secundária TAG no *peer* A contém as mesmas informações da tabela secundária TAG, incluindo os apontadores para o *peer* A, indicando que A era o *peer* do sistema que mantinha aquelas informações. O algoritmo de *Splitting* determina que predicado será usado como base para o particionamento da tabela-índice. Como exibido nos índices secundários das figuras 4.7 e 4.8, o predicado escolhido é composto pelo campo *Tag_Name* e pelo valor base "$Rain$". Essas informações são adicionadas no final do índice secundário correspondente aquela tabela-índice, em ambos os *peers*. Podemos observar que o índice secundário do *peer* A, indica que o *peer* B é responsável pelas informações de localização das *tags* cujo nome é maior que "$Rain$". Como para o sistema a *tag* "$Wind$" é maior que "$Rain$", as informações sobre onde ela está localizada na rede devem ficar armazenadas na tabela-índice do *peer* B, como podemos

observar na figura 4.8.

Note também que, como a *tag* "$Wind$" está armazenada em ambos os *peers*, o *peer* B mantém dois apontadores na sua tabela-índice referente a essa *tag*. Assim, consultas pela *tag* "$Wind$" serão roteadas para ambos os *peers* que devolverão como resultado os respectivos recursos associados com a *tag*.

É importante ressaltar que, mantendo os dados e tabelas-índice fragmentadas em diversos *peers* da rede, o NodeWiz-R garante que mecanismo de balanceamento da carga do sistema não será afetado pelo novo modelo de indexação. O particionamento das tabelas-índice também é fundamental para a eliminação de possíveis *hot spots* em consultas por tabelas secundárias, já que o não-particionamento das tabelas-índices causaria um acúmulo de informações em um determinado *peer* da rede. Ele então passaria a responder por todas as consultas referentes à tabela secundária à qual aquela tabela-índice se refere, sobrecarregando o *peer*, e consequentemente aumentando o tempo de resposta das consultas.

4.2.1 Árvore K-Dimensional

No NodeWiz-R, conforme vão sendo feitas as divisões dos subespaços entre os *peers*, a topologia da rede assume a estrutura lógica de uma árvore binária de busca k-dimensional (*kd-tree*). A figura 4.9 mostra a formação da árvore de distribuição do NodeWiz-R após a entrada do *peer* B no sistema. Inicialmente, apenas o *peer* A faz parte da rede. Com a entrada do *peer* B, a estrutura de árvore começa a se formar, com a divisão do espaço de atributos do *peer* A. Os nós não-folha da árvore são nomeados com o nome de ambos os *peers*. Os nós folha representam os *peers* do sistema. Todas as operações de atualização e consultas cujo atributo *Att1* é maior que 15 serão roteadas para o *peer* B, enquanto que os com valores menores ou iguais a 15 serão roteados para o *peer* A. Caso uma consulta seja feita na tabela secundária *TAG*, ela será roteada baseada no atributo *Tag_Name*, onde as consultas por *tags* maiores que "$Rain$" serão roteadas pela tabela-índice do *peer* B, enquanto que as demais serão roteadas pela tabela-índice do *peer* A. A figura 4.10 mostra como ficaria a árvore de distribuição após a entrada de novos *peers*. Essa estrutura confere ao NodeWiz-R uma eficiente indexação das informações e roteamento de consultas por faixa de valores e com múltiplos atributos.

Uma árvore de roteamento bem formada garante ao sistema que o roteamento de uma

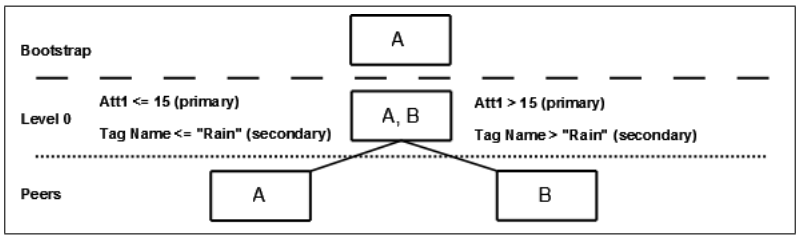


Figura 4.9: Formação da árvore de distribuição do NodeWiz-R após a divisão do espaço de atributos do Peer A

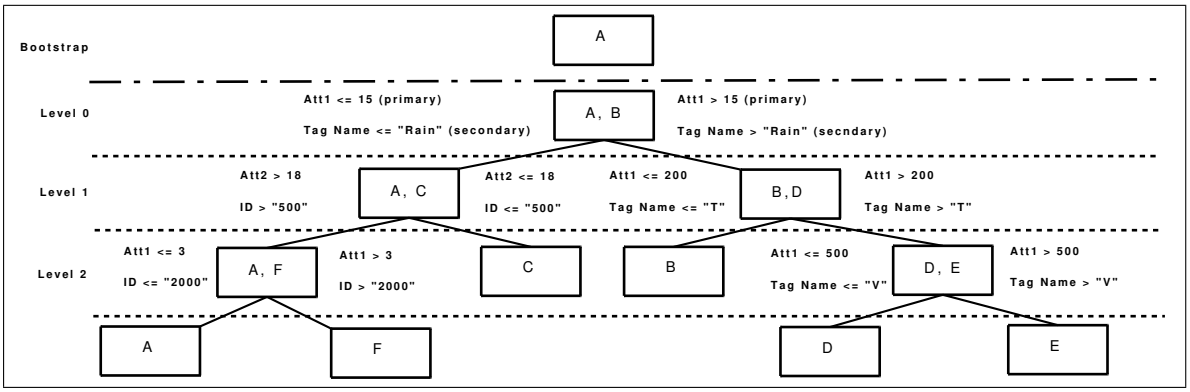


Figura 4.10: Árvore de distribuição do NodeWiz-R após a entrada de outros peers

consulta pelo índice primário é feita em no máximo $log_2 N$ *hops*, onde N é o número de *peers* da rede, e no mínimo com zero *hops*, quando o primeiro *peer* que recebe a consulta é o responsável pelas informações requisitadas. Para uma consulta roteada pelo índice secundário, o roteamento é feito em até $log_2 N + 1$ *hops*, já que uma vez encontrada a tabela-índice, um *hop* a mais é necessário para se alcançar os *peers* que armazenam as informações requisitadas. O mínimo nesse caso também é zero *hops*, quando o primeiro *peer* que recebe a consulta mantém a tabela-índice e ele mesmo possui os dados requisitados.

4.2.2 Saída voluntária de um peer do sistema

Embora seja feito o balanceamento da carga dos *peers* com a divisão dos subespaços entre eles, um *peer* pode ainda receber uma quantidade de consultas maior que o esperado, uma vez que o padrão de consultas pode mudar com o tempo. Caso isso ocorra, o NodeWiz-R pode ainda lançar mão da técnica de *leave* e *rejoin*, implementada pelo NW. A técnica consiste na saída voluntária de um *peer* do sistema, caso o *workload* sobre aquele *peer* alcance um determinado limiar pré-estabelecido. O *peer* então deixa o sistema e seu espaço de atri-

butos passa a ser de responsabilidade do *peer* através do qual ele ingressou no sistema. O *peer* que deixou o sistema posteriormente entra novamente e divide o espaço de atributos com o *peer* mais sobrecarregado da rede naquele momento.

4.2.3 Manutenção dos índices

A manutenção dos índices do NodeWiz-R é feita de forma automática pelo próprio sistema, mas de forma diferente para os tipos de índices. Os índices primário e secundários são atualizados apenas quando ocorre a entrada ou saída de um *peer* da rede. No caso da entrada de um novo *peer* no sistema, os índices são copiados para o novo *peer* e uma entrada é adicionada no final de cada índice, como foi mencionado na seção 4.2. No caso da saída de um *peer* da rede, o *peer* que dividiu o seu espaço de atributos com o *peer* que está deixando o sistema é avisado e a entrada nos seus índices apontando para aquele *peer* é retirada. O mesmo ocorre com os *peers* que dividem seu espaço de atributos com o *peer* que está saindo do sistema.

Já a atualização das tabelas-índice ocorre da seguinte forma: sempre que uma nova informação é adicionada em uma tabela secundária em qualquer *peer* da rede, sua respectiva tabela-índice deve ser atualizada. O índice secundário roteia uma operação interna de atualização de tabela-índice, que é gerada automaticamente, para os *peers* que são responsáveis pelas tabelas-índice, e a atualização é realizada. Dessa forma, as tabelas-índice estão sempre sincronizadas em relação às novas informações publicadas no catálogo.

É importante ressaltar que, embora seja possível manter as tabelas-índices sempre atualizadas, existe um custo associado, uma vez que o número de atualizações nas tabelas-índice aumenta na mesma proporção que o número de atualizações das suas respectivas tabelas secundárias. Portanto, se uma tabela secundária é atualizada frequentemente, operações de atualização também serão disparadas pelo sistema para a respectiva tabela-índice, com a mesma freqüência, o que pode elevar o tráfego de mensagens na rede.

4.2.4 Armazenamento das informações

O NodeWiz-R utiliza a abordagem de atualização de estado fraca (*soft-state*) para facilitar a manutenção das informações no sistema. Nessa abordagem, as informações sobre os re-

cursos são armazenadas temporariamente no sistema. Uma estampilha de tempo ($TOUT$) é associada a cada tupla armazenada nas tabelas do banco de dados. O $TOUT$ determina por quanto tempo a tupla deve permanecer no sistema. Quando o $TOUT$ expira, a tupla é automaticamente removida. Assim, as informações sobre os recursos devem ser publicadas periodicamente no sistema pelos provedores, mantendo-as atualizadas com relação ao estado atual do recurso. O $TOUT$ é determinado por quem está publicando as informações.

A remoção de informações desatualizadas portanto é feita sem que esforços administrativos sejam requeridos. O mecanismo atua como um coletor de lixo automático, onde as informações cujo o tempo de vida expiraram são removidas automaticamente.

O $TOUT$ também é utilizado nas tabelas-índices para garantir a consistência entre as informações armazenadas nos *peers* e os apontadores referentes a essas informações contidas nas tabelas-índice. O mesmo $TOUT$ estabelecido para uma tupla de tabela secundária, será atribuído a seu apontador na respectiva tabela-índice. Dessa forma, quanto o $TOUT$ de uma tupla de tabela secundária expira, a tupla é removida juntamente com o seu apontador na respectiva tabela-índice. Portanto, a abordagem *soft-state* também torna-se importante na manutenção das tabelas-índices, fazendo a remoção das informações que encontram-se desatualizadas, mantendo-as em um estado consistente.

Essa abordagem se mostra adequada para os casos em que as informações sobre os recursos são dinâmicas e intermitentes. Não obstante, ela requer maior trabalho do lado publicador. Em contra partida, essa abordagem propicia o incremento da disponibilidade das informações sobre os recursos em caso de falha do *peer* mantenedor, pois as informações são republicados rapidamente no sistema pelo provedor de recursos (publicador).

É importante notar também que, com o uso dessa abordagem, há um relaxamento das propriedades ACID requeridas nos SBD tradicionais. Isso porque o NodeWiz-R não garante a durabilidade dos dados, uma vez que as informações sobre os recursos são removidas após o tempo determinado pelo $TOUT$. Entretanto, o suporte às propriedades ACID em sistemas P2P onde os nós são fracamente acoplados ainda é um problema em aberto devido às dificuldades impostas pela própria arquitetura, como também por não se tratar de um requisito essencial no contexto dos sistemas para descoberta de recursos [13].

4.2.5 Formato das operações internas

Uma operação de atualização no NW é representada por um conjunto de pares formados por $< atributo, valor >$. Já uma operação de consulta é uma tripla formada por $< atributo, operador, valor >$. O NodeWiz-R amplia a definição de operações para se adequar ao modelo relacional, e ao novo mecanismo de indexação. Cada operação tem o seguinte formato:

$$OP ::= (Expr, batch[, TOUT]),$$

onde:

- *Expr*: é a expressão usada para o roteamento da operação;
- *batch*: é um conjunto de instruções SQL de atualização ou consulta;
- *TOUT*: é o tempo que as informações deverão ser mantidas no índice P2P (apenas para operações de atualização);

A expressão gerada é uma simplificação do *batch* SQL fornecido. Ela é formada por um conjunto de restrições do tipo $< atributo, operador, valor >$, assim como no NW, e é utilizada pela camada P2P para rotear a operação para os nós que são responsáveis por manter as partes do catálogo que casam com as restrições especificadas na expressão (mais detalhes sobre como uma expressão é gerada são descritos no Capítulo 5, na seção 5.1.2). O *batch* contém as instruções de atualização ou uma consulta, ambas expressas em SQL. O SQL é executado apenas no banco de dados do *peer* que é responsável pela informação contida ou requerida no *batch*. O $TOUT$ é utilizado nas operações de atualização e é associado a cada instrução de atualização contida no *batch*. Dessa forma, todas as informações terão o mesmo tempo de vida dentro do sistema, mantendo as tabelas e suas referências em um estado consistente.

4.2.6 Publicando informações no NodeWiz-R

Uma informação é publicada no NodeWiz-R através do envio de um *batch* SQL contendo instruções de atualização e o respectivo $TOUT$. O *batch* de atualização é tipicamente um

conjunto de instruções SQL do tipo $INSERT$ ou $UPDATE$. O $TOUT$ é especificado para todo o *batch*, ou seja, ele será atribuído à todas as instruções SQL contidas no *batch*.

Caso o *batch* SQL possua apenas instruções de atualização para a tabela principal, a expressão será gerada contendo apenas atributos da tabela principal e a operação será roteada pelo índice primário. Se o *batch* contém atualizações para a tabela principal, tabelas secundárias e tabelas de ligação relacionadas, a expressão também terá somente atributos da tabela principal e será roteada pelo índice primário. Todas as instruções serão executadas no devido *peer* destino. Mas, se o *batch* SQL contém apenas instruções de atualização para tabelas secundárias, será gerada uma expressão contendo apenas atributos da tabela secundária. Nesse caso a operação será roteada pelo índice secundário. Após ser roteado até o *peer* responsável pela tabela-índice, o *batch* SQL é então executado no seu banco de dados local. Caso a informação já exista no banco, ela é apenas atualizada, e um novo $TOUT$ é atribuído à informação existente.

Após as execuções das instruções SQL de atualização de tabela secundária, uma operação de atualização nas tabelas-índice referente à tabela secundária é criada, roteada para os *peers* responsáveis por mantê-la e executada, mantendo as tabelas-índice atualizadas em relação a nova informação adicionada no sistema.

4.2.7 Realizando consultas no NodeWiz-R

Uma operação de consulta no NodeWiz-R requer apenas o envio de um *batch* SQL contendo uma instrução do tipo $SELECT$. Quando uma consulta que contém atributos da tabela principal é submetida por um usuário, os seguintes procedimentos são realizados:

1. A aplicação cliente recebe o *batch* SQL e faz a validação, certificando-se que a consulta submetida refere-se ao esquema utilizado;

2. Uma expressão de roteamento é gerada baseada no SQL e a operação contendo o *batch* e a expressão é enviada para o *peer* conhecido;

3. O *peer* que recebe a consulta verifica se ele é responsável pelo subespaço de informações requeridas na consulta, através da expressão dada e informações internas de estado mantidas por ele. Caso haja o casamento do seu subespaço de atributos

com a expressão, ele processará a consulta e os resultados encontrados serão enviados diretamente para o cliente.

4. O *peer* verifica no índice se existem outros *peers* que casam com a expressão. A operação é roteada para os *peers* em que há o casamento, para que também possa ser processada por eles. Os procedimentos 3 e 4 são executados por cada *peer* que recebe a operação após o roteamento, até que não existam mais *peers* que casem com a expressão ou o último nível (campo *level*) do índice seja alcançado.

A figura 4.11 ilustra o processamento pelo NodeWiz-R de uma operação de consulta contendo atributos da tabela principal. No caso de consultas referentes à tabela principal, elas são roteadas pelo índice primário e processadas em todos os *peers* em que há um casamento do seu subespaço de atributos com a expressão gerada. Os *peers* A e B fazem o roteamento e processamento da consulta e os resultados encontrados são imediatamente enviados para o cliente. Já o *peer* C apenas processa a operação recebida e retorna os resultados, mas não faz o roteamento da operação.

