PARALELISMO INTRA-CONSULTA EM CLUSTERS DE BANCOS DE DADOS

Alexandre de Assis Bento Lima

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE DOUTOR EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

Profa. Marta Lima de Queirós Mattoso, D.Sc.

Prof. Marco Antônio Casanova, Ph.D.

Prof. Alexandre Plastino de Carvalho, D.Sc.

Prof. Jano Moreira de Souza, Ph.D.

Prof. Nelson Francisco Favilla Ebecken, D.Sc.

RIO DE JANEIRO, RJ - BRASIL

DEZEMBRO DE 2004

LIMA, ALEXANDRE DE ASSIS BENTO

Paralelismo Intra-consulta em Clusters de Bancos de Dados [Rio de Janeiro] 2004

IX, 105 p. 29,7 cm (COPPE/UFRJ, D.Sc., Engenharia de Sistemas e Computação, 2000)

Tese - Universidade Federal do Rio de Janeiro, COPPE

1. Clusters de Bancos de Dados

I. COPPE/UFRJ II. Título (série)

Ao meu avô e professor,

José Luiz Fernandes Lima

AGRADECIMENTOS

Agradeço à minha orientadora, professora Marta Lima de Queirós Mattoso, por toda a sua ajuda, dedicação, atenção e palavras de incentivo, sem os quais esta tese não teria sido concluída. Sua crença no meu potencial, sua paciência e grande visão serviram de estímulo e exemplo, que pretendo seguir em toda a minha carreira.

Agradeço ao meu orientador, professor Patrick Valduriez, que tão bem me recebeu na Université de Nantes e soube ouvir e discutir minhas idéias e propostas, provendo meios para que eu pudesse torná-las realidade. Sua grande experiência, capacidade intelectual e disposição para o trabalho são exemplares para qualquer pesquisador.

Agradeço à minha esposa, Cláudia, pelo seu amor, carinho, dedicação, paciência e compreensão. Durante a elaboração desta tese, houve vários momentos difíceis. Em todos eles, ela soube exatamente o que dizer e fazer para que eu não desistisse. Houve também bons momentos, e ela soube comigo compartilhá-los. É uma companheira de valor inestimável na minha caminhada.

Agradeço aos meus pais, Francisco e Maria, por toda a ajuda, incentivo e carinho demonstrados através de atos e palavras, e pela infinita compreensão nos meus momentos de ausência.

Agradeço à minha irmã e amiga, Luciana. De várias formas e em muitos momentos, ela me ajudou e estimulou com palavras e atitudes.

Agradeço à toda a minha família, por sua compreensão nos momentos em que estive ausente e pelas várias demonstrações de carinho.

Agradeço ao CNPq, à CAPES, ao Programa CAPES/COFECUB, ao Programa de Engenharia de Sistemas e Computação e ao Núcleo de Computação de Alto Desempenho da COPPE/UFRJ, pela disponibilização de recursos utilizados para a elaboração desta tese.

Agradeço ao Atlas Group (INRIA, Université de Nantes), no LINA (Laboratoire d'Informatique de Nantes Atlantique), Nantes, pela acolhida e pelos recursos disponibilizados durante o tempo em que estive realizando parte deste trabalho na França e mesmo após o meu retorno ao Brasil.

Agradeço à Universidade do Grande Rio – Professor José de Souza Herdy, em especial ao seu reitor, professor Arody Cordeiro Herdy, pelo apoio durante o período em que desenvolvia meus estudos na França.

Agradeço ao professor Arnaldo Vieira da Rocha Filho, grande incentivador do meu trabalho, que sempre me estimulou e colaborou para que eu atingisse meus objetivos.

Agradeço aos professores Álvaro Coutinho, Claudio Esperança e Gerson Zaverucha, da COPPE/UFRJ, pelas suas sugestões e pelos incentivos.

Agradeço aos meus amigos, Ricardo Choren Noya, Eduardo Bezerra, Flavio Tavares, Frederico Cabral, Gabriela Ruberg, Nicolaas Ruberg, Vidal Martins e Cédric Coulon. Alguns deles ajudaram com valiosas sugestões para esta tese. Todos eles ajudaram ao permitirem que eu desfrutasse de sua amizade.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Doutor em Ciências (D.Sc.)