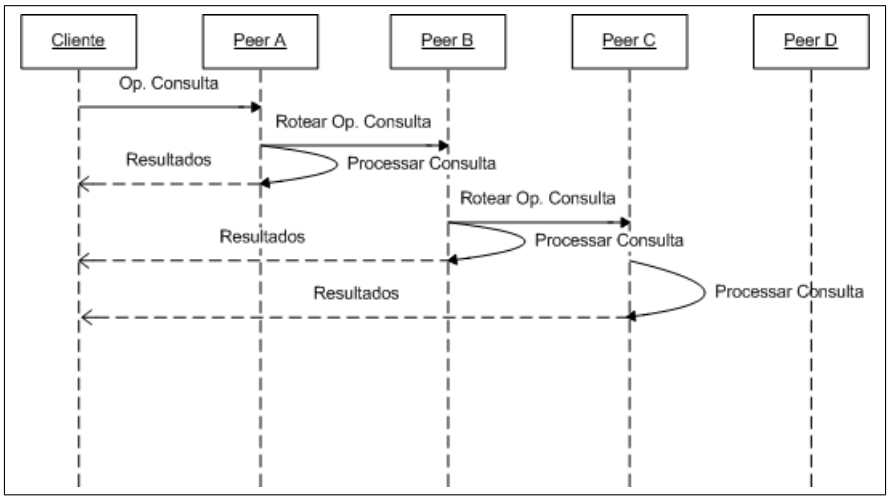


Figura 4.11: Processamento de uma consulta com atributos da tabela principal (Roteamento de uma operação de consulta)

Já as consultas referentes às tabelas secundárias possuem um tratamento diferente, e são realizadas em duas fases, como é descrito abaixo:

Fase I

A primeira fase consiste em rotear a operação até o *peer* que mantém a tabela-índice referente à tabela secundária referenciada na consulta. Os seguintes procedimentos são realizados:

1. A aplicação cliente recebe o *batch* SQL e faz a validação, certificando-se que a consulta submetida refere-se ao esquema utilizado;

2. Uma expressão de roteamento é gerada baseada no SQL e a operação contendo o *batch* e a expressão é enviada para o *peer* conhecido;

3. O *peer* que recebe a consulta verifica se ele é responsável pela tabela-índice referente às informações requeridas na consulta, através da expressão e informações internas de estado mantidas por ele. Caso ele seja o responsável, a segunda fase do processamento da consulta será realizada. Caso ele não seja o responsável, o *peer* verifica no índice secundário referente à tabela secundária consultada, se existem outros *peers* que casam com a expressão. A operação é roteada para os *peers* que casam com a expressão. O procedimento 3 é executado por cada *peer* que recebe a operação após o roteamento.

Fase II

Na segunda fase, a operação é roteada pelo *peer* que mantém a tabela-índice, até os *peers* que de fato mantêm os dados requisitados. Os seguintes procedimentos são realizados:

1. O *peer* que detém a tabela-índice requerida, gera uma consulta interna no padrão SQL que é executada sobre a tabela-índice. O resultado são os endereços dos *peers* que mantém as informações requisitadas no SQL.

2. A operação é portanto roteada para cada um dos *peers* apontados pela tabela-índice.

3. Os *peers* que recebem a operação executam o *batch* SQL no banco de dados local e os resultados são retornados diretamente para o cliente. Nenhum roteamento extra é realizado após o processamento da consulta.

A figura 4.12 ilustra o processamento pelo NodeWiz-R de uma operação de consulta contendo apenas atributos de uma tabela secundária. O *peer* A faz o roteamento da operação

através do índice secundário, para o *peer* B que mantém a *tabela-índice* referente a tabela secundária consultada. Essa consiste na primeira fase do processamento. O *peer* B inicia a segunda fase consultando a tabela-índice no seu banco de dados local e posteriormente, faz o roteamento para os *peers* que casaram com a consulta interna que foi gerada. Os *peers* C, D e E que casaram com a consulta, recebem portando a operação para realização do processamento e os resultados encontrados são enviados para o cliente.

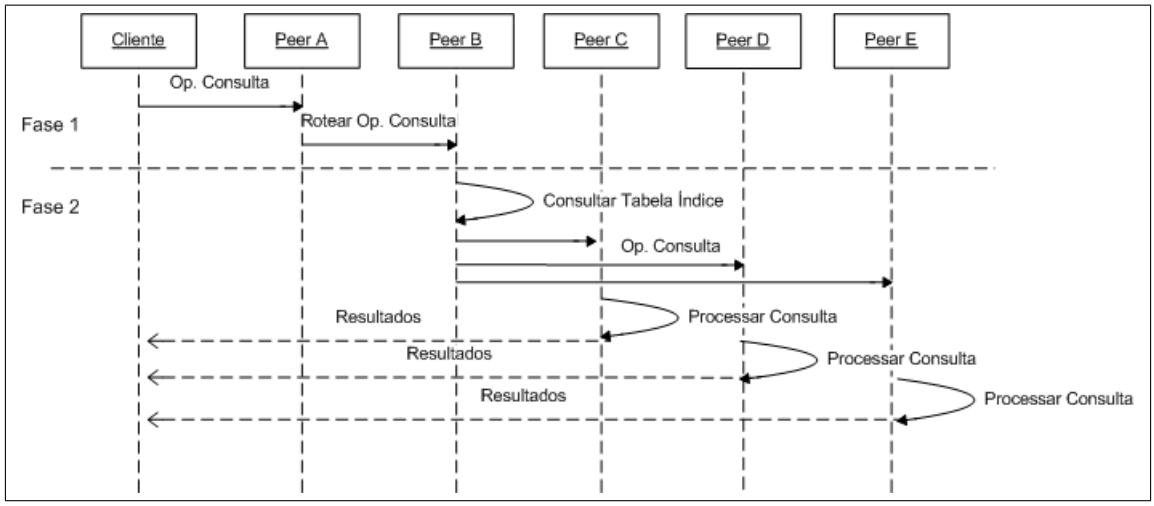


Figura 4.12: Processamento de uma consulta com atributos de uma tabela secundária

No caso de uma consulta onde várias tabelas secundárias estão envolvidas, mais de uma expressão de roteamento pode ser gerada, uma para cada uma das tabelas referenciadas. Com isso, mais de um índice secundário será usado para o roteamento da operação e várias tabelas-índice poderão ser consultadas. As várias tabelas-índice consultadas, por sua vez, rotearão a operação para diversos *peers* ao mesmo tempo. Esse comportamento pode fazer com que um mesmo *peer* seja consultado várias vezes, gerando resultados repetidos para o usuário. Para solucionar esse problema, cada *peer* mantém um *cache* LRU (*Least-Recently Used*) que armazena os códigos *hash* das consultas mais recentes respondidas. O funcionamento dos cache LRU é descrito a seguir.

Quando uma consulta roteada através de uma tabela-índice chega em um *peer*, o seu código *hash* é gerado. Novos códigos *hash* são adicionados no início do *cache*. Quando o limite de tamanho do *cache* é excedido, o último código *hash* é eliminado, permitindo que um novo código seja adicionado no topo do cache. Caso uma consulta chegue no *peer* e seu *hash* já conste no *cache*, significa que ela já foi respondida. Dessa forma, o *peer* não

responderá mais aquela consulta. Entretanto, caso o *peer* deixe de ser consultado por muito tempo, os códigos *hash* das últimas consultas podem permanecer armazenados por um longo período, e impedir que consultas sejam respondidas mesmo tendo passado um grande espaço de tempo entre a sua última e uma nova submissão.

Para solucionar esse problema, um *timeout* é estabelecido e atribuído a cada código *hash* no *cache*. Quando uma consulta chega e o seu código *hash* se encontra no *cache*, o *timeout* do *hash* é verificado. Caso ele tenha expirado, ele é movido para o topo do *cache* com o *timeout* renovado e a consulta pode ser respondida normalmente. Caso o *hash* não tenha expirado, a consulta não é respondida e é imediatamente descartada. Isso garante que uma mesma consulta não seja respondida diversas vezes, caso ela seja submetida repetidamente em um curto espaço de tempo, mas que possa ser re-submetida depois de um período de tempo mais longo e os resultados sejam obtidos novamente.

Consultas com operações de junção e agregação

Uma das principais preocupações dos processadores de consultas distribuídos está em prover suporte eficiente a operações de junção e agregação, geralmente necessárias no processamento distribuído de cláusulas SQL [38]. Isso porque, além de não ser uma tarefa trivial em ambientes amplamente distribuídos, o custo para a realização dessas operações é muito elevado devido a considerável quantidade de dados que necessita ser transferida para a realização de um processamento (*high-throughput*). Apesar de existirem técnicas eficientes para o processamento desse tipo de consulta (*e.g.,* Bloomjoins [43], Hash Join [29]), elas ainda apresentam altas taxas de transferência de dados no processamento das consultas [49], gerando um *overhead* indesejado em sistemas que operam na escala da Internet.

No NodeWiz-R, não se faz necessário lidar com junções distribuídas. Isso é possível pois todas as informações que estão relacionadas de alguma forma, estão presentes no mesmo *peer*. Um exemplo de como isso realmente ocorre, foi dado na seção 4.2, onde a *tag* "Wind" que estava relacionada a dois recursos distintos, também estava presente nos dois *peers* onde estão armazenadas informações com as quais ela possui relacionamento. O custo associado com essa solução está principalmente na replicação das informações e manutenção dos índices para identificar onde elas estão dispostas na rede. Entretanto, elimina-se o custo bastante elevado do processamento de junções distribuídas.

Operações de agregação também são executadas apenas localmente. Entretanto, um problema ocorre neste caso, pois não é possível realizar uma operação de agregação envolvendo dados armazenados em vários *peers*. Dessa forma, o NodeWiz-R deixa para o cliente a responsabilidade de filtrar os resultados recebidos dos diversos *peers*. Uma possível solução para o problema é discutida na seção 5.1.3 do Capítulo 5.

4.3 Conclusão

Neste capítulo, apresentamos a arquitetura e o funcionamento do NodeWiz-R, uma solução distribuída que se apoia no modelo consagrado para o desenvolvimento sistemas distribuídos autônomos, a arquitetura P2P, e no modelo de representação de dados amplamente utilizado pelos sistemas de bancos de dados, o modelo relacional, para permitir que recursos com semântica de descrição complexa sejam descritos naturalmente, e que consultas mais ricas e expressivas sejam executadas pelo sistema.

O NodeWiz-R, que é baseado no sistema de informação de grade NW, amplia o seu modelo de indexação para permitir que diversas tabelas que podem compor um esquema de descrição de recursos, possam ser indexadas eficientemente na rede, sem que técnicas de inundação sejam utilizadas.

Diferente de outras soluções apresentadas, o seu modelo de indexação é baseado em uma estrutura de árvore de busca *k*-dimensional (*kd-tree*), que é formada à medida que novos *peers* entram no sistema. Essa estrutura permite que as operações sejam roteadas eficientemente para os *peers* que são responsáveis pelas informações referidas na operação. Além disso, o NodeWiz-R adiciona o suporte a um subconjunto da linguagem SQL[1], permitindo que consultas mais elaboradas sejam realizadas.

No NodeWiz-R, cada *peer* é responsável por um subespaço do espaço total de atributos do sistema. Isso permite que cada *peer* responda apenas por parte das informações do sistema, dividindo a responsabilidade com os demais. O NodeWiz-R utiliza ainda uma abordagem baseada no armazenamento temporário das informações sobre os recursos, de forma que a remoção de informações desatualizadas sejam feitas sem que esforços administrati-

[1] Alguns operadores utilizados na cláusula $WHERE$, como $LIKE$ e $BETWEEN$ não são suportados pela implementação atual, mas que não significa uma limitação da solução em si

vos sejam requeridos. O mecanismo atua como um coletor de lixo automático, onde as informações cujo o tempo de vida expiraram, são removidas automaticamente. Nessa abordagem, as informações devem ser publicadas periodicamente pelos provedores de recursos. Dessa forma, o catálogo se mantém sempre atualizado com as informações mais recentes sobre os recursos que estão sendo disponibilizados. Essa abordagem tem se mostrado adequada para casos onde os recursos são dinâmicos e intermitentes, que exigem constante atualização.

Capítulo 5

Implementação do protótipo

Neste capítulo são apresentados os detalhes da implementação do protótipo do NodeWiz-R. São apresentados diversos aspectos que julgamos mais importantes que não são contemplados pelo projeto arquitetural e portanto, são também relevantes para esclarecimento do funcionamento do sistema. Dá-se ênfase aos aspectos de mais baixo nível como os componentes do *peer* e o funcionamento do algoritmos utilizados.

5.1 Principais Componentes

Nesta seção são apresentados os principais componentes de um *peer* do NodeWiz-R. A figura 5.1 ilustra quais são e como estão dispostos estes componentes. Um *peer* NodeWiz-R é composto por três camadas distintas e conexas. A primeira, é a camada de comunicação (*communication layer*). Ela é responsável por prover a comunicação entre cliente-peer e entre *peers*. A segunda é a camada lógica (*logic layer*). Ela é formada por módulos que se comunicam entre sí para gerenciar as principais funções de um *peer*. Os principais módulos dessa camada são o *Schema Manager*, *SQL-Parser*, *Matcher* e *Supporting Algorithms*. A terceira e última é a camada de banco de dados (*database layer*). Ela é responsável por fazer a comunicação com o banco de dados embarcado do *peer* e prover métodos para criar, atualizar e consultar um esquema de dados. As três camadas que compõem um *peer* NodeWiz-R são descritas em detalhes nas seções seguintes.

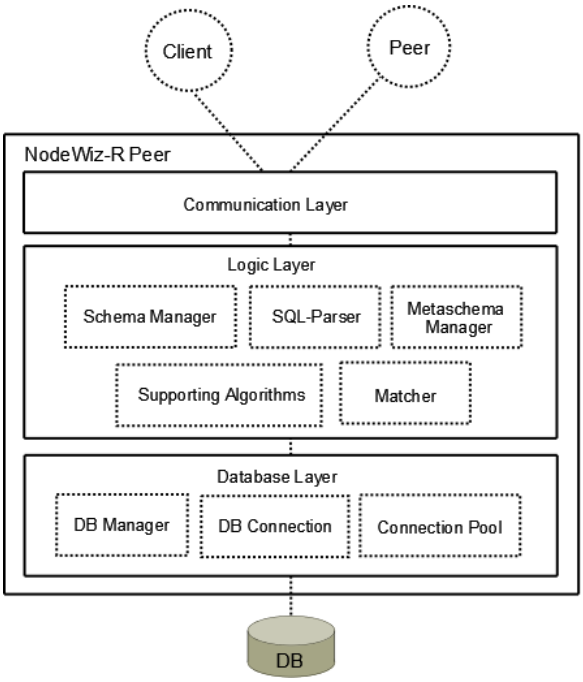


Figura 5.1: Principais componentes do Peer NodeWiz-R

5.1.1 Comunication Layer

A *Communication Layer* é a primeira das três camadas lógicas que interagem entre sí para realizar as operações de um *peer* NodeWiz-R. Ela possibilita a comunicação entre *peers* e aplicação cliente baseada na troca de mensagens e eventos assíncronos. Essa abordagem possibilita um maior grau de multiprogramação em um *peer* uma vez que várias requisições podem ser tratadas simultaneamente sem que seja necessário ficar esperando o término umas das outras. Dessa forma, quando um *peer* NodeWiz-R recebe uma requisição, ela é processada e os resultados são retornados rapidamente para o cliente que fez a requisição ou encaminhada para outros *peers*, sem que chamadas remotas bloqueantes sejam realizadas. Devido ao uso de comunicação assíncrona, a camada provê métodos de *call back* para que respostas às mensagens sejam enviadas de volta para o *peer* ou para um cliente emissor de uma requisição. As respostas são armazenadas em uma fila e tratadas à medida que elas chegam.

O NodeWiz-R usa o *Commune* [46] para prover a comunicação entre os nós da rede. *Commune* é um *framework* de comunicação assíncrona baseada em eventos para aplicações distribuídas escritas em Java [37]. Ele foi escolhido como padrão de comunicação do

NodeWiz-R por várias razões. Primeiro, porque ele simplifica em vários aspectos a implementação, ocultando os detalhes de baixo nível relacionados a processos e trocas de mensagens sem muito *overhead* de programação. *Frameworks* tradicionais, como RMI [24], também provêem comunicação transparente de objetos distribuídos. Entretanto, devido ao uso do paradigma de comunicação bloqueante, os projetistas de sistema têm que lidar com múltiplas *threads* a fim de possibilitar que a aplicação trate múltiplas requisições ao mesmo tempo. Raciocinar sobre programas *multi-threaded* é mais difícil, visto que o modelo de *threads* é não-determinístico.

Commune usa um modelo assíncrono, baseado em uma fila de eventos *mono-thread*, que provê um ambiente de programação *thread-safe*, com desempenho comparável ao RMI [40]. Por isso, *Commune* não possui os problemas de concorrência decorrentes do modelo de *threads*, não desviando assim o programador de sua lógica de negócios. Outro diferencial é que o *Commune* permite comunicação na presença de *firewalls* e NATs, uma vez que ele usa servidores *Jabber* [2] e o protocolo XMPP (*eXtensible Messaging and Presence Protocol*) [5] como infra-estrutura para troca de mensagens. Isso facilita a implantação do NodeWiz-R sobre múltiplos domínios administrativos. O protocolo XMPP é baseado em XML e é usado principalmente para troca de mensagens instantâneas, permitindo que entidades troquem mensagens e informações sobre presença de maneira próxima ao tempo real [40]. Finalmente, o *Commune* provê um serviço de detecção de falhas de fácil utilização, que será útil para a implementação futura do tratamento de falhas no NodeWiz-R.