PARALELISMO INTRA-CONSULTA EM CLUSTERS DE BANCOS DE DADOS

Alexandre de Assis Bento Lima

Dezembro/2004

Orientadores: Marta Lima de Queirós Mattoso
Patrick Valduriez

Programa: Engenharia de Sistemas e Computação

Consultas de alto custo estão presentes em vários tipos de aplicações como, por exemplo, as aplicações OLAP (*On-Line Analytical Processing*). Elas demandam grande capacidade de armazenamento e poder de processamento dos sistemas de bancos de dados que as suportam. Nesta tese, propomos uma solução para essa demanda implementando paralelismo intra-consulta em um cluster de bancos de dados com SGBDs utilizados como componentes do tipo "caixa-preta". Nossa proposta emprega uma técnica simples e eficiente denominada fragmentação virtual adaptativa, que não requer nenhum conhecimento prévio a respeito do banco de dados ou do SGBD. Ela inclui ainda uma técnica distribuída para balanceamento dinâmico de carga a fim de lidar com problemas causados por distorção na distribuição de dados. Para validar nossa solução, implementamos um protótipo de um cluster de bancos de dados utilizando a linguagem Java e o SGBD PostgreSQL em um cluster com 64 nós e realizamos experimentos com o *benchmark* TPC-H. Os resultados mostram que nossa solução é capaz de obter aceleração linear e, diversas vezes, superlinear durante o processamento de consultas. Utilizando dados com distribuição uniforme, ela apresenta melhor desempenho do que a fragmentação virtual simples na maioria dos casos e é superior em todos os casos com distribuição não uniforme. Finalmente, resultados com múltiplas submissões de consultas também apontam a superioridade de nossa solução intra-consulta sobre a solução que utiliza exclusivamente paralelismo inter-consultas.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Doctor of Science (D.Sc.)

INTRA-QUERY PARALLELISM IN DATABASE CLUSTERS

Alexandre de Assis Bento Lima

December/2004

Advisors: Marta Lima de Queirós Mattoso
Patrick Valduriez

Department: Systems and Computing Engineering

Heavy-weight queries are found in many applications, e.g., OLAP (On-Line Analytical Processing) applications. They typically require high storage capacity and processing power from the underlining database system. In this thesis, we address this problem implementing intra-query parallelism in a database cluster with black-box DBMSs. Our solution to heavy-weight query processing uses a simple, yet efficient, adaptive virtual partitioning technique, without requiring any knowledge about the database and the DBMS. It includes a distributed load balancing technique to deal with attribute data skew. To validate our solution, we implemented a Java database cluster prototype using PostgreSQL DBMS on a 64-node cluster and ran experiments with the TPC-H benchmark. The results show that our solution yields linear, and often super-linear, speedup. It outperforms traditional virtual partitioning in most situations with uniform data distribution and in all cases with data skew. Finally, multi-query experiments show our intra-query solution is superior to those that employ only inter-query parallelism.

Índice

I. Introdução

Aplicações para processamento analítico on-line (OLAP – *On-Line Analytical Processing*) necessitam de suporte de alto desempenho para operações em bancos de dados. Tipicamente, essas aplicações acessam grandes conjuntos de dados através de consultas de alto custo. Operações de atualização são menos freqüentes. O *benchmark* TPC-H (TPC, 2003a), por exemplo, voltado para representar genericamente aplicações OLAP, apresenta vinte e quatro operações de acesso ao banco de dados. Entre elas, vinte e duas são consultas de alto custo enquanto apenas duas são atualizações. O alto custo não é a única característica relevante das consultas de aplicações OLAP. Estudos feitos por GORLA (2003) mostram que, à medida em que os usuários de sistemas desse tipo adquirem mais experiência, maior é a demanda por suporte ao processamento de consultas *ad-hoc*. A otimização de bancos de dados para suporte a consultas *ad-hoc* é um pouco mais complexa, uma vez que elas não são predefinidas. Torna-se então mais difícil a escolha das estruturas de acesso a serem criadas e das formas de organização dos registros em disco entre outras.

Tradicionalmente, alto desempenho em aplicações de banco de dados tem sido obtido através de utilização de processamento paralelo em servidores com hardware especializado, utilizando processadores fortemente acoplados (VALDURIEZ, 1993). Nesses servidores, os dados são replicados ou fragmentados entre os nós com o objetivo de se obter paralelismo durante o processamento de consultas. Apesar de bastante eficiente, essa é uma solução dispendiosa em termos de hardware e software. Além disso, requer que o sistema de banco de dados paralelo tenha total controle sobre os dados, o que torna também dispendiosa a migração de dados a partir de ambientes seqüenciais.

Clusters de bancos de dados são agora uma alternativa de menor custo para sistemas de bancos de dados paralelos. Um "cluster de bancos de dados" consiste em um cluster de PCs, executando, cada um deles, um Sistema Gerenciador de Bases de Dados (SGBD) padrão, ou seja, seqüencial (AKAL *et al.*, 2002). Alguns fabricantes de SGBDs provêem hoje extensões de seus produtos para sua utilização em clusters de PCs. Como exemplos, temos o Oracle Real Application Cluster 10g (ORACLE, 2003), o IBM DB2 ICE (DB2, 2004), o MySQL Cluster (MYSQL, 2004a), o PG-Cluster (PGCLUSTER, 2004) e o Clusgres (CLUSGRES, 2004). A proposta dos clusters de

bancos de dados porém é utilizar SGBDs seqüenciais como componentes do tipo "caixa-preta", ou seja, sem modificar seu código-fonte para incluir funcionalidades que melhorem sua utilização em clusters. Essa abordagem evita as dispendiosas operações de migração de bancos de dados, normalmente necessárias nos sistemas paralelos (GANÇARSKI, 2002a). A utilização de SGBDs como "caixas-pretas" em clusters exige, no entanto, o desenvolvimento de uma camada de software intermediária (*middleware*), entre a aplicação e os SGBDs, capaz de coordenar as operações dos mesmos a fim de obter o paralelismo desejado.

As principais soluções de clusters de bancos de dados com SGBDs caixas-pretas são PowerDB (POWERDB, 2004), Leg@Net (GANÇARSKI *et al.*, 2002a) e C-JDBC (CECCHET *et al.*, 2004). Entretanto, a única que permite o paralelismo intra-consulta é o PowerDB. Os demais se baseiam no paralelismo inter-consultas. Esse tipo de paralelismo possibilita a obtenção de alto desempenho no processamento de números elevados de pequenas transações concorrentes, típicas do ambiente OLTP (*On-Line Transaction Processing*). Um número elevado de consultas concorrentes não costuma ser, no entanto, um problema em ambientes OLAP. Normalmente, poucas consultas são submetidas simultaneamente. Porém, são consultas de alto custo, que costumam apresentar longo tempo de processamento mesmo se executadas isoladamente. Nesse caso, a solução inter-consultas não é atraente, pois não consegue reduzir esse tempo.

Uma solução simples e eficiente para processamento paralelo intra-consulta com consultas OLAP em clusters de bancos de dados é a "fragmentação virtual" (AKAL *et al.*, 2002). O principal requisito para a utilização da fragmentação virtual é a necessidade de que cada nó do cluster seja capaz de acessar todo o banco de dados. Isso pode ser obtido se for utilizada uma arquitetura de disco compartilhado ou uma arquitetura de memória distribuída (incluindo tanto a memória principal quanto a memória em disco) com replicação total da base de dados. Cada consulta submetida ao cluster é reescrita como uma série de sub-consultas, uma para cada nó, através da adição de predicados que definem intervalos correspondentes a diferentes fragmentos virtuais de uma relação. As consultas reescritas podem ser então processadas em paralelo por todos os nós. Em comparação com a fragmentação física (real), a fragmentação virtual apresenta as seguintes vantagens: ela elimina a necessidade de um projeto de banco de dados distribuído e provê grande flexibilidade para a definição dos fragmentos e alocação destes aos nós durante o processamento de consultas.