A figura 5.2 mostra o diagrama com as principais classes da implementação do NodeWiz-R e as suas interfaces de comunicação. Existem basicamente duas interfaces remotas, *PeerApplicationControl* que permite a interação da aplicação cliente com um *peer* NodeWiz-R (implementada por *PeerApplicationController*) e a interface *Peer*, para troca de informações entre os *peers* do sistema (implementada por *PeerController*). Quando um *peer* é instanciado, a classe *PeerApplication* exporta os objetos remotos *PeerApplicationController* e chama seu método *start* para iniciar o objeto remoto. A aplicação cliente ou outro *peer* pode então fazer chamadas aos métodos das interfaces para obter serviços de um *peer*. A classe *PeerDAO* mantém apenas as informações de estado de um *peer*. Por isso, a classe *PeerDAO* é invocada apenas para recuperação de informações de um *peer* e não realiza nenhum processamento.

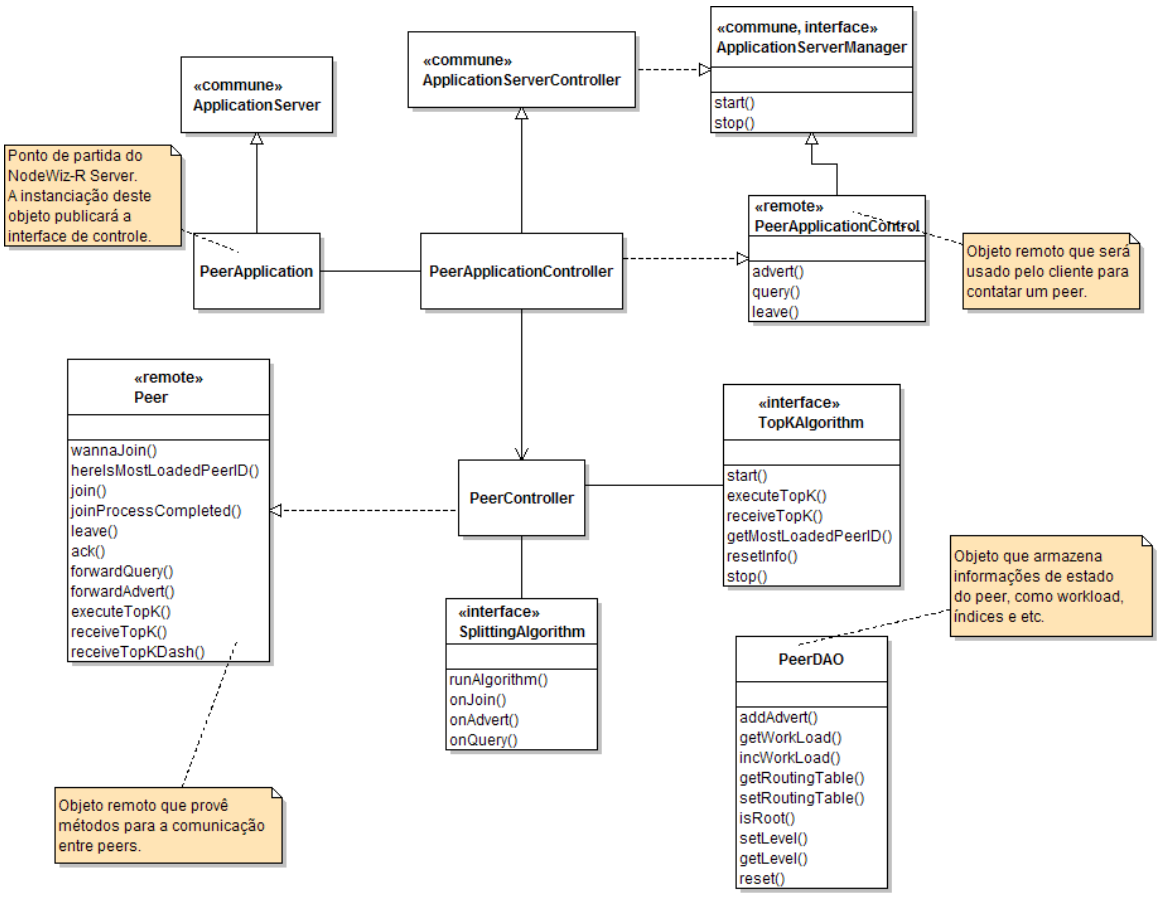


Figura 5.2: Diagrama das principais classes do NodeWiz-R e interfaces de comunicação

5.1.2 Logic Layer

A *Logic Layer* é formada por módulos que se comunicam entre sí para gerenciar as funções de um *peer*. Os principais módulos são o *Schema Manager*, *SQL-Parser*, *Matcher* e *Supporting Algorithms*.

Schema Manager

O *Schema Manager* é responsável por validar o esquema de dados fornecido, prover informações e fazer modificações no mesmo, adicionando novos campos e tabelas necessários para o funcionamento do sistema, tais como as tabelas-índice. As informações providas pelo *Schema Manager* são utilizadas para a criação do *squema-descriptor*. O *squema-descriptor* é uma espécie de meta-esquema, contendo informações sobre as tabelas do esquema no índice P2P e quais são as tabelas-índice existentes. Além disso, informa ao *peer* qual é a tabela principal e quais tabelas secundárias serão indexadas. O *squema-descriptor* é criado quando o *peer* entra no sistema e recebe o esquema de dados a ser instanciado no seu banco de dados. Ele é mantido em memória até que o *peer* deixe o sistema.

SQL-Parser

O *SQL-Parser* é responsável por gerar a expressão que será utilizada para o roteamento das operações no catálogo P2P. A expressão é uma lista de predicados formado pelo nome de um atributo, operador lógico e valor. O *SQL-Parser* gera a expressão baseada nas informações contidas no SQL fornecido, como também nas informações providas pelo *schema-descriptor*.

No caso de operações de atualização, o *SQL-Parser* analisará se alguma cláusula SQL é referente à atualização da tabela principal. Por simplificação, assume-se que um *batch* que contém uma atualização para a tabela principal é referente a um único recurso e por isso, considera-se que as demais instruções SQL contidas no mesmo *batch* são atualizações das tabelas secundárias e de ligação referentes ao mesmo recurso. Dessa forma, todo o *batch* deve ser roteado para o mesmo *peer*. Para que isso ocorra, a expressão gerada deverá conter apenas atributos da tabela principal, que determinará o roteamento de todo o *batch*. Os atributos e valores são extraídos da cláusula SQL referente à atualização da tabela principal

e são usadas para compor a expressão. Um operador lógico "AND" é usado entre os predicados que irão compor a expressão. Caso a operação seja referente à atualização de uma tabela secundária apenas, a expressão conterá apenas atributos da tabela secundária a que a instrução SQL se refere.

Planejador de Consultas

Internamente, o NodeWiz-R implementa um mecanismo para planejar a execução de consultas na rede. O planejador de execução de consultas [43; 17] do NodeWiz-R determinará quais índices serão utilizados para rotear uma consulta. Consultas para a tabela principal e para tabelas secundárias especificamente são distinguidas e várias expressões de roteamento poderão ser geradas para uma só consulta SQL submetida. As expressões são geradas estaticamente quando a consulta é submetida pelo usuário.

Em uma consulta, todos os predicados após a cláusula *WHERE* do SQL serão analisados pelo *SQL-Parser*. Ele então seleciona apenas os predicados que possuem valores numéricos ou textuais após o operador lógico de comparação. Comparação entre dois atributos de qualquer tabela não são considerados na expressão e serão analisados somente quando o SQL é executado no banco de dados. As cláusulas que sucedem as restrições da cláusula *WHERE*, como *GROUP BY*, *ORDER BY* e *HAVING*, são desconsideradas na geração da expressão. Para essa versão do protótipo do NodeWiz-R, os operadores *BETWEEN* e *LIKE* ainda não são suportados.

O planejador de consultas analisa o *batch* SQL fornecido para saber quais tabelas serão consultadas. Se a consulta SQL envolve atributos da tabela principal, apenas seus atributos serão utilizados para compor a expressão, como ilustrado na figura 5.3 (considerando que *resource* é a tabela principal). O planejador de consultas determina então que nesse caso a consulta deve ser roteada pelo índice primário. Isso permite que a operação de consulta seja roteada até o *peer* responsável pelas informações dos recursos requeridos na consulta, já que todas as informações relacionadas com uma tupla da tabela principal estão armazenadas no mesmo *peer* e portanto, todas as informações requeridas no SQL poderão ser localizadas através do roteamento pelo índice primário.

No caso de SQLs contendo somente atributos de tabelas secundárias, a expressão será gerada baseada nesses atributos. Se o SQL contém o operador lógico *AND* entre atributos

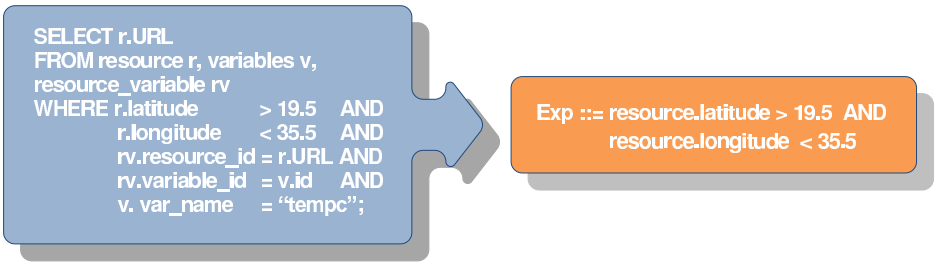


Figura 5.3: Conversão de uma consulta SQL para uma expressão NodeWiz-R

de tabelas secundárias diferentes, apenas os atributos de uma das tabelas serão usados na expressão. Isso porque como a condição *AND* tem de ser satisfeita entre as restrições, a expressão só poderá casar com aqueles *peers* que tenham pelo menos uma das informações requeridas. Portanto, a expressão pode conter apenas um dos atributos do SQL dado. Caso um *peer* não seja responsável pelo subespaço requerido na expressão, o SQL não poderá ser satisfeito pois a condição *AND* não poderá ser satisfeita. Caso seja satisfeito por algum *peer*, o SQL completo será processado no banco de dados local dos *peers* onde houve o casamento com a expressão e somente as informações armazenadas que casam completamente com todas as restrições estabelecidas no SQL serão retornadas. O planejador de consultas determinará que o índice secundário a ser consultado será aquele que referente aos atributos utilizados na expressão gerada e o roteamento será feito através desse índice.

Na existência do operador *OR* entre atributos de tabelas secundárias diferentes no mesmo SQL, os atributos das tabelas contidas na expressão envolvidas no *OR* serão utilizados para gerar uma expressão correspondente a cada tabela. Assim, várias expressões poderão ser geradas pelo *SQL-Parser* para um mesmo SQL e esse poderá ser executado em vários *peers* diferentes. Isso é feito uma vez que qualquer *peer* que satisfaça apenas uma condição do *OR* pode responder à consulta SQL enviada. No caso da geração de mais de uma expressão, serão criadas operações de consultas distintas, uma para cada expressão, contendo o mesmo SQL. O planejador de consultas verificará quais tabelas estão envolvidas na consulta e quais índices deverão ser consultados. As operações serão roteadas de forma independente, pelos índices indicados pelo planejador de consultas, para os respectivos *peers* que mantém as tabelas-índice necessárias para a realização da segunda fase do roteamento das operações. Os resultados encontrados serão enviados para a aplicação cliente que é responsável pelo processamento final dos resultados.

Matcher

O Matcher é responsável por verificar se uma dada expressão casa com as informações nos índices de um *peer*, seja ele primário ou secundário. A verificação é feita analisando se o valor do atributo de uma expressão está dentro da faixa de valores mantidas por algum *peer*. A checagem é feita a partir do último nível consultado no índice do *peer* anterior. Por exemplo, quando o primeiro *peer* recebe uma consulta, o *matcher* verifica a partir do nível zero do índice correspondente se existe casamento com a expressão. Caso haja um casamento, a operação será roteada para o *peer* indicado no nível zero. No *peer* para onde a operação foi roteada, o *matcher* verificará no mesmo índice, se há algum casamento com as entradas nos níveis posteriores ao nível zero. Caso ele alcance o último nível do índice e nenhum casamento com a expressão for constatado, então a operação é roteada para todos os *peers* dos níveis posteriores ao nível zero.

Isso ocorre pois um atributo presente na expressão pode não ter sido utilizado como base na divisão do espaço de atributos de nenhum *peer*, e portanto não se encontra nos índices do *peer* corrente. Caso a entrada no *peer* que roteou a operação corresponda ao último nível, então o *matcher* do *peer* que recebeu a operação roteada não deverá mais verificar se há casamentos nas entradas de seus índices, a operação é executada e os resultados são retornados. O caminhamento sequencial pelos níveis dos índices, que representa o caminhamento entre os níveis da árvore binária de indexação, impede que *peers* que já foram consultados anteriormente sejam consultados novamente.

Supporting Algorithms

Os *Supporting Algorithms* são os algoritmos que dão suporte a tarefas de manutenção do índice. Eles são os mesmos utilizados no NW, porém, adaptados para trabalhar com o modelo relacional do SGBD. Os principais algoritmos são: *Top-K*, que identifica qual é o *peer* mais sobrecarregado na rede, o qual será escolhido para dividir seu espaço de atributos com um novo *peer* que ingresse na rede; *Splitting Algorithm*, que define qual atributo e valor serão utilizados como base na divisão do espaço de atributos de um *peer*.

O algoritmo *Top-K* é usado para detectar qual é o *peer* mais sobrecarregado da rede. Ele reúne informações sobre os *peers* e ordena os nós baseado num indicador de *workload*

que é mantido por cada *peer*. Esse indicador é incrementado a cada consulta ou atualização recebida pelo *peer*. Periodicamente esse indicador é dividido por 2 para dar mais peso ao *workload* mais recente. O *Top-K* é executado periodicamente ou sob demanda, quando um novo *peer* entra no sistema. Isso vai depender de quão freqüente é a entrada de novos *peers*. Se um novo *peer* entra no sistema depois de uma requisição ao *Top-K* ter sido enviada, a informação sobre quem é o próximo *peer* mais sobrecarregado da lista é encaminhada para esse novo nó [9]. O *Top-K* é distribuído e roda em duas fases. Na primeira fase, cada nó envia uma mensagem para outros nós selecionados do seu índice primário. As mensagens devem percorrer toda a árvore do NodeWiz-R. Os nós não folha devem aguardar por um período de tempo pelas mensagens dos seus respectivos filhos. Quando a mensagem chega, o nó inclui seu *workload* juntamente com sua identificação e ordena as informações para serem enviadas para o nó anterior, que enviou a requisição. Esse procedimento se repete até que todas as mensagens cheguem ao nó *root* da árvore. O nó *root* monta a lista dos *Top-K workloads* e envia na segunda fase para todos os *peers* [9].

O *Splitting Algorithm* é usado para dividir em duas a faixa de valores mantida por um *peer*. Ele determinará qual atributo e valor de corte garantirão uma boa divisão da carga de trabalho e garantirá que o crescimento do sistema seja feita de forma balanceada. O *Splitting Algorithm* analisa um histograma com valores de cada atributo em operações de atualização e consultas recebidas por um *peer* desde a última vez que o algoritmo foi executado. Os valores dos atributos são agrupados em dois grupos. O atributo escolhido deve satisfazer duas condições: Primeiro, seus valores devem ter alta probabilidade de se agruparem em dois grupos pertencentes a duas faixas de valores. Segundo, os agrupamentos formados pelos seus valores devem estar bem balanceados, ou seja, um agrupamento deve conter uma quantidade de valores aproximada ao do outro agrupamento de forma a não formar uma faixa de valores desbalanceada. O algoritmo *K-Means Clustering* [30] é utilizado para fazer o agrupamento e classificação dos valores dos atributos. A idéia do algoritmo é fazer o agrupamento de acordo com dados dos próprios valores, baseada em análises e comparações entre eles. O valor de corte é determinado pelo limite entre o último valor do primeiro agrupamento e o primeiro valor do segundo agrupamento do atributo escolhido. No NodeWiz-R, o *Splitting Algorithm* é utilizado tanto sobre a tabela principal, para compor o índice primário, como também sobre as tabelas-índices para compor seus respectivos índices secundários.

5.1.3 Database Layer

A terceira e última camada que forma um *peer* NodeWiz-R é a camada de banco de dados (*database layer*). Ela é responsável por fazer a comunicação com o banco de dados embarcado do *peer* e prover métodos para criar, atualizar e consultar um esquema de dados. Ela também gerencia o número de conexões criadas com o banco de dados através de um *pool* de conexões. Isso possibilita que conexões abertas sejam reusadas, aumentando assim o desempenho das operações sobre o banco de dados pois reduz-se o *overhead* gerado ao abrir e fechar conexões constantemente a cada requisição.