Fragmentar virtualmente uma relação garante apenas que cada nó processará diferentes sub-conjuntos de tuplas da relação fragmentada. Porém, uma vez que o tamanho de uma tupla é normalmente menor do que o tamanho de um bloco de disco, alguns fragmentos virtuais podem envolver tuplas localizadas em várias páginas de disco diferentes. Eventualmente, as tuplas de um único fragmento podem estar localizadas em todos os blocos alocados para a relação fragmentada, o que implicaria em um alto custo no acesso ao disco. Para resolver esse problema, o atributo escolhido para a determinação dos intervalos correspondentes aos fragmentos, denominado "atributo de fragmentação", deve ter sido previamente utilizado para a ordenação física das tuplas em disco. Com isso, garante-se que as tuplas pertencentes a um mesmo fragmento se encontram agrupadas no disco. Além disso, deve haver um índice ordenado por esse atributo associado à relação fragmentada, a fim de facilitar a localização do fragmento por parte do SGBD (AKAL *et al.*, 2002).

Nesta tese, nos referimos à proposta original da fragmentação virtual feita por AKAL *et al.* (2002) como "fragmentação virtual simples" (SVP – *Simple Virtual Partitioning*). A SVP assume distribuição uniforme dos valores associados ao atributo de fragmentação entre as tuplas da relação. Os tamanhos dos intervalos correspondentes aos fragmentos são obtidos calculando-se a diferença entre o maior e o menor valores do atributo de fragmentação presentes nas tuplas e dividindo-se essa diferença pelo número de nós utilizados para o processamento da consulta. Todos os fragmentos virtuais são então definidos por intervalos do mesmo tamanho. Essa abordagem simples torna o processo de reescrita da consulta leve e ágil. Ele pode porém ser prejudicado por dois sérios problemas que podem diminuir dramaticamente o desempenho da SVP: grandes fragmentos virtuais e distorção de dados.

Grandes fragmentos virtuais. Na SVP, o tamanho do fragmento virtual utilizado durante o processamento das sub-consultas é determinado pela cardinalidade da relação virtualmente fragmentada e pelo número de nós utilizados. Quanto maior a cardinalidade da relação, maiores as dos seus fragmentos, para um número fixo de nós. Para a fragmentação virtual ser bem sucedida, é necessário que o plano de execução gerado pelo SGBD acesse a tabela virtualmente fragmentada através de um índice construído sobre o atributo de fragmentação. Porém, em vários SGBDs, quando, para o processamento de uma consulta, o número estimado de tuplas de uma relação a serem acessadas ultrapassa determinado limite, essa relação é acessada utilizando-se a busca

linear. Se um ou mais nós decidem recuperar toda a relação do disco, ou seja, realizar a busca linear, o ganho de desempenho obtido com o uso de paralelismo será prejudicado. Na SVP, o único modo de se evitar isso é aumentando o número de nós do cluster.

Distorção de Dados. Em muitas aplicações, é comum a existência de distorção de dados (*data skew*), em particular distorção relacionada a valores de atributos, que ocorre quando os valores de alguns atributos não se encontram distribuídos uniformemente entre as tuplas de uma relação (WALTON *et al.*, 1991). A distorção de valores de atributos pode levar a SVP a produzir fragmentos virtuais de tamanhos muito diferentes no que diz respeito ao número de tuplas. Apesar dos tamanhos dos seus intervalos serem iguais, um número diferente de tuplas pode pertencer a cada um deles, gerando desbalanceamento inicial de carga entre os nós do cluster. Quanto maior a distorção, mais severo o desbalanceamento. Há outros tipos de distorção de dados (MÄRTENS, 1999, WALTON *et al.*, 1991), que podem levar a problemas de desbalanceamento de carga mesmo que a carga inicial seja igualmente distribuída.

Considerando as deficiências da única proposta de paralelismo intra-consulta em clusters de bancos de dados que encontramos na literatura (AKAL *et al.*, 2002) e considerando o forte apelo do custo/benefício da solução de alto desempenho para OLAP através de clusters de bancos de dados, surgiu a motivação desta tese. Assim, propomos uma estratégia de paralelismo intra-consulta baseada na fragmentação virtual para processamento de consultas OLAP em clusters de bancos de dados que soluciona os problemas da SVP, mantendo sua simplicidade e flexibilidade. Essa estratégia inova ao propor a Fragmentação Virtual Adaptativa (AVP – *Adaptive Virtual Partitioning*), com resultados de sucesso já reportados em LIMA *et al.* (2004b). A AVP é uma técnica simples e eficiente que ajusta dinamicamente os tamanhos dos fragmentos virtuais, sem utilizar, para isso, nenhuma informação prévia a respeito do banco de dados nem do SGBD. Ela resolve os problemas ligados a grandes fragmentos virtuais. Além da AVP, nossa estratégia também contribui com um algoritmo descentralizado para redistribuição dinâmica de tarefas a fim de lidar com os problemas de desbalanceamento de carga, em especial com a distorção de dados.