Embedded Lightweight Database

O *Embedded Lightweight Database* é um componente essencial na arquitetura do NodeWiz-R, uma vez que ele armazena as informações sobre os recursos publicados pelos usuários/provedores usando o modelo relacional e provê o suporte final para o processamento das consultas ricas expressas na linguagem SQL. A escolha do SBD usado pelo NodeWiz-R foi definida de acordo com os seguintes requisitos levantados:

- *Open Source*: O SBD deve ser *Open Source* e ter uma comunidade ativa de usuários e desenvolvedores;
- *Embedded*: Deve ser embarcado na aplicação, para facilitar o trabalho de implantação do *peer* em qualquer domínio administrativo;
- *Lightweight*: Deve ter um pequeno *footprint*, ou seja, ocupar pouco espaço na memória de forma a não sobrecarregar o *host* onde o *peer* será hospedado;
- *Suporte a gatilhos*: Deve prover suporte a gatilhos (*triggers*), a fim de automatizar algumas checagens de restrições de integridade;
- *Alto desempenho*: O SBD deve apresentar bom desempenho nas operações de consulta e atualização, de forma a interferir da menor forma possível no tempo de resposta do sistema.

Com base nesses requisitos, constatou-se que as melhores opções de SBD que poderiam atender esses requisitos são as seguintes: **Derby**[1], **HSQLDB**[2] and **H2**[3]. Para auxiliar na decisão sobre qual tecnologia utilizar, foi realizado um teste de *benchmark* com a ferramenta OSDB (*The Open Source Database Benchmark*)[4]. Ele apontou que o desempenho do **H2** foi superior em várias métricas avaliadas, principalmente o tempo de resposta de variados tipos de consultas. Além disso, o *H2* preenche os demais requisitos acima elencados. Por essa razão, o *H2* foi escolhido como a implementação de SBD relacional do NodeWiz-R. Como o foco principal deste trabalho não é apontar qual das soluções de SBD é a melhor, achamos desnecessária a divulgação dos resultados do *benchmark* realizado.

O *H2* é um SGBD relacional *Open Source* escrito inteiramente em Java que está em produção desde 2005. O consumo de espaço em memória persistente para sua instalação é de aproximadamente 1MB. Ele pode ser embarcado em aplicações Java e permite que o banco de dados inteiro seja criado na memória, o que lhe confere muita rapidez e portanto um ótimo desempenho nas operações. Para o NodeWiz-R, o armazenamento em memória é ideal, uma vez que os dados sobre os recursos são transientes. *H2* suporta um grande subconjunto do padrão ANSI SQL, incluindo *triggers*. Entretanto, o NodeWiz-R possui algumas limitações e por isso, não provê suporte a todo o conjunto de operações SQL disponíveis para o *H2*.

Suporte SQL

Como mencionado na seção anterior, para essa primeira versão, o NodeWiz-R não dispõe de suporte total ao conjunto de operações SQL provido pelo *H2*. Não obstante, o NodeWiz-R permite que consultas ricas sejam elaboradas. A gramática para elaboração de consultas e atualizações no NodeWiz-R é apresentada abaixo. Ela é baseada na gramática fornecida pelo *H2* e portanto, está estritamente relacionada com o suporte SQL fornecido pelo *H2*. A gramática abaixo correspondente ao comando $SELECT$, que permite selecionar dados de uma ou múltiplas tabelas do esquema e ainda aplicar restrições, fazer agrupamentos, aplicar restrições sobre os agrupamentos, ordenar e limitar os resultados. Operações de junção implícitas também são suportadas. Já junções explícitas, expressas com as instruções

[1] http://db.apache.org/derby/

[2] http://hsqldb.org/

[3] http://www.h2database.com

[4] http://osdb.sourceforge.net/

INNER JOIN ou *OUTER JOIN* e *sub queries*, não são suportadas atualmente.

```
{SELECT selectPart FROM fromPart}
[WHERE expression]
[GROUP BY expression [,...]] [HAVING expression]
[{UNION [ALL] | MINUS | EXCEPT | INTERSECT} select]
[ORDER BY order [,...]] [LIMIT expression [OFFSET expression]
[SAMPLE_SIZE rowCountInt]] [FOR UPDATE]
```

No caso de consultas com agregação, o resultado da agregação será gerado por cada *peer*, independente de outros resultados sendo produzidos por outros *peers*. Dessa forma, se vários *peers* estiverem respondendo à mesma consulta, os resultados da agregação apresentados serão referentes ao que foi encontrado no SBD de cada *peer* e não do sistema como um todo. Esse problema pode ser mitigado da seguinte maneira: o primeiro *peer* que recebeu a consulta, denotado por $F(P)$, deverá aguardar por um tempo determinado, pelos resultados produzidos pelos outros *peers*, caso ele tenha feito o roteamento da consulta. Os resultados produzidos pelos outros *peers* são então enviado para $F(P)$. $F(P)$ cria uma tabela temporária vazia no seu banco de dados, e essa tabela é atualizado com os resultados enviados pelos outros *peers* até o *timeout* determinado. A consulta é reformulada com os mesmo critérios de agrupamento determinados na consulta submetida originalmente, mas adequada para o formato da tabela temporária que armazena os resultados, e é executada sobre ela. Um novo agregado é produzido, agora correspondente a todas as informações encontradas no sistema, e devolvido ao usuário.

O problema da solução apresentada é que o conjunto de resultados é entregue apenas de uma só vez, após o *timeout* determinado, enquanto que sem essa solução, os resultados são devolvidos imediatamente ao usuário, logo que são encontrados por cada *peer*. O mesmo problema de agregação ocorre com as cláusulas *LIMIT* e *ORDER BY*, mas podem ser solucionados da mesma forma.

A gramática para elaboração de atualizações é apresentada abaixo. São aceitas as instruções *INSERT INTO* ou *MERGE INTO*, como também a instrução de atualização *UP-*

DATE. Uma instrução usando *INSERT INTO* é transformada internamente em *MERGE INTO*, para permitir a atualização direta de uma informação, caso ela já esteja inserida na tabela. Caso a informação já exista, e possua os mesmos valores para todos os seus atributos, apenas o seu $TOUT$ será atualizado.

```
INSERT INTO tableName
{VALUES {( {DEFAULT} [,...] )}

UPDATE tableName SET {columnName={DEFAULT} } [,...]
[WHERE expression]
```

5.2 Mantendo a consistência das informações

Um problema a ser considerado na solução proposta pelo NodeWiz-R, é a questão da consistência das informações publicadas. Embora esse não seja o foco principal desse trabalho e não seja algo preocupante no contexto da descoberta de recursos, a consistência das informações armazenadas sobre os recursos são tratadas de forma simples e eficiente pelo NodeWiz-R.

A validade das informações em relação aos provedores é garantida principalmente através do uso da abordagem de armazenamento *soft-state*, já comentado no capítulo 4. Como as informações expiram com o passar do tempo e devem ser republicadas pelos seus provedores, garantimos que as informações sobre os recursos estão sempre no estado mais atualizado em relação aos seus provedores. Entretanto, o quão atualizada está a informação em relação ao seu provedor irá depender do espaço de tempo definido pelo provedor para fazer as atualizações. Essa abordagem também garante a consistência das tabelas-índice, pois como já foi dito no capítulo 4, um $TOUT$ também é associado às informações contidas nas tabelas-índice e também são removidas quando os seus $TOUT$ expiram e são atualizadas quando as informações sobre os recursos são republicadas no catálogo P2P.

Mas, se considerarmos que diferentes recursos podem ter informações de tabelas secundárias em comum associadas, podemos ter um problema de consistência caso uma

atualização nas informações comuns seja feita. Por exemplo, uma mesma *tag* é usada para classificar recursos distintos de provedores distintos (*tag* não-exclusiva). Caso um dos provedores resolva atualizar o nome de uma *tag* não-exclusiva, os demais recursos também teriam suas *tags* alteradas sem qualquer conhecimento prévio.

Para mitigar esse problema, o NodeWiz-R adiciona em cada tabela do sistema um campo *USER_ID*. O *USER_ID* é formado pelo *ID Jabber* do usuário que está publicando as informações no NodeWiz-R. O *ID Jabber* é o identificador único do usuário nos servidores *Jabber* que são utilizados na rede NodeWiz-R. O *USER_ID* é utilizado para compor as chaves primárias de todas as tabelas do esquema do índice. Se uma tabela já possui chave primária definida no esquema, o NodeWiz-R modificará a sua definição no esquema e adicionará o novo campo, *USER_ID*, para compor a chave primária da tabela. Dessa forma, a tabela passará a ter uma chave composta. Caso a tabela não tenha uma chave previamente definida, o *USER_ID* passa então a ser a sua chave primária.

A lógica dessa solução é fazer com que todas as informações publicadas no NodeWiz-R tenham a identificação (ID) do usuário que a publicou. Portanto, se uma mesma informação é compartilhada por vários recursos de diferentes provedores, até mesmo a chave primária, uma operação de atualização não afetará as informações publicadas por outros provedores. A atualização será feita apenas nas informações cujo *USER_ID* seja igual ao do provedor que está enviando a atualização. Isso é feito de forma transparente e por isso, o usuário não precisa se preocupar com conflitos de chave primária ou atualizações não autorizadas de dados compartilhados.

5.3 Tolerância a falhas

Em um sistema que pode ter potencialmente muitas centenas de componentes instalados, a falha de alguns desses componentes é praticamente uma norma e não uma exceção [9]. Portanto, uma característica importante e desejável nos sistemas distribuídos de modo geral, é a capacidade de tolerar falhas. Assim como qualquer sistema distribuído, o NodeWiz-R está sujeito a falhas que podem culminar na perda de parte das informações sobre os recursos e dos seus índices. O substrato P2P **NW** implementa um mecanismo para tolerar falhas e lidar apropriadamente com a saída voluntária dos *peers* do sistema. Esse mesmo mecanismo

pode ser implementado pelo NodeWiz-R para detectar eventuais falhas e saída de *peers* do sistema. O mecanismo é descrito em resumo a seguir. Maiores detalhes podem ser obtidos em [15; 9].

Quando um *peer* (L) deixa o sistema voluntariamente, o subespaço de atributos que ele armazena deve ser assumido por outro *peer* do sistema. O *peer* escolhido para assumir é o último *peer* (R) com quem ele dividiu seu subespaço de atributos ou o seu substituto, caso ele já tenha deixado o sistema. Todas as entradas nos índices dos outros *peers* que contêm uma referência para o *peer* L deve ser removida ou atualizada com uma referência para o *peer* R, que está assumindo seu subespaço. O *peer* L envia para R todas as informações sobre os recursos que ele é responsável e R se encarrega de enviar as mensagens de atualização para o *peer* imediatamente anterior e os inferiores a L no seu índice primário. Após receber uma mensagem de autorização, o *peer* L pode então deixar o sistema efetivamente.

Para lidar com saídas involuntárias (falhas), um mecanismo para detecção de falhas foi proposto. O mecanismo propõe recuperar somente a consistência dos índices, e portanto, prevenindo que falhas nas operações do sistema ocorram. As informações sobre os recursos armazenadas do *peer* falho são perdidas. Entretanto, como as informações são renovadas constantemente, se os índices forem recuperados, essa perda será apenas temporária.

A detecção da falha é o principal componente do mecanismo de tolerância a falhas do NW. Os *peers* ficam responsáveis pelo monitoramento uns dos outros, enviando e recebendo periodicamente mensagens de monitoramento. O *peer* correspondente à última entrada na tabela de roteamento de um outro *peer* é responsável por monitorá-lo. Por exemplo, a última entrada na tabela de roteamento do *peer* O possui uma referência para o *peer* P. Portanto, o *peer* O é responsável por monitorar o *peer* P e é referido como $M(P)$. Quando a resposta esperada não é enviada para o *peer* monitor, ele suspeita da falha do *peer* monitorado. Quando o *peer* P falha, $M(P)$ detecta a falha e cria um *peer* virtual que assume a identidade de P. O *peer* virtual que personifica o *peer* P é chamado de *peer* sombra (*shadow*), e é referido como $S(P)$. O nó $M(P)$ também mantém o *peer* virtual $S(P)$ temporariamente. Quando o $S(P)$ é iniciado, ele comunica a sua saída voluntária do sistema. O mesmo procedimento para saída voluntária de um *peer* do sistema é realizado, o que mantém a rede consistente. O *peer* $S(P)$ apenas aguarda a autorização para deixar o sistema. Chegada a autorização, todos os *peers* já estão com suas tabelas de rotas atualizadas e o *peer* sombra pode deixar o

sistema.

Esse mesmo mecanismo pode ser implementado no NodeWiz-R sem que grandes esforços sejam requeridos. Até mesmo as tabelas-índice podem ser recuperadas usando a mesma abordagem, bastando para isso que os índices secundários estejam em um estado consistente para que as tabelas-índices perdidas sejam recriadas após novas atualizações. O mecanismo de detecção de falhas pode ser facilmente ativado, bastando para isso usar todas as funcionalidades disponíveis no *framework Commune*, que dispõe de um mecanismo para notificação de falhas e recuperação de objetos distribuídos.

Capítulo 6

Avaliação da Solução

Neste capítulo, é apresentada uma avaliação de desempenho do NodeWiz-R. A avaliação foi feita sobre uma perspectiva *quantitativa*, onde avaliamos duas métricas usando um modelo simulado do sistema. A primeira consiste em avaliar o desempenho da aplicação em termos da quantidade de informação (sobrecarga) que os processos distribuídos que implementam o serviço precisam trocar, sendo uma indicação indireta da eficiência do serviço. Sobre essa ótica, consideramos que quanto menor a sobrecarga, mais eficiente é a solução. A segunda é uma medida da acurácia do sistema (cobertura) e é calculada como sendo a relação entre a quantidade de recursos que foram efetivamente descobertos e a quantidade de recursos que deveriam ter sido descobertos. Nessa perspectiva, consideramos que a cobertura ideal deve ser de 100%.

Em ambas as avaliações, o NodeWiz-R será comparado com o banco de dados P2P *PeerDB*, que é um dos sistemas P2P que mais se aproxima da solução proposta pelo NodeWiz-R em termos de funcionalidades. Mais precisamente, queremos comparar o desempenho do mecanismo de localização de informações utilizado pelo *PeerDB*, que é baseado em *flooding*, versus o sistema de indexação do NodeWiz-R.

Na primeira avaliação, a expectativa é que o NodeWiz-R apresente um bom desempenho, mantendo uma alta taxa de retorno independente do tipo de consulta realizado e do tamanho da rede. Na segunda avaliação, também se espera que o NodeWiz-R apresente uma alta taxa de cobertura com uma sobrecarga muito menor, uma vez que a quantidade de mensagens necessárias para executar suas operações é substancialmente inferior, comparado ao mecanismo de *flooding* implementado pelo *PeerDB*.

6.1 Avaliação de desempenho

Para avaliar a solução, foram desenvolvidos dois simuladores. Um simulador para o NodeWiz-R e outro para a solução P2P PeerDB. Nas simulações nós analisamos a eficiência do mecanismo de roteamento do NodeWiz-R versus o mecanismo adotado por PeerDB. No NodeWiz-R especificamente, verificamos a eficiência dos índices secundários e tabelas-índice no roteamento de consultas sobre valores contidos nas tabelas secundárias e em consultas por faixa de valores. Esse parâmetro foi escolhido para ser avaliado uma vez que é de extrema importância para qualquer sistema distribuído conhecer a quantidade de mensagens produzidas para realização de suas operações. A taxa de retorno de resultados das consultas também é analisada uma vez que grande parte dos sistemas P2P tradicionais não garantem uma alta taxa de retorno. Ao contrário dos sistemas de bancos de dados tradicionais, onde é tipico afirmar que todo fato armazenado no banco é retornado quando requisitado por uma consulta. Portanto, avaliamos a capacidade de retorno de informações do NodeWiz-R frente ao PeerDB. Optou-se pela técnica de simulação pois ela permite a resolução mais fácil do modelo proposto, do que se resolvêssemos o modelo matematicamente, e por possibilitar um maior controle dos fatores que afetam a execução e redução da complexidade do sistema, do que se fosse utilizada a implementação real da solução.

Como foi dito anteriormente, duas métricas foram usadas para comparar os dois sistemas. A **cobertura** média, que é dada pela razão entre o número de respostas recebidas e o número de respostas esperadas; e a **sobrecarga** média, que foi medida em $Kbytes$ e foi obtida através da equação:

$$S = \frac{n \cdot b}{1024 \cdot k},$$

onde n é o número de mensagens transmitidas, b é o tamanho da mensagem em $bytes$ e k é o número de vezes (séries) que as simulações foram rodadas. O número 1024 corresponde ao valor para conversão de $byte$ para $kilobyte$. Uma mensagem do NodeWiz-R tem aproximadamente 110 $bytes$, enquanto que uma mensagem do PeerDB tem aproximadamente 65 $bytes$.