Assim como a SVP, nossa solução pode ser utilizada tanto em um cluster com arquitetura de disco compartilhado como em um cluster com arquitetura de memória distribuída utilizando replicação total do banco de dados. A utilização em clusters de memória distribuída com replicação parcial exige algumas adaptações na

implementação. Para validar nossa solução, implementamos um protótipo de cluster de banco de dados na linguagem Java e executamos experimentos utilizando consultas representativas do *benchmark* TPC-H em um cluster de memória distribuída com 64 nós executando o SGBD PostgreSQL (POSTGRESQL, 2004). O banco de dados definido no TPC-H possui distribuição uniforme de valores para os atributos de fragmentação virtual utilizados. Por isso, simulamos distorção de dados em alguns experimentos a fim de avaliar o comportamento da nossa estratégia. Nos experimentos sem distorção, nossa solução apresenta melhoria de desempenho consistente na medida em que quantidades maiores de nós são utilizados. Ela apresenta melhor desempenho do que a SVP na maioria das situações e maior robustez em todas. Nos experimentos com distorção de dados simulada, o desempenho obtido é sempre melhor do que o apresentado pela SVP, exceto em raros casos de configurações com dois nós.

Realizamos ainda experimentos com submissão concorrente de lotes de consultas para simular a utilização do sistema por usuários em processo de análise de dados. Com isso, pudemos comparar a utilização exclusiva de paralelismo inter-consultas com a nossa proposta de combinar paralelismo inter-consultas e intra-consulta. Vários cenários foram testados, desde os típicos para sistemas OLAP sugeridos pelo TPC-H até cenários com grande número de lotes e distorção de dados acentuada. No caso de distribuição de dados uniforme, aumentos super-lineares de vazão foram obtidos em várias situações. Na presença de distorção de dados extrema, o aumento da vazão também foi significativo. Comparando a utilização exclusiva de paralelismo inter-consultas com a utilização da combinação inter-/intra-consulta, concluímos que a estratégia que melhor utiliza os recursos de processamento do cluster é a segunda. Além disso, é a que proporciona maior agilidade nos processos de análise de dados.

Para apresentar nossas propostas, organizamos esta tese da seguinte maneira: no capítulo II, apresentamos os conceitos de clusters de computadores, clusters de bancos de dados, fragmentação real e virtual, e analisamos os principais trabalhos relacionados ao nosso tema. No capítulo III, apresentamos nossa estratégia para processamento de consultas em clusters de bancos de dados descrevendo a fragmentação virtual adaptativa e o algoritmo para redistribuição dinâmica de carga entre nós do cluster. Propomos a arquitetura de um processador de consultas para clusters de bancos de dados no capítulo IV. Os resultados dos experimentos realizados com nosso protótipo são apresentados no capítulo V. Nossas conclusões são apresentadas no capítulo VI.

II. Clusters e Sistemas de Bancos de Dados

As técnicas propostas nesta tese se baseiam nos conceitos de *clusters de bancos de dados* e *fragmentação virtual* de dados. Na seção II.1 descrevemos o conceito de clusters de bancos de dados. Na seção II.2, analisamos as questões de fragmentação e alocação de dados em sistemas paralelos e mostramos o conceito de fragmentação virtual. Em seguida, analisamos trabalhos correlatos na seção II.3. Concluímos o capítulo com a seção II.4.

II.1. Clusters de Bancos de Dados

Clusters de PCs (*Personal Computers* – computadores pessoais) são hoje uma alternativa viável aos computadores com arquiteturas paralelas com processadores fortemente acoplados – como os sistemas com processadores simétricos ou com arquitetura NUMA (*Non Uniform Memory Architecture*) – para várias classes de aplicações que demandem alto desempenho. Um *cluster* pode ser definido como um "sistema de computação local que compreende um conjunto de computadores independentes e uma rede para interconectá-los." (STERLING, 2001). Por *local* entenda-se que os componentes normalmente residem em uma mesma sala e são gerenciados como um sistema unificado. A rede de interconexão deve ser dedicada à ligação entre os computadores (nós) que constituem o cluster, sendo separada da rede responsável por conectar o cluster a outros ambientes.

Quando comparados com sistemas com arquiteturas paralelas, os clusters se mostram soluções menos dispendiosas e mais flexíveis (STERLING, 2001). Há, porém, limitações nessa arquitetura. Largura de banda e latência da rede de interconexão; dificuldade de se implementar um modelo de computação com memória compartilhada; e necessidade de se ter um sistema operacional sendo executado em cada nó são alguns exemplos que tornam a utilização de clusters difícil para certos tipos de aplicações. Porém, para várias outras classes, o desempenho obtido é satisfatório. O sítio "TOP500 Supercomputer Sites" (TOP500, 2004), que tem o objetivo de manter uma lista com os quinhentos sistemas computacionais de maior desempenho, possui, segundo consulta realizada em novembro de 2004, duzentos e noventa e quatro entradas correspondentes a clusters de computadores.

Especificamente na área de bancos de dados, podemos notar que vários fabricantes têm investido nessa direção. O uso de processamento paralelo em arquiteturas com processadores fortemente acoplados é um recurso já bastante utilizado para a obtenção de alto desempenho em aplicações de bancos de dados (VALDURIEZ, 1993). Nota-se agora um grande interesse na utilização de clusters de computadores com esse objetivo. O "Transaction Processing Performance Council" (TPC) é uma organização com fins não-lucrativos cuja missão, encontrada em seu sítio na Internet, é "definir *benchmarks* para processamento de transações e bancos de dados, e disseminar para a indústria dados objetivos sobre desempenho" (TPC, 2004). São definidos *benchmarks* para aplicações OLTP (*On-Line Transaction Processing*), aplicações de apoio à decisão com consultas *ad-hoc* e relatórios pré-definidos, e aplicações de comércio eletrônico na *World Wide Web*. Uma consulta realizada em novembro de 2004 às suas listas de sistemas de melhores desempenhos mostra, para o *benchmark* TPC-C (aplicações OLTP), um sistema baseado em cluster em primeiro lugar. O mesmo acontece para o benchmark TPC-H (específico para consultas *ad-hoc* em sistemas de apoio à decisão) com conjuntos de dados de tamanhos iguais a 100 Gb, 300 Gb, 1.000 Gb e 10.000 Gb. Isso é um forte indicador de que a utilização dessa arquitetura para sistemas de bancos de dados com necessidade de alto desempenho pode trazer excelentes resultados.

Também na área acadêmica o interesse pela utilização de clusters por parte da comunidade de pesquisa em bancos de dados é grande. Tanto que levou à definição do termo *cluster de bancos de dados*, encontrada no trabalho de AKAL *et al.* (2002) e que mostramos a seguir.

Definição 1 (Cluster de Bancos de Dados) *Um cluster de bancos de dados consiste em um cluster de PCs, executando, em cada nó, um Sistema Gerenciador de Bancos de Dados padrão.*

Por Sistema Gerenciador de Bancos de Dados (SGBD) *padrão* entenda-se um SGBD seqüencial, sem características especiais que o permitam tirar vantagens do fato de estar sendo utilizado em um cluster e sem algoritmos paralelos para a execução de operações. AKAL *et al.* (2002) especificam que os SGBDs são relacionais. Modificamos um pouco a definição original para ganhar em generalidade. Acreditamos

que o modelo lógico implementado pelo SGBD não afeta sua utilização em clusters de bancos de dados.

Definido dessa forma, o cluster de bancos de dados demanda a criação de uma camada de software (*middleware*) com o objetivo de orquestrar os SGBDs para a implementação de técnicas de paralelismo. Os SGBDs são tratados como componentes do tipo "caixas-pretas", ou seja, seus códigos-fonte não são modificados para a implementação dessas técnicas. A Figura 1 ilustra os principais componentes de um cluster de bancos de dados.