6.1.1 Simulador do PeerDB

Nesta seção descreve-se como é o funcionamento do PeerDB, de acordo com sua especificação [47] e como foi implementado o seu simulador. No sistema PeerDB, cada *peer* se conecta diretamente a um número fixo de nós da rede. A lista de nós que um *peer* deverá se conectar é fornecida por um servidor de nomes (*LIGLO - Global Names Lookup Server*), que gerencia a identidade dos *peers* e conhece todos os *peers* que estão conectados na rede. A lista é passada quando um novo *peer* ingressa no sistema e os nós contidos nessa lista são escolhidos aleatoriamente pelo *LIGLO Server*. As consultas enviadas para um *peer* são encaminhadas para todos os nós presentes na lista de nós conectados diretamente, mantida por cada *peer*.

As consultas são realizadas em duas fases: a primeira consiste em contatar todos os nós que cada *peer* da rede conhece, usando para isso a técnica de *flooding*. Essa etapa tem a finalidade de se obter os metadados que descrevem as informações que estão armazenadas em cada SBD mantido por cada *peer*. A segunda fase consiste em selecionar dentro dos metadados obtidos, aqueles que têm as informações desejadas pelo usuário e a consulta é definitivamente enviada para os *peers* referentes aos metadados selecionados. Portanto, os *peers* para os quais as consultas serão enviadas na segunda fase são previamente escolhidos após toda ou parte da rede ser consultada na primeira fase, e só então as consultas são realizadas efetivamente e os resultados são retornados para o usuário. Um TTL é associado a cada consulta como artifício para limitar a quantidade de mensagens transmitidas na rede. O TTL é decrementado em cada nó que recebe a consulta e quando o valor zero é atingido, a consulta não é mais propagada.

Por simplificação, para o simulador, consideramos apenas a primeira fase do processamento das consultas, uma vez que a primeira fase é a que realmente caracteriza o mecanismo de *flooding* adotado pela solução e por outras do gênero. Além disso, na segunda etapa, a escolha dos *peers* é realizada sem um critério específico definido, podendo uma consulta ser enviada para qualquer *peer* da rede sob escolha do usuário.

O simulador do PeerDB foi implementado para operar em três modos distintos: *estático*, *dinâmico* e *estático com constante*. No modo *estático*, que nos gráficos chamamos apenas de **PeerDB**, cada nó se conecta diretamente a $log(n)$ nós. No modo dinâmico (**PeerDB D**), cada nó se conecta diretamente a $log(n)$ nós e pode se conectar diretamente a outros nós mais

promissores gradativamente, sob o critério do número de resultados retornados pelos nós em consultas anteriores. No terceiro modo (**PeerDB +C**), cada nó se conecta a $log(n) + c$ nós, onde c é uma constante inteira fornecida como parâmetro da simulação. Essa constante foi previamente calculada para que se obtivesse uma rede totalmente conectada de acordo com o número de nós utilizados na simulação. Em todos os casos, variou-se o nível máximo de inundação usado (TTL) a fim de se verificar qual valor seria necessário para se obter a maior taxa de cobertura. Como o NodeWiz-R não utiliza TTL para limitar a troca de mensagens na rede, esse parâmetro foi considerado apenas para o simulador do PeerDB. O TTL foi configurado com os valores 1, 3, 5, 6, 8 e 10, já que foi constatado no início das simulações que para valores superiores não se obtinha ganhos significativos.

Ambos os simuladores foram implementados com a linguagem de programação Java (versão 1.6) devido a sua independência de plataforma operacional, facilitando assim a condução das simulações. Os simuladores implementados, bem como os cenários utilizados nas simulações estão disponíveis para download no seguinte endereço: *http://redmine.lsd.ufcg.edu.br/projects/list_files/nodewiz*.

6.1.2 Configuração do ambiente das simulações

As simulações foram executadas para redes lógicas compostas por 500, 1.000, 3.000, 5.000 e 10.000 nós. Embora o número máximo de nós que o simulador conseguiu simular tenha sido 10 mil, essa não é uma limitação da arquitetura proposta em si. Foram geradas 10 séries com uma carga de trabalho (*workload*) sintética, cada uma com informações sobre 100.000 recursos distintos. Optou-se por utilizar dados sintéticos uma vez que não se dispunha de dados adequados e suficientes para produzir um cenário com centenas de milhares de recursos distintos. Os atuais provedores de recursos do SegHidro produzem dados diariamente, mas estes são descartados por questões de limitação de espaço em disco. Por isso, seriam necessários dias ou até meses para se obter o *workload* desejado. Além disso, os dados produzidos são similares, com variações apenas no espaço temporal a que se refere. O uso da *workload* sintética possibilitou simular um cenário onde os dados sobre os recursos possuíam proximidade de valores sobre uma ampla faixa, caracterizando uma diversificação dos recursos publicados.

No caso do NodeWiz-R, a distribuição das informações foi feita automaticamente pelo

mecanismo de distribuição e balanceamento de carga do sistema. O sistema foi colocado em funcionamento da seguinte maneira: inicialmente a rede continha apenas um único *peer* participante. Todas as informações foram adicionadas nesse *peer*. Após o término da atualização do *peer* com as informações, foram adicionados novos nós ao sistema. A escolha do *peer* o qual o novo nó deveria contatar foi feita aleatoriamente. Portanto, todos os *peers* que já estavam na rede tinham a mesma probabilidade de receber uma requisição de junção à rede. O *peer* mais sobrecarregado era detectado e dividia o seu subespaço de atributos com o novo nó. Após todos os nós se juntarem à rede, as consultas eram submetidas.

No PeerDB, as informações foram distribuídas de forma que cada nó mantivesse um conjunto de informações com valores aproximados entre si, simulando a proximidade semântica e de valores geralmente encontrada em um cenário real num provedor de recursos do PeerDB. As informações foram divididas de forma igualitária entre todos os nós. O sistema foi colocado em funcionamento da seguinte maneira: uma lista de nós participantes era carregada juntamente com as informações que deveriam ser armazenadas por todos os nós. As informações foram divididas pelo número de nós e eram atribuídas à medida que um nó era adicionado no sistema. Após fazer a divisão das informações com todos os nós, eles recebiam a lista de nós com os quais cada um deveria se conectar diretamente. Os $log(n)$ nós que formavam cada lista dada aos participantes eram sorteados aleatoriamente, simulando o comportamento do *LIGLO Server*.

O esquema de dados utilizado continha uma tabela principal e uma tabela secundária. Ambas as tabelas continham dois atributos do tipo *Double*. Os valores para os atributos foram atribuídos usando uma distribuição uniforme, sobre uma ampla faixa de valores. Por simplicidade, no NodeWiz-R nós assumimos que os dados não expiravam, o que facilitou a identificação da quantidade de resultados esperados em cada consulta submetida.

Para ambos os simuladores, foram submetidas 1.000 consultas de dois tipos diferentes: (i) consultas com casamento exato de valores (usando o operador de igualdade); (ii) e consultas por faixa de valores usando o operador $>$. Os formatos das consultas submetidas são apresentados nas tabelas 6.1 e 6.2. As consultas foram submetidas sobre a tabela principal (Tabela1) e a tabela secundária (Tabela2) separadamente. Elas foram geradas a partir de um subconjunto das informações publicadas. Dessa forma, todas as consultas submetidas casavam com pelo menos um recurso. No caso de consultas por faixa de valores, mais de um

resultado poderia ser retornado. A escolha do nó para o qual a consulta seria submetida foi feita aleatoriamente, de forma que qualquer nó possuísse a mesma chance de ser selecionado para recebê-la.

```
SELECT Att1, Att2
FROM [Table1 | Table2]
WHERE Att1 = VALUE1
[AND Att2 = VALUE2];
```

Tabela 6.1: Formato das consultas com casamento exato de valores usadas nas simulações

```
SELECT Att1, Att2
FROM [Table1 | Table2]
WHERE Att1 > VALUE1
[AND Att2 > VALUE2];
```

Tabela 6.2: Formato das consultas por faixa de valores usadas nas simulações

Todas as simulações foram executadas na infra-estrutura de grade computacional OurGrid [20], o que possibilitou que várias simulações fossem realizadas em paralelo, aproveitando os recursos computacionais ociosos disponíveis na grade. Uma vez que os recursos da grade são heterogêneos, o tempo de execução das simulações foi bastante variável. O tempo mínimo de uma simulação foi de 10 minutos, para um cenário com 500 nós e TTL igual a 3, enquanto que aproximadamente 6 horas foram necessárias para o processamento do cenário com 10 mil nós e TTL igual a 10 para o simulador do PeerDB. Como o tempo de execução das consultas não foi uma métrica analisada, as simulações foram conduzidas sem a preocupação de qualquer influência negativa nos resultados devido à contenção dos recursos da grade computacional.

6.1.3 Cenários

Foram gerados 6 tipos de cenários que nos permitiram analisar o comportamento do sistema diante de diferentes tipos de consultas. Os cenários para as simulações são descritos abaixo:

- *Cenário 1*: consultas com casamento exato de valores (*exact-match queries*) sobre os dois atributos da tabela principal;
- *Cenário 2*: consultas por faixa de valores (*range queries*) com o operador $>$ (maior que) sobre 1 atributo da tabela principal;
- *Cenário 3*: consultas com casamento exato de valores sobre 2 atributos da tabela secundária (valores exclusivos);
- *Cenário 4*: consultas com casamento exato de valores sobre 1 atributo da tabela secundária. A fim de se aproximar de um caso real de uso, neste cenário foi reduzida a faixa de valores possíveis para os atributos da tabela secundária. Isso fez com que os dados da tabela secundária se relacionassem com mais de uma tupla da tabela principal e fossem replicados em diversos nós no momento da divisão do espaço de atributos. Portanto, as tabelas secundárias continham valores não exclusivos.
- *Cenário 5*: consultas por faixa de valores com operador $>$ sobre 1 atributo da tabela secundária e com replicação dos valores da tabela secundária (não exclusivos);
- *Cenário 6*: consultas por faixa de valores com operador $>$ sobre 2 atributos da tabela secundária e com replicação dos valores da tabela secundária (não exclusivos).

Os cenários 1 e 2 foram configurados para medir a eficiência do índice primário no roteamento de consultas contendo atributos da tabela principal, tanto para consultas com casamento exato de valores, quanto para consultas por faixa de valores. Os cenários 3, 4, 5 e 6 foram configurados para medir a eficiência do índice secundário e tabela-índice. Os cenários 4, 5 e 6 particularmente foram configurados para simular um caso onde as informações da tabela secundária não são exclusivas e portanto, podem ser replicadas devido ao seus relacionamentos com outros dados da tabela principal. As consultas escolhidas fazem parte do subconjunto das consultas tipicamente enviadas para os serviços de descoberta de recursos.

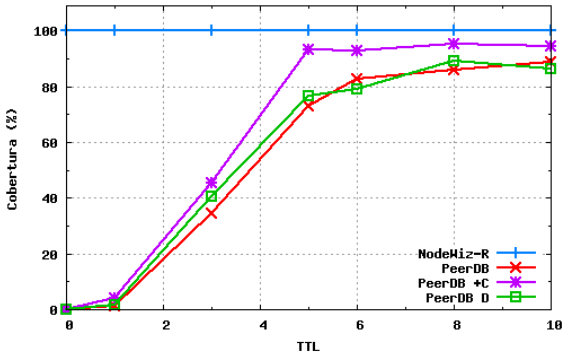
Evidentemente que os cenários escolhidos não cobrem todos as possíveis configurações de consultas existentes. Mas esses cenários poderão fornecer informações suficientes para a indicação da eficiência do mecanismo de roteamento do sistema.

6.1.4 Resultados

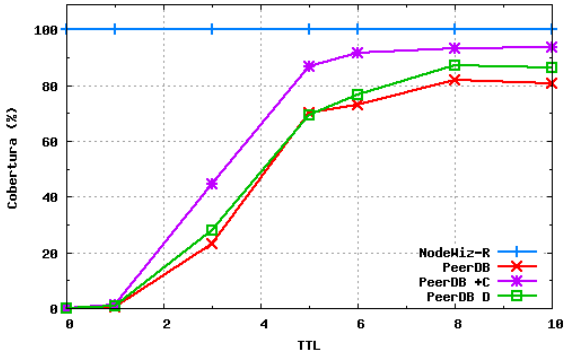
Cobertura x TTL

Primeiramente, analisamos a métrica *Cobertura* versus o TTL usado pelo PeerDB. Como podemos observar na Figura 6.1 em todos os cenários avaliados o NodeWiz-R consegue obter uma taxa de cobertura de 100%. No caso do **PeerDB** e **PeerDB D**, a taxa de cobertura média ultrapassa 70% apenas quando o TTL é superior a 6. A partir daí o aumento na cobertura é discreto, permanecendo sempre abaixo de 90%, mesmo com o incremento do TTL. O **PeerDB +C** obtém resultados um pouco melhores, conseguindo taxas de cobertura superiores a 90%. No caso do **PeerDB +C** a quantidade de nós conectados diretamente é maior desde o início da simulação. Ou seja, desde o início cada *peer* se conecta a um número fixo de nós, enquanto que no modo dinâmico (**PeerDB D**) novos nós são adicionados gradativamente na lista de nós conectados diretamente. Isso faz com que mais nós sejam consultados com o mesmo TTL, aumentando a quantidade de resultados retornados para o modo **PeerDB +C**. Todavia, quanto maior o valor para o TTL, maior será a sobrecarga (como veremos a seguir). Pode-se perceber ainda, nas figuras 6.1(e) e 6.1(f), que há uma pequena queda na taxa de cobertura do PeerDB para os cenários com TTL = 10 em relação aos cenários com TTL = 8. Essa redução na taxa de cobertura é justificada pelo uso de diferentes configurações do ambiente em cada cenário, o que significa que as execuções foram condizidas com diferentes *workloads*.

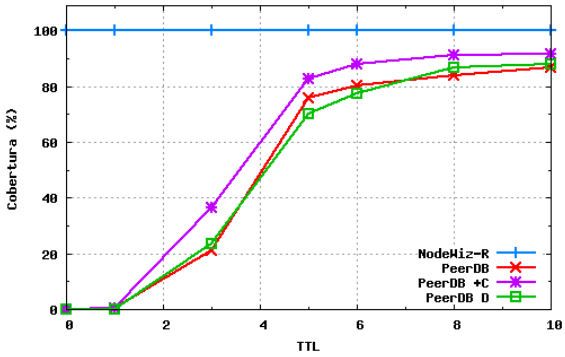
Um outro ponto importante a se observar é que, mesmo com o incremento no tamanho da rede, o NodeWiz-R ainda consegue obter uma cobertura de 100%, enquanto PeerDB apresenta uma taxa de cobertura sempre inferior, com desempenho um pouco melhor para redes menores, como as com 500 e 1.000 *peers*. Os diferentes tipos de consultas ou o número de atributos não exercem qualquer tipo de influência para os resultados apresentados pelo simulador do PeerDB, visto que nenhum roteamento é feito baseado nos predicados contidos nas consultas. Já o NodeWiz-R faz uso direto dos predicados contidos nas consultas, usados



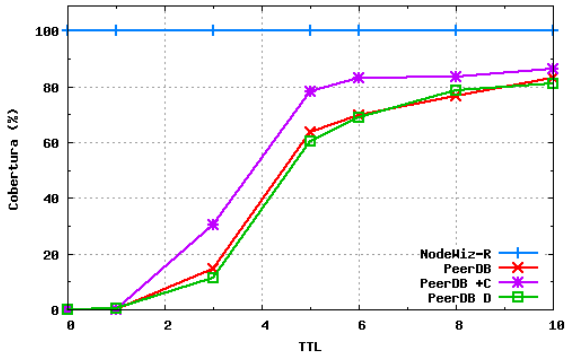
(a) Consultas com casamento exato de valores sobre 2 atributos da tabela principal em uma rede com 500 peers (cenário 1)



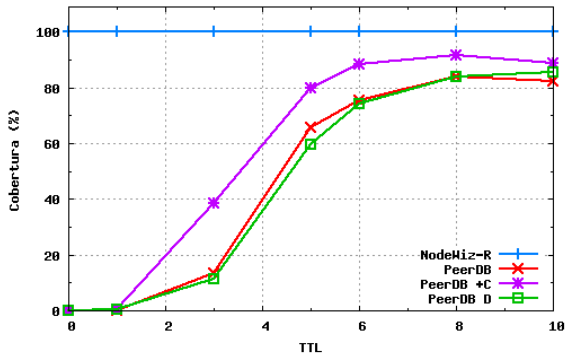
(b) Consultas por faixa de valores sobre 1 atributo da tabela principal em uma rede com 1.000 peers (cenário 2)



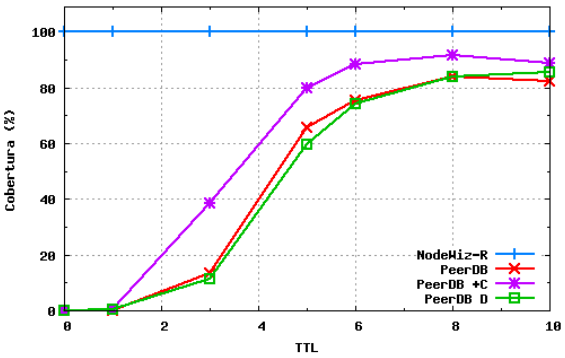
(c) Consultas com casamento exato de valores sobre 2 atributos da tabela secundária em uma rede com 3.000 peers (cenário 3)



(d) Consultas por faixa de valores sobre 2 atributos da tabela secundária em uma rede com 5.000 peers (cenário 4)



(e) Consultas por faixa de valores sobre 1 atributo da tabela secundária em uma rede com 10.000 peers (cenário 5)



(f) Consultas por faixa de valores sobre 2 atributos da tabela secundária em uma rede com 10.000 peers (cenário 6)

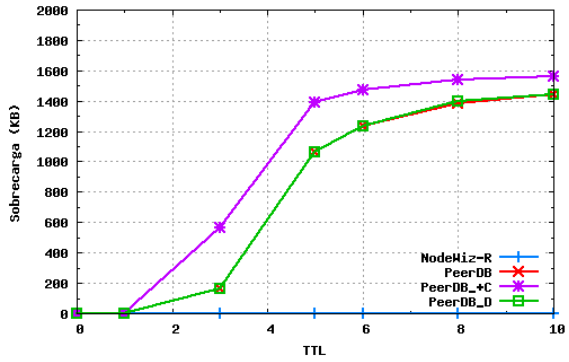
Figura 6.1: Taxa de Cobertura x TTL

na expressão de roteamento. Por isso, o NodeWiz-R pode ter desempenho diferente diante de diferentes tipos de consultas e da quantidade de atributos nelas contidas. Mesmo assim, para todos os cenários, os resultados esperados foram obtidos, independente do tipo e da quantidade de atributos contidos nas consultas. Os resultados para todos os outros cenários cujos os gráficos não são apresentados foram semelhantes, com variações mínimas para o PeerDB e por isso seus gráficos não foram exibidos. O NodeWiz-R consegue obter 100% de cobertura em todos os cenários simulados.

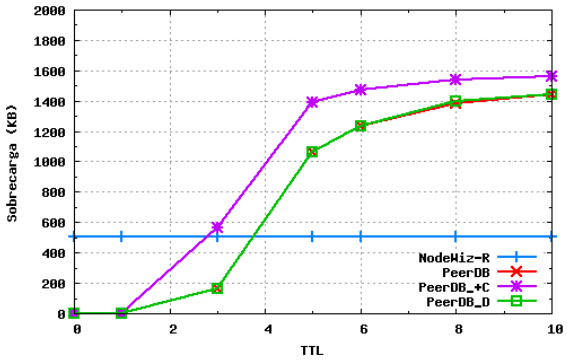
Sobrecarga x TTL

A Figura 6.2 mostra a sobrecarga gerada versus o TTL usado para se alcançar a taxa de cobertura apresentada anteriormente. Embora PeerDB (em todos os modos de operação) tenha alcançado uma taxa de cobertura entre 80% e 90%, a quantidade de informação que os processos distribuídos precisam trocar para alcançar esse resultado foi substancial. Podemos observar que, para os cenários onde consultas com casamento exato de valores são executadas, representadas nas figuras 6.2(a) e 6.2(c), a sobrecarga gerada por NodeWiz-R foi mínima em relação à sobrecarga gerada por PeerDB. Isso se deve ao fato de uma menor quantidade de nós serem contatados e, conseqüentemente, uma menor quantidade de mensagens serem trocadas. Esse comportamento já era esperado, uma vez que no NodeWiz-R apenas um seleto número de nós são escolhidos para responder uma consulta. Além disso, quanto mais seletiva for a consulta, mais eficiente será o roteamento da mesma. No caso de consultas que envolvem apenas casamento exato de valores, que são altamente seletivas, poucos nós devem respondê-las. Por exemplo, uma consulta que inclua o predicado $Att_1 = 100$ só deve ser respondida pelo *peer* que é responsável pelo subespaço de atributos que inclui o valor 100 para o atributo Att_1.

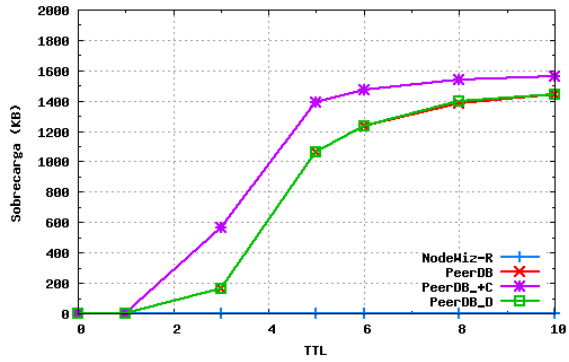
Por outro lado, como podemos observar nas Figuras 6.2(b), 6.2(e) e 6.2(f), a sobrecarga gerada pelo NodeWiz-R para consultas por faixa, é bem mais elevada em relação às consultas com casamento exato de valores, dos cenários 1 e 3 (Figura 6.2(a) e 6.2(c)). Nesse caso a seletividade das consultas deve ser levada em consideração para que possamos comparar os resultados obtidos com os resultados anteriores. A seletividade é um dos conceitos fundamentais em otimização de consultas [42]. A seletividade de uma consulta é determinada basicamente pela quantidade de respostas que são esperadas como resultado de seu proces-



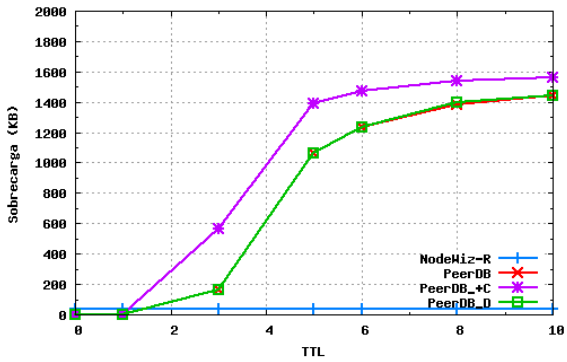
(a) Consultas com casamento exato de valores sobre 2 atributos da tabela principal em uma rede com 10.000 peers (cenário1)



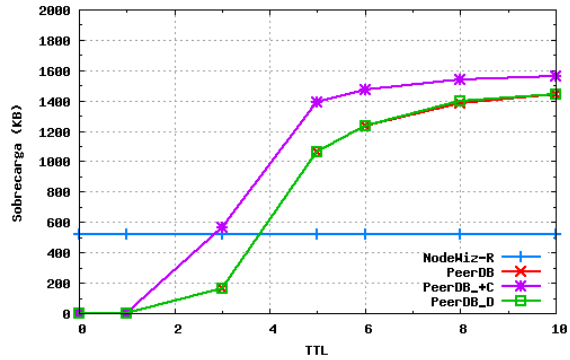
(b) Consultas por faixa de valores sobre 1 atributo da tabela principal em uma rede com 10.000 peers (cenário2)



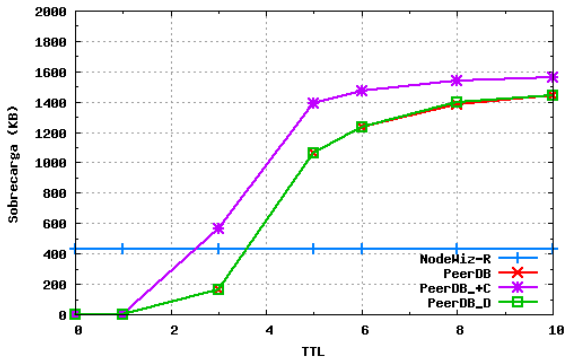
(c) Consultas com casamento exato de valores sobre 2 atributos da tabela secundária em uma rede com 10.000 peers (cenário3)



(d) Consultas por faixa de valores sobre 2 atributos da tabela secundária em uma rede com 10.000 peers (cenário4)



(e) Consultas por faixa de valores sobre 1 atributo da tabela secundária em uma rede com 10.000 peers (cenário5)



(f) Consultas por faixa de valores sobre 2 atributos da tabela secundária em uma rede com 10.000 peers (cenário6)

Figura 6.2: Sobrecarga x TTL

samento. Em sistemas de bancos de dados, ela é determinada pelo número de tuplas que satisfazem um predicado. Um predicado dentro de uma consulta é mais seletivo se ele é satisfeito por um menor número de tuplas. Portanto, uma consulta é mais seletiva se existem poucas tuplas que casam com seus predicados [42].

Uma consulta por faixa é, geralmente, menos seletiva do que uma consulta com casamento exato de valores, pois um maior número de resultados pode satisfazer uma consulta desse tipo. Por exemplo, se trocarmos o operador de igualdade do predicado exemplificado anteriormente pelo operador de faixa $>$, aumenta-se substancialmente o número de subespaços que podem satisfazer tal consulta, e todos os *peers* que mantêm esses subespaços devem ser consultados. Como os valores para as consultas são escolhidos uniformemente entre valores positivos e negativos, a escolha de valores negativos junto com o operador "$>$" usado nas consultas por faixa, pode ter resultado em uma grande faixa de valores a ser selecionada. Isso justifica o aumento na quantidade de mensagens trocadas para se alcançar a cobertura apresentada anteriormente, para consultas por faixa de valores. Embora essas não sejam consultas típicas (faixas tão abrangentes) ou um cenário freqüente no mundo real, elas podem ocorrer e o resultado pode ser bem próximo ao que estamos apresentando nas simulações.

Faixas de valores largas representam portanto baixa seletividade, uma vez que um número maior de resultados pode combinar com a faixa especificada. Assim, o número de nós contatados pode aumentar de acordo com a faixa de valores requerida na consulta.

O problema relacionado a baixa seletividade de consultas é encontrado em várias soluções distribuídas de roteamento/processamento de consultas, até mesmo nos sistemas baseados em DHT. Por exemplo, para que uma DHT possa resolver uma consulta do tipo `SELECT * FROM Table1`, todos ou grande parte dos nós terão que ser consultados. Isso porque todos os dados da tabela `Table1` são requisitados. Como em mecanismos baseados em DHT, os atributos são usados para gerar as chaves para a distribuição e localização das informações na DHT, seria necessário uma consulta por todos os *peers* já que nenhuma informação para geração da chave de consulta pode ser extraído dessa consulta.

É importante perceber também que, a forma como os dados estão relacionados pode influenciar na quantidade de nós que precisam ser contatados para satisfazer consultas por tabelas secundárias. Com muitas informações relacionadas e com relacionamentos não-exclusivos

(*e.g.,* uma mesma *tag* associada a vários recursos distintos), as informações podem tornar-se mais fragmentadas e portanto, mais *peers* devem ser consultados para satisfazer uma determinada consulta. Além disso, o NodeWiz-R replica as informações de tabelas secundárias para evitar que *joins* distribuídos sejam realizados para satisfazer consultas mais complexas localmente. Essa replicação também ocasiona uma dispersão maior das informações no catálogo. Com isso, consultas por tabelas secundárias podem exigir que muitos nós sejam contatados para satisfazê-la plenamente. Entretanto, não há um aumento exponencial no número de mensagens requeridas para o processamento das operações, uma vez que o envio de mensagens é feito de forma controlada pelos índices que roteiam as operações apenas para os *peers* responsáveis pelo subespaço requerido ou que mantêm as informações de tabela secundária requerida em uma consulta. Além disso, se várias tuplas na tabela-índice, que casam com uma consulta, apontam para um mesmo *peer*, a operação de consulta será roteada uma única vez para aquele *peer*, evitando que sucessivas mensagens desnecessárias sejam enviadas. Ao contrário de técnicas de *flooding* que permitem que *loops* ocorram e que as buscas continuem até mesmo depois que as informações requeridas já foram encontradas. Além disso, técnicas de *flooding* são ineficientes para a localização de informações que não estão altamente replicadas sobre a rede [54].

Outro ponto a ser considerado é que consultas por tabelas secundárias isoladas geralmente são incomuns. As consultas mais comuns envolvem a tabela principal, que contém os principais atributos que descrevem um recurso, e estas são roteadas eficientemente pelo índice primário. Portanto, em consultas típicas para a descoberta de um recurso no NodeWiz-R poucos *peers* são consultados.

A tabela 6.3 apresenta a quantidade média de *peers* contatados em todos os simuladores, em cada cenário, para uma rede com 10.000 nós e TTL igual a 10. O erro máximo, para um nível de confiança de 95% para a média, foi de 3%.

Podemos observar que para os cenários 1 e 3 o número de *peers* contatados pelo NodeWiz-R é realmente inferior. Já para o cenário 2, onde consultas por faixa de valores sobre um atributo da tabela principal são realizadas, houve um aumento expressivo no número de *peers* contatados em relação ao cenário 1, confirmando a hipótese de que consultas menos seletivas requerem que um maior número de *peers* sejam visitados para que todos os resultados sejam obtidos. Embora o número de nós contatados não seja o melhor indica-

Cenário/Simulador	NodeWiz-R	PeerDB	PeerDB D	PeerDB +C
Cenário 1	32	8580	8712	9279
Cenário 2	5152	8540	8695	9255
Cenário 3	18	8562	8703	9260
Cenário 4	405	8583	8708	9270
Cenário 5	5367	8590	8715	9272
Cenário 6	4388	8579	8707	9269

Tabela 6.3: Média de peers contatados para rede com 10.000 peers e TTL=10

dor para se afirmar a eficiência do mecanismo de indexação, as simulações mostram que o NodeWiz-R é mais eficiente também nesse critério, já que uma quantidade inferior de nós são contatados. De modo geral, através das simulações podemos concluir que o mecanismo de indexação do NodeWiz-R é mais eficiente que o uso de técnicas de *flooding* usada pelas soluções que fornecem suporte a consultas complexas expressas na linguagem SQL, para a resolução de consultas em ambientes amplamente distribuídos.

6.2 Validação das simulações

Para validarmos as simulações, comparamos os resultados de alguns cenários simulados com o protótipo da solução. Isso foi feito uma vez que, apesar das vantagens de se usar simulações, falhas podem ocorrer desde a execução até mesmo no código do simulador implementado. Portanto, os experimentos com o protótipo da solução podem esclarecer dúvidas remanescentes ou até mesmo apresentar divergências a serem esclarecidas, e que podem motivar mais estudos e otimizações para a solução proposta.

6.2.1 Execução de experimentos

Os experimentos com o protótipo do NodeWiz-R foram conduzidos em um ambiente controlado, na rede local do Laboratório de Sistemas Distribuídos (LSD), da Universidade Federal de Campina Grande (UFCG). As máquina eram dedicadas e tinham configurações semelhantes, grande parte configuradas com Processador Intel Pentium IV de 2.0 GHz, com 1 gigabyte

de memória RAM. Uma máquina também foi dedicada para hospedar o servidor Jabber. A implementação do servidor Jabber utilizada foi o *Openfire*[1] (v3.6.4). Um pré-requisito para a execução dos experimentos era ter a máquina virtual Java na versão 1.6 instalada, uma vez que o protótipo do NodeWiz-R foi desenvolvido sobre essa versão do Java. Como o tempo de resposta não foi avaliado nesse experimento, desconsiderou-se o tráfego na rede no horário em que os experimentos foram conduzidos.

Uma aplicação cliente foi desenvolvida para interagir com os *peers* do sistema. Ela foi usada para invocar *scripts* que faziam o *deployment* do sistema, submetiam informações e consultas, coletavam os *logs* e geravam o *log* sintético de cada execução para análise posterior. O *layout* e as informações contidas nos *logs* do experimento foram os mesmos contidos nos *logs* do simulador, o que possibilitou o reuso dos scripts de análise feitas para o simulador.

Foram utilizadas ao todo 25 máquinas para configurar redes NodeWiz-R com 64 e 128 *peers*. Vários *peers* foram instalados em cada uma das 25 máquinas utilizadas. Por questões de limitação de hardware, não foi possível experimentar redes com quantidades maiores de *peers*. Houve uma redução das *workloads* de 100.000 para 30.000 informações e de consultas de 1.000 para 100. O processo de publicação das informações na rede também seguiu a mesma lógica feita com o simulador. Um único *peer* foi iniciado e todas as informações foram publicadas no mesmo. Após a publicação das informações, o restante dos *peers* foram adicionados, fazendo assim a divisão do espaço de atributos e balanceamento do sistema. Em seguida, as consultas foram enviadas para *peers* escolhidos aleatoriamente.

6.2.2 Análise das execuções

Os resultados obtidos para a métrica cobertura foram os mesmos apresentados pelo simulador, alcançando 100% dos resultados esperados em todos os cenários. Já para a métrica sobrecarga, os resultados também foram satisfatórios, com o máximo de 5,96 *Kbytes* transmitidos para a rede com 64 *peers* e 11,13 *Kbytes* para a rede com 128 *peers*, para satisfazer uma operação de consulta.