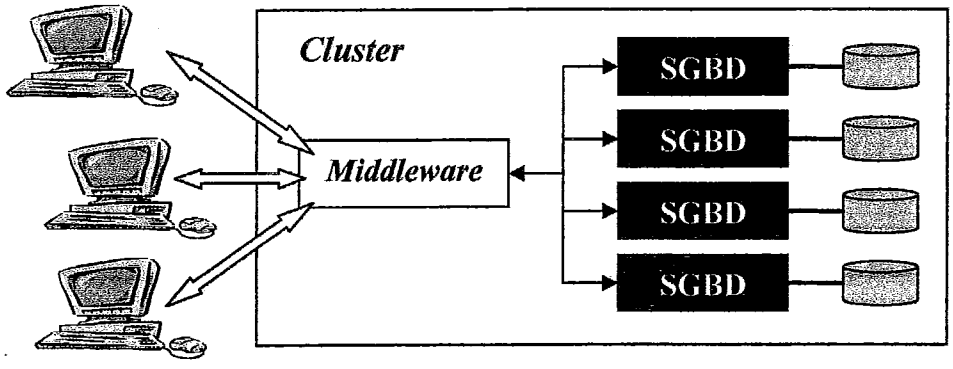

Figura 1 - Cluster de bancos de dados

Como podemos notar a partir da definição de clusters, os clusters de bancos de dados são sistemas com arquitetura do tipo memória distribuída. Tal arquitetura possui como característica principal sua expansibilidade, sendo a que proporciona maior facilidade para expansão do sistema através da adição de unidades de processamento, quando comparada às arquiteturas de memória compartilhada e disco compartilhado.

II.2. Projeto de Distribuição em Clusters de Bancos de Dados

Outro fator importante em sistemas de bancos de dados, além da arquitetura, é o projeto de distribuição dos dados entre as unidades de armazenamento. Seu objetivo é definir a função de fragmentação que determina a composição de cada fragmento, sua alocação e sua replicação nos nós do cluster. Em arquiteturas de memória distribuída, a distribuição de dados influencia o balanceamento de carga entre os nós mais do que em outros tipos de arquitetura. O acesso a dados armazenados em discos de outros nós envolve troca de mensagens pela rede, o que pode torná-la um ponto de contenção na arquitetura. O projeto de distribuição é elaborado então com os objetivos de reduzir a

comunicação e, ao mesmo tempo, tentar manter a carga de trabalho balanceada entre os nós.

A distribuição de dados também influencia o tipo de paralelismo utilizado durante o processamento de consultas. Há, basicamente, dois tipos de paralelismo possíveis em sistemas de bancos de dados para o processamento de consultas: "paralelismo inter-consultas" e "paralelismo intra-consulta" (ÖZSU e VALDURIEZ, 1999). O primeiro consiste em se executar simultaneamente consultas diferentes utilizando-se nós (ou conjuntos de nós) distintos. O segundo consiste em se utilizar vários nós para a execução de uma única consulta. Vale ressaltar que essas estratégias não são mutuamente exclusivas.

O projeto de distribuição de dados se dá em duas fases: fragmentação e alocação. Na próxima seção, descrevemos de forma sucinta o objetivo de cada uma delas. Na seção seguinte, descrevemos mais detalhadamente a fragmentação virtual, técnica que alia o paralelismo intra-consulta à flexibilidade da replicação total.

II.2.1. Fragmentação e Alocação de Dados

A fragmentação de dados consiste na divisão de relações em sub-relações disjuntas ou "fragmentos" (ÖZSU e VALDURIEZ, 1999). Seu objetivo é tornar mais eficiente a execução paralela de transações que acessem subconjuntos diferentes de tuplas de uma mesma relação. Há basicamente três opções de fragmentação: "horizontal", "vertical" e "híbrida".

A fragmentação horizontal é aquela em que os fragmentos são obtidos através de operações de seleção feitas sobre a relação original. Os fragmentos resultantes possuem esquemas idênticos. Na fragmentação vertical, os fragmentos são obtidos através de projeções feitas sobre a relação a ser fragmentada. É importante que os atributos que formam a chave primária estejam presentes em todos os fragmentos verticais, a fim de que o conjunto de dados original possa ser reconstruído, caso seja necessário. Por fim, a fragmentação híbrida representa uma combinação das anteriores: os fragmentos são obtidos através de seleções seguidas de projeções.

Após a fragmentação, deve-se proceder à alocação