A figura 6.3 mostra o percentual de *peers* contatados nas simulações versus experimentos. Os valores são médias de vários experimentos com nível de confiança de 95% e um erro

[1] http://www.igniterealtime.org/projects/openfire

máximo de 7,39%. A tabela 6.4 apresenta os intervalos de confiança para as simulações e experimentos executados. Apesar de ter sido evidenciado diferenças nos resultados de alguns cenários, foi comprovado, utilizando procedimentos estatísticos (método *t*-**test**) [36] comparando os intervalos apresentados, que na maioria dos cenários os sistemas são indistinguíveis, e que em outros cenários os sistemas não apresentam diferenças significativas.

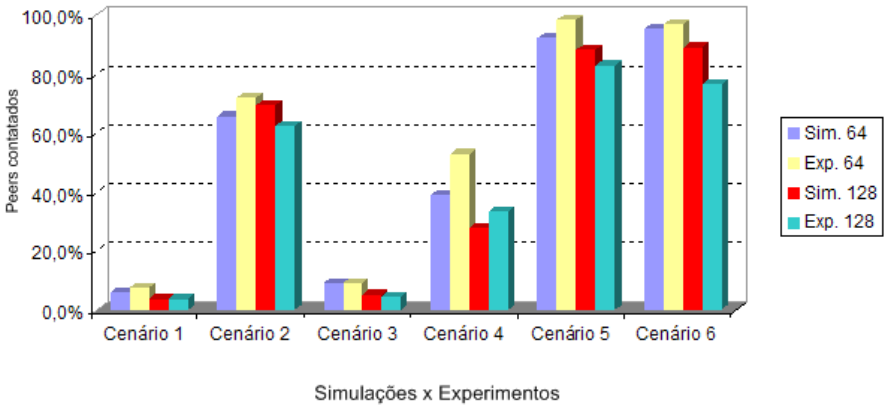


Figura 6.3: Percentual de *peers* contatados nas simulações x experimentos

	Sim. 64	**Exp. 64**	**Sim. 128**	**Exp. 128**
Cenário 1	3,23 - 5,43	5,32 - 6,14	5,64 - 6,44	5,66 - 6,41
Cenário 2	40,88 - 44,79	44,64 - 48,83	86,42 - 92,72	77,12 - 84,61
Cenário 3	6,48 - 7,41	6,15 - 6,66	7,70 - 8,93	6,29 - 6,86
Cenário 4	25,33 - 26,75	34,03 - 36,07	35,76 - 37,96	43,11 - 46,11
Cenário 5	59,17 - 60,34	63,61 - 64,03	112,48 - 115,14	104,08 - 109,33
Cenário 6	56,00 - 57,81	62,63 - 63,71	102,33 - 106,32	94,78 - 102,65

Tabela 6.4: Intervalo de confiança de 95% para o número médio de *peers* contatados

É importante observar na figura 6.3, que houve um aumento no percentual médio de *peers* contatados se comparado com as simulações anteriores para redes maiores. Houve um incremento nos cenários onde consultas por faixa de valores foram executadas, principalmente sobre tabelas secundárias (cenários 5 e 6). Esse resultado pode ser justificado pela cópia dos dados das tabelas secundárias em diversos *peers*, abrangendo grande parte da rede.

Em uma rede com uma quantidade pequena de nós, cada *peer* ficará responsável por uma grande massa de informações sobre recursos, e com isso os dados de tabelas secundárias, quando não-exclusivos, podem aparecer replicados em grande parte dos *peers* da rede. Portanto, é inevitável que todos os *peers* sejam consultados se as informações requisitadas se encontram em todos eles. Podemos observar que, no geral os resultados apresentados para a rede com 128 *peers* foi melhor do que com 64 *peers*, o que reforça essa hipótese.

Adicionalmente, para comprovar que a seletividade das consultas por faixa de valores exerce grande influência no desempenho do sistema, foram executadas consultas com uma faixa menos abrangente tanto sobre a tabela principal, como sobre a tabela secundária. Houve uma redução considerável no número de *peers* contatados para ambas as consultas, com 31,25% e 24,22% para consultas sobre a tabela principal nas redes com 64 e 128 *peers* respectivamente, e 64,06% e 50,78 % para consultas sobre a tabela secundária.

6.3 Conclusão

As simulações e experimentos mostraram que o NodeWiz-R apresenta-se como uma melhor solução em relação ao PeerDB para as métricas analisadas, e de modo geral aos sistemas que utilizam técnicas de *flooding* para disseminar consultas mais elaboradas expressas em SQL. O roteamento de requisições constitui-se um enorme aprimoramento dos sistemas P2P de modo geral comparado com a técnica de inundação. Em um ambiente livre de falhas, o NodeWiz-R sempre encontrou todas as informações existentes, ou seja apresentou uma eficiência de 100% para a métrica cobertura. Esse resultado foi obtido com uma baixa sobrecarga, principalmente para consultas mais seletivas, independente da tabela do esquema consultada. Para consultas com casamento exato de valores utilizando atributos da tabela principal e com 2 atributos da tabela secundária (cenários 1 e 3 respectivamente), o número de nós consultados é aproximado a uma função logarítmica do número de *peers* do sistema. Conseguiu-se também mostrar que mesmo dados replicados sobre a rede podem ser localizados de forma eficiente. Embora o número de *peers* contatados possa ser elevado, apenas os *peers* que mantém os resultados realizam o processamento das consultas. Com isso, fica mais evidente que buscas baseadas em redes P2P estruturadas são mais eficien-

tes, uma vez que apenas um seleto número de *peers* precisam ser consultados. De modo geral, os resultados obtidos compensam a quantidade de informações mantidas por cada *peer*, já que a escalabilidade, autonomia e expressividade do sistema é garantida, sem que sua eficiência seja comprometida. Os experimentos executados validaram as simulações, uma vez que os resultados apresentados nas simulações foram reproduzidas nos experimentos com o protótipo da solução implementado, apresentando apenas pequenas variações nas médias de *peers* contatados para alguns cenários, mas comprovado estatisticamente que os sistemas não são diferentes.

Capítulo 7

Conclusões e Trabalhos Futuros

Este trabalho apresenta o NodeWiz-R, um sistema P2P relacional para a descoberta de recursos, que permite indexar eficientemente recursos cujo modelo de descrição requer um nível maior de complexidade e não podem ser representados fielmente utilizando um modelo de descrição baseado em simples pares de atributo-valor. A solução propõe o uso de um substrato P2P sobre uma camada relacional de forma a se obter autonomia do sistema e expressividade na descrição dos recursos como também nas consultas. O NodeWiz-R provê o suporte a um subconjunto da linguagem SQL sobre o substrato P2P, possibilitando que consultas mais expressivas sejam realizadas. Seu mecanismo de indexação, baseado em uma estrutura de árvore de distribuição e com vários níveis de indexação, permite que o roteamento das operações de consulta e atualização sejam feitas de forma eficiente, contatando apenas os *peers* que potencialmente são responsáveis por manter as informações referidas na operação. A utilização de vários níveis de indexação também evita o uso de técnicas de inundação, frequentemente utilizadas por outras soluções do gênero, que apesar de ser de simples implementação, gera um *overhead* indesejado na rede.

O NodeWiz-R utiliza uma abordagem de armazenamento baseada em atualização de estado fraca, onde as informações são mantidas no índice temporariamente, criando uma espécie de coletor de lixo automático para informações desatualizadas. Essa abordagem garante uma fácil manutenção das informações no catálogo e possibilita lidar melhor com recursos intermitentes, cuja atualização de estado é freqüente ou que podem entrar e sair do sistema sem aviso prévio. Seu desempenho é garantido com uma técnica de balanceamento da carga de trabalho, onde os *peers* mais sobrecarregados dividem seu espaço de atributos

com novos *peers*. Além disso, ele é autogerenciável, não requerendo esforços administrativos para que se adicione ou remova um *peer* do sistema, ou para mantê-lo operacional.

A solução foi avaliada usando um modelo simulado. Foram desenvolvidos dois simuladores: um para o NodeWiz-R e outro para a solução PeerDB, usada na comparação. As métricas escolhidas para serem avaliadas foram a cobertura, medida pela razão entre a quantidade de respostas retornadas e a quantidade esperada, e a sobrecarga, medida como a quantidade de informação trocada pelo sistema para resolver uma operação de consulta. As simulações foram realizadas em redes lógicas de diferentes tamanhos e com variados tipos de consulta. Em todos os cenários observou-se um melhor desempenho do NodeWiz-R frente ao PeerDB, perfazendo uma taxa de retorno de 100% contra uma taxa variante de 85% a 95% do PeerDB. Como era esperado, o NodeWiz-R obteve os resultados com uma baixa sobrecarga, ao contrário do PeerDB que para obter seus resultados gerou uma grande quantidade de mensagens, sobrecarregando a rede. No caso de consultas com casamento exato de valores, os resultados mostraram que o NodeWiz-R é bastante eficiente, apresentando uma sobrecarga baixa e constante, independente do tamanho da rede e da tabela consultada.

Foi possível observar também que o NodeWiz-R pode requerer que muitos *peers* sejam consultados quando o tipo de consulta submetido possui baixa seletividade. A elaboração de consultas com baixa seletividade é um problema que pode levar a um baixo desempenho do sistema, com a elevação do tempo de resposta das consultas, diminuindo assim a confiança dos usuários em relação ao sistema. Embora o NodeWiz-R possa indexar de forma eficiente todas as tabelas de um esquema, esse problema não pode ser resolvido a nível de indexação. Pelo mesmo motivo, alguns sistemas de banco de dados implementam mecanismos para otimizar suas as requisições. Basicamente, o mecanismo analisa as restrições após a cláusula *WHERE* de uma consulta SQL, e verifica qual predicado retorna uma menor quantidade de resultados. A consulta é reescrita apenas com esse predicado e é executada. Os resultados obtidos são usados para serem processados com os demais predicados contidos na consulta. Apenas os resultados que casam com todos os predicados são retornados [42]. Portanto, a otimização prévia das requisições pode ser uma solução para melhorar o desempenho do NodeWiz-R, nos casos onde as consultas apresentam baixa seletividade. Não obstante, a otimização de consultas é um tópico complexo no contexto de processamento de consultas e vários algoritmos foram propostos com essa finalidade. Portanto, um estudo mais aprofun-

dado deve ser realizado para que se possa concluir qual algoritmo é mais adequado e se os resultados obtidos após a implementação trazem ganhos expressivos para a solução.

Com o objetivo de aumentar a confiança dos resultados obtidos através das simulações, foi desenvolvido um protótipo da solução e com ele realizado experimentos. Os experimentos foram conduzidos em um ambiente controlado para que as mesmas condições simuladas fossem alcançadas no ambiente real. Entretanto, pôde-se implementar apenas redes com 64 e 128 *peers*, por questões de limitação de número de máquinas e de configurações de hardware. Os resultados do experimento mostraram que o protótipo possui a mesma eficiência apresentada no simulador, obtendo 100% dos resultados esperados nas consultas. A sobrecarga produzida pelo protótipo do NodeWiz-R também foi mínima, comprovando assim a eficiência dos índices do sistema.

Atualmente, está sendo desenvolvida a primeira versão totalmente funcional do NodeWiz-R, reimplementando todos os módulos do sistema, já que para o protótipo, foram reusados e feitas adaptações no código do NW. Além disso, como foi dito no capítulo 5, um mecanismo de tolerância a falhas pode ser implementado no sistema, uma vez que a interface de comunicação *Commune* provê um detector de falhas embarcado que facilita a sua implementação. Esse mecanismo permitirá a restauração da árvore de roteamento do NodeWiz-R, evitando assim que o sistema fique em um estado inconsistente. Uma simplificação no código também deve ser feita, onde estruturas de dados que são armazenadas na memória principal, como por exemplo os índices primários e secundários, serão persistidos no próprio banco de dados, simplificando as operações internas de consulta nos índices e o sistema de *matching*. Grande parte do processamento feito no código do NodeWiz-R seria transferido portanto para o banco de dados. O algoritmo *K-Means* também pode ter seu processamento transferido direto para o banco de dados, evitando assim que os dados tenham que ser recuperados no banco e posteriormente reprocessados pelo algoritmo externamente.

Ainda sobre a implementação, apesar do NodeWiz-R dar suporte a consultas expressas em SQL, alguns operadores da linguagem ainda não são suportados, como o *LIKE* e *BETWEEN*. Essa é uma limitação do *SQL-Parser* que pode ser resolvida facilmente, convertendo esses operadores em predicados especificando faixas de valores. O mecanismo que permite agregar os valores de consultas com operações de agregação, antes de serem

entregues ao cliente, também pode ser implementado e tornar-se de uso opcional, já que alguns sistemas requerem o retorno imediato dos resultados e essa solução retarda o envio das respostas ao cliente até que todos os resultados sejam computados. Outra limitação relacionada a sua implementação diz respeito ao suporte para apenas um único esquema. Por isso, recursos com uma estrutura de descrição diferente da utilizada pela rede não podem ser publicados. De forma a mitigar esse problema, é necessário que o NodeWiz-R dê suporte à múltiplos esquemas simultaneamente, permitindo assim que recursos com estruturas de descrição diferentes possam ser indexados por uma mesma rede NodeWiz-R já implantada.

Um problema da solução de indexação do NodeWiz-R é o armazenamento das informações das tabelas secundárias também nas suas respectivas tabelas-índice. Se muitas informações são publicadas em tabelas secundárias, suas tabelas-índice também armazenaram uma quantidade substancial de dados. Uma possível solução seria usar as técnicas da modelagem relacional para criar apenas relacionamentos entre essas tabelas, evitando assim que dados das tabelas secundárias fossem copiados. Entretanto, essa solução não é aplicável uma vez que as tabelas-índice normalmente não residem no mesmo *peer* onde estão todos os dados da tabela secundária a qual ela se refere. Portanto esse pode ser considerado um problema em aberto da solução apresentada, requerendo um estudo mais aprofundado para que seja resolvido futuramente.

Com relação a avaliação da solução, a obtenção de uma *workload* real também pode melhorar a qualidade do trabalho, trazendo resultados mais próximos da realidade. A sobrecarga na publicação das informações no catálogo também deve ser avaliada bem como o tempo de resposta (*round-trip time*) das consultas.

O NodeWiz-R resolve um problema prático, anunciado pela comunidade científica e evidenciado pelo projeto SegHidro, onde dados ambientais que possuem uma estrutura complexa não poderiam ser representados, indexados e consultados de forma eficiente pelos atuais mecanismos de descoberta de recursos distribuídos existentes.

Pretende-se utilizar o NodeWiz-R para melhorar o sistema de *match-making* na comunidade do OurGrid, além de utilizá-lo como serviço de localização de dados ambientais no projeto SegHidro, substituindo o NW e o SGBD centralizado que é utilizado na ferramenta de localização de dados ambientais OPeNDAP Meta Finder[1]. O NodeWiz-R também será

[1] http://opendapmetafinder.lsd.ufcg.edu.br

utilizado em um trabalho de Mestrado em andamento na Universidade Federal de Campina Grande, atuando como catálogo distribuído em uma solução de armazenamento de dados distribuída para a execução de aplicações *data intensive* no OurGrid. Pretende-se também ao fim desse trabalho, publicar os resultados dessa pesquisa na comunidade científica, em uma conferência a ser definida.

Bibliografia

[1] Gnutella. Gnutella Development Home Page, Disponível em http://gnutella.com, Acessado em Junho 2009.

[2] JSF. Jabber software foundation web site. Disponível em http://www.jabber.org, Acessado em Julho 2009.

[3] Kazaa. The Kazaa Homepage, Disponível em http://www.kazaa.com, Acessado em Junho 2009.

[4] Napster. The Napster Homepage, Disponível em http://free.napster.com/, Acessado em Junho 2009.

[5] XMPP. Extensible Messaging and Presence Protocol Specification. Disponível em http://www.xmpp.org/specs/, Acessado em Junho 2009.

[6] ARAÚJO, E., CIRNE, W., WAGNER, G., OLIVEIRA, N., SOUZA, E., GALVÃO, C., AND MARTINS, E. The SegHidro Experience: Using the Grid to Empower a Hydro-Meteorological Scientific Network. *Proceedings of* 1^{st} *IEEE Conference on e-Science and Grid Computing* (2005), 64–71.

[7] BARREN, M., HELLERSTEIN, J., HUEBSCH, R., LOO, B., SHENKER, S., AND STOICA, I. Complex Queries in DHT-based Peer-to-Peer Networks. *Peer-To-Peer Systems: First International Workshop, IPTPS 2002, Cambridge, MA, USA, March 7-8, 2002: Revised Papers* (2002).

[8] BASU, S., BANERJEE, S., SHARMA, P., AND LEE, S. NodeWiz: Peer-to-peer Resource Discovery for Grids. *Proceedings of 5th International Workshop on Global and Peer-to-Peer Computing (in conjunction with CCGRID 2005)* (May 2005), 213–220.

[9] BASU, S., COSTA, L., BRASILEIRO, F., BANERJEE, S., SHARMA, P., AND LEE, S.-J. NodeWiz: Fault-tolerant Grid Information Service. *Peer-to-Peer Networking and Applications* (2009).

[10] BHARAMBE, A., AGRAWAL, M., AND SESHAN, S. Mercury: Supporting Scalable Multi-Attribute Range Queries. *Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications* (2004), 353–366.

[11] BLANNING, R. W. Data Management and Model Management: a Relational Synthesis. In *ACM-SE 20: Proceedings of the 20th annual Southeast regional conference* (New York, NY, USA, 1982), ACM, pp. 139–147.

[12] BONCZ, P., AND TREIJTEL, C. AmbientDB: Relational Query Processing in a P2P Network. *Databases, Information Systems, and Peer-To-Peer Computing: First International Workshop, DBISP2P 2003: Berlin, Germany, September 7-8, 2003: Revised Papers* (2004).

[13] BONIFATI, A., CHRYSANTHIS, P. K., OUKSEL, A. M., AND SATTLER, K.-U. Distributed Databases and Peer-to-Peer Databases: Past and Present. *SIGMOD Rec. 37*, 1 (2008), 5–11.

[14] BOUGUETTAYA, A., BENATALLAH, B., AND ELMAGARMID, A. An Overview of Multidatabase Systems: Past and Present. *Management of heterogeneous and autonomous database systems* (1999), 1–31.

[15] BRASILEIRO, F., COSTA, L. B., ANDRADE, A., CIRNE, W., BASU, S., AND BANERJEE, S. A Large Scale Fault-tolerant Grid Information Service. In *MCG '06: Proceedings of the 4th international workshop on Middleware for grid computing* (New York, NY, USA, 2006), ACM, p. 14.

[16] BRICKLEY, D., AND GUHA, R. RDF Vocabulary Description Language 1.0: RDF Schema. *W3C Recommendation 10* (2004).

[17] BRUNKHORST, I., DHRAIEF, H., KEMPER, A., NEJDL, W., AND WIESNER, C. Distributed Queries and Query Optimization in Schema-Based P2P-Systems. *Lecture Notes in Computer Science* (2004), 184–199.

[18] CAI, M., FRANK, M., CHEN, J., AND SZEKELY, P. MAAN: A Multi-Attribute Addressable Network for Grid Information Services. In *Proceedings of the* 4^{th} *Int. Workshop on Grid Computing (GRID 2003)* (2003), pp. 184–191.

[19] CARZANIGA, A., D., R., AND A., W. Achieving Expressiveness and Scalability in an Internet-Scale Event Notification Service. In *In Proceedings of the* 19^{th} *ACM Symposium on Principles of Distributed Computing* (Portland, OR, USA, July 2000).

[20] CIRNE, W., BRASILEIRO, F., ANDRADE, N., COSTA, L., ANDRADE, A., NOVAES, R., AND MOWBRAY, M. Labs of the World, Unite!!! *Journal of Grid Computing 4*, 3 (2006), 225–246.

[21] CLARK, D. The Design Philosophy of the DARPA Internet Protocols. *ACM SIGCOMM Computer Communication Review 18*, 4 (1988), 106–114.

[22] CODD, E. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM, v.13 n.6* (June 1970), 377–387.

[23] CRAINICEANU, A., LINGA, P., GEHRKE, J., AND SHANMUGASUNDARAM, J. Querying Peer-to-Peer Networks Using P-Trees. In *WebDB '04: Proceedings of the* 7^{th} *International Workshop on the Web and Databases* (New York, NY, USA, 2004), ACM, pp. 25–30.

[24] CURTIS, D. Java, RMI and Corba. *The Object Management Group* (1997).

[25] D., S., AND T., H. XenoSearch: Distributed Resource Discovery in the XenoServer Open Platform. In *Proceedings of the* 20^{th} *IEEE Int. Symposium on High Performance Distributed Computing (HPDC-12)* (2003), pp. 216–225.

[26] DASWANI, N., GARCIA-MOLINA, H., AND YANG, B. Open Problems in Data-Sharing Peer-to-Peer Systems. *Proceedings of the* 9^{th} *International Conference on Database Theory* (2003), 1–15.

[27] DOUGLAS COMER. The ubiquitous B-tree. *ACM Computing Surveys 11* (1979), 121–137.

[28] FOSTER, I., AND KESSELMAN, C. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2004.

[29] GRAEFE, GOETZ. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv. 25*, 2 (1993), 73–169.

[30] HAN, J., AND KAMBER, M. *Data Mining: Concepts and Techniques, Chap. 8: Cluster Analysis.* Morgan Kaufmann, 2006.

[31] HARCHOL-BALTER, M., LEIGHTON, T., AND LEWIN, D. Resource Discovery in Distributed Networks. In *PODC '99: Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing* (New York, NY, USA, 1999), ACM, pp. 229–237.

[32] HEIMBIGNER, D. Expressive and Efficient Peer-to-Peer Queries. *Hawaii International Conference on System Sciences 9* (2005), 302b.

[33] HEINE, F., HOVESTADT, M., AND KAO, O. Processing Complex RDF Queries Over P2P Networks. In *Proceedings of the 2005 ACM workshop on Information retrieval in peer-to-peer networks* (2005), ACM New York, NY, USA, pp. 41–48.

[34] HUEBSCH, R., CHUN, B., HELLERSTEIN, J., LOO, B., MANIATIS, P., ROSCOE, T., SHENKER, S., STOICA, I., AND YUMEREFENDI, A. The Architecture of PIER: An Internet-Scale Query Processor. *Proc. Conference on Innovative Data Systems Research (CIDR)(Jan. 2005)* (2005).

[35] IAMNITCHI, A., FOSTER, I., AND NURMI, D. C. A Peer-to-Peer Approach to Resource Location in Grid Environments. In *HPDC '02: Proceedings of the* 11^{th} *IEEE International Symposium on High Performance Distributed Computing* (Washington, DC, USA, 2002), p. 419.

[36] JAIN, R. *The Art of Computer Systems Performance Analysis: Techniques For Experimental Measurement, Simulation, And Modeling*. John Wiley & Sons New York, 1991.

[37] JOY, B., STEELE, G., GOSLING, J., AND BRACHA, G. The Java Language Specification. *World Wide Web, http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html, Acessado em Agosto 2009*.

[38] KOSSMANN, D. The State of the Art in Distributed Query Processing. *ACM Computing Surveys (CSUR) 32*, 4 (2000), 422–469.

[39] LIAROU, E., IDREOS, S., AND KOUBARAKIS, M. Continuous RDF Query Processing Over DHTs. In *In Proceedings of* 6^{th} *International Semantic Web Conference (ISWC), 2nd Asian Semantic Web Conference (ASWC)* (2007), pp. 324–339.

[40] LIMA, A., CIRNE, W., BRASILEIRO, F., AND FIREMAN, D. A Case for Event-Driven Distributed Objects. In *Proceedings of the* 8^{th} *International Symposium on Distributed Objects and Applications (DOA)* (October 2006).

[41] LOO, A. The Future Of Peer-to-peer Computing. *Communications of the ACM 46*, 9 (2003), 57.

[42] LYNCH, C. Selectivity Estimation and Query Optimization in Large Databases with Highly Skewed Distributions of Column Values. In *Proceedings of the* 14^{th} *VLDB Conference* (1988), pp. 240–251.

[43] MACKERT, L., AND LOHMAN, G. R* Optimizer Validation and Performance Evaluation for Distributed Queries. *Proceedings of the* 12^{th} *International Conference on Very Large Data Bases table of contents* (1986), 149–159.

[44] MARZOLLA, M., MORDACCHINI, M., AND ORLANDO, S. Resource Discovery in a Dynamic Grid Environment. In *Database and Expert Systems Applications, 2005. Proceedings.* 16^{th} *International Workshop on* (2005), pp. 356–360.

[45] MASTROIANNI, C., TALIA, D., AND VERTA, O. A Super-Peer Model for Building Resource Discovery Services in Grids: Design and Simulation Analysis. *Lecture notes in computer science 3470* (2005), 132.

[46] MIRANDA, R. Providing Security and a Consistent Failure Detection Semantic for Asynchronous Distributed Objects. In *In Proceedings of the* 4^{th} *Latin-American Symposium on Dependable Computing (LADC)* (João Pessoa, PB, Brasil, September 2009).

[47] NG, W., OOI, B., TAN, K., AND ZHOU, A. Peerdb: A P2P-Based System for Distributed Data Sharing. *Proceedings of the* 19^{th} *International Conference on Data Engineering* (2003).

[48] OPPENHEIMER, D., ALBRECHT, J., PATTERSON, D., AND VAHDAT, A. Design and Implementation Tradeoffs for Wide-Area Resource Discovery. In *HPDC '05: Proceedings of the High Performance Distributed Computing, 2005. HPDC-14. Proceedings.* 14^{th} *IEEE International Symposium* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 113–124.

[49] OZSU, M., AND VALDURIEZ, P. Distributed and Parallel Database Systems. *ACM Computing Surveys (CSUR) 28*, 1 (1996), 125–128.

[50] RAMAN, R., LIVNY, M., AND SOLOMON, M. Matchmaking: Distributed Resource Management for High Throughput Computing. *Proceedings of the* 7^{th} *IEEE International Symposium on High Performance Distributed Computing 146*, 10 (1998).

[51] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SCHENKER, S. A Scalable Content-Addressable Network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications* (New York, NY, USA, 2001), ACM, pp. 161–172.

[52] RATNASAMY, S., HELLERSTEIN, J., AND SHENKER, S. Range Queries over DHTs. *IRB-TR-03-009, Intel Corporation* (2003).

[53] RHEA, S., GEELS, D., ROSCOE, T., AND KUBIATOWICZ, J. Handling churn in a DHT. In *Proceedings of the USENIX Technical Conference* (June 2004).

[54] RISSON, J., AND MOORS, T. Survey of Research Towards Robust Peer-to-Peer Networks: Search Methods. *Computer Networks 50*, 17 (2006), 3485–3521.

[55] ROWSTRON, A. I. T., AND DRUSCHEL, P. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg* (London, UK, 2001), Springer-Verlag, pp. 329–350.

[56] SARTIANI, C., MANGHI, P., GHELLI, G., AND CONFORTI, G. XPeer : A Self-Organizing XML P2P Database System. In *Proceedings of the First International Workshop on Peer-to-Peer Computing and Databases, Heraklion, Crete, Greece, March* (2004), vol. 14, Springer.

[57] SHETH, A., AND LARSON, J. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys (CSUR) 22*, 3 (1990), 183–236.

[58] SIDIROURGOS, L., KOKKINIDIS, G., AND DALAMAGAS, T. Efficient Query Routing in RDFS schema-based P2P Systems. In *Proceedings of the* 4^{th} *Hellenic Data Management Symposium (HDMS'05)* (Athens, Greece, 2005).

[59] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications* (New York, NY, USA, 2001), ACM, pp. 149–160.

[60] TAYLOR, C., AND PELLEGRINI, F. The End of Napster As We Know It, Time news service, 2001, Disponível em http://www.time.com/time/nation/article/0,8599,101268,00.html, Acessado em Agosto 2009.

[61] TRIANTAFILLOU, P., AND PITOURA, T. Towards a Unifying Framework for Complex Query Processing Over Structured Peer-to-Peer Data Networks. *Lecture Notes in Computer Science* (2004), 169–183.

[62] TRUNFIO, P., TALIA, D., PAPADAKIS, H., FRAGOPOULOU, P., MORDACCHINI, M., PENNANEN, M., POPOV, K., VLASSOV, V., AND HARIDI, S. Peer-to-Peer Resource

Discovery in Grids: Models and Systems. *Future Generation Computer Systems 23*, 7 (2007), 864–878.

[63] WILSCHUT, A., AND APERS, P. Dataflow Query Execution in a Parallel Main-memory Environment. *Distributed and Parallel Databases 1*, 1 (1993), 103–128.

[64] YANG, B., AND GARCIA-MOLINA, H. Designing a Super-Peer Network. In *Proceedings of the International Conference on Data Engineering* (2003), IEEE Computer Society Press; 1998, pp. 49–62.

Apêndice A

Gramática para a Definição de Esquemas NodeWiz-R

Esta é a gramática para a definição de esquemas no NodeWiz-R. Ela foi criada baseada na gramática para definição de esquemas do SBD *H2*.

SCHEMA DEFINITION

```
SCHEMA DEFINITION schemaName (
   <table definition>
   [...]
   PRIMARY TABLE tableName
   [, SECONDARY TABLE (tableName1 [,tableName2])]
);
```

Define um esquema NodeWiz-R

Primary Table: define a tabela principal do esquema.

Secondary Table: define quais são as tabelas secundárias do esquema que se deseja indexar.

TABLE DEFINITION

```
CREATE TABLE [IF NOT EXISTS] name (
   { name dataType [{AS computedColumnExpression
```

```
      | DEFAULT expression}]
      [[NOT] NULL]
      [AUTO_INCREMENT]
      [PRIMARY KEY | UNIQUE]
    | constraint} [,...]
);
```

Cria uma nova tabela.

Exemplo:

```
CREATE TABLE TEST(
 ID INT PRIMARY KEY,
 NAME VARCHAR(255)
);
```

Name

```
{ { A-Z|_ } [ { A-Z|_|0-9} [...] ] } | quotedName
```

Nomes são case insensitive.

Não existe tamanho máximo.

Exemplo:

RESOURCE_TABLE_01

Expression (computedColumnExpression)

```
andCondition [OR andCondition]
```

```
andCondition ::= condition [AND condition]
```

Valor ou condição.

Exemplo:

ID=1 OR NAME="Hello"

Constraint

```
PRIMARY KEY (columnName [,...])
   | [CONSTRAINT [IF NOT EXISTS] newConstraintName] {
     CHECK expression
     | UNIQUE (columnName [,...])
   | referentialConstraint}
```

Define uma restrição de integridade (constraint).

Exemplo:

PRIMARY KEY(ID, NAME)

Referential Constraint

```
FOREIGN KEY (columnName [,...])
REFERENCES [refTableName] [(refColumnName[,...])]
[ON DELETE {CASCADE | RESTRICT | NO ACTION | SET DEFAULT | SET NULL}]
[ON UPDATE {CASCADE | SET DEFAULT | SET NULL}]
```

Define uma restrição de integridade referencial.

Se o nome da tabela não é especificado, então a mesma tabela é referenciada.

Se as colunas referenciadas não são especificadas, então as colunas que são chaves primárias serão usadas.

Os índices internos do banco de dados serão criados se necessário.

Exemplo:

FOREIGN KEY(ID) REFERENCES RESOURCE_TABLE_01(ID)

Quoted Name

São nomes case sensitive envolto de aspas duplas, e que podem conter espaços.

Não existe tamanho máximo para o nome.

Duas aspas duplas podem ser usadas para criar uma única aspas duplas dentro do nome.

Exemplo:

“João” ou “José Flávio”

Data Type

`intType | booleanType | tinyintType | smallintType | bigintType | identityType | decimalType | doubleType | realType | dateType | timeType | timestampType | binaryType | varcharType | varcharIgnorecaseType | charType blobType | clobType | uuidType | arrayType`

Definição dos tipos de dados que podem ser utilizados para criar campos de uma tabela do esquema.

Tipo de Dado	**Nomes**	**Valores Possíveis**
INT	INT |INTEGER |MEDIUMINT |INT4 |SIGNED	-2147483648 to 2147483647
BOOLEAN	BOOLEAN |BIT |BOOL	TRUE e FALSE
TINYINT	TINYINT	-128 até 127
SMALLINT	SMALLINT |INT2 |YEAR	-32768 até 32767
BIGINT	BIGINT |INT8	-9223372036854775808 até 9223372036854775807
IDENTITY	IDENTITY	-9223372036854775808 até 9223372036854775807
DECIMAL	DECIMAL |NUMBER |DEC |NUMERIC	Tipo de dado com precisão e escala fixada
DOUBLE	DOUBLE [PRECISION] |FLOAT |FLOAT4 |FLOAT8	Número com ponto flutuante.
REAL	REAL	Número com um único ponto flutuante.
TIME	TIME	Hora com o formato hh:mm:ss
DATE	DATE	Data com o formato yyyy-MM-dd
TIMESTAMP	TIMESTAMP |DATETIME |SMALLDATETIME	O formato é yyyy-MM-dd hh:mm:ss
VARCHAR	{VARCHAR |LONGVARCHAR |VARCHAR2 |NVARCHAR |NVARCHAR2 |VARCHAR_CASESENSITIVE} [(precisionInt)]	Unicode String. Não existe precisão máxima. O tamanho máximo é a memória disponível. A precisão é uma restrição de tamanho definida.
VARCHAR_IGNORECASE	VARCHAR_IGNORECASE [(precisionInt)]	O mesmo que VARCHAR, mas não é case sensitive quando comparado.
CHAR	CHAR |CHARACTER |NCHAR [(precisionInt)]	A diferença para VARCHAR é que espaços deixados são ignorados e não são persistidos.
UUID	UUID	Identificador Único Universal. É um valor de 128 bits.

Tabela A.1: Tipos de dados suportados em um esquema NodeWiz-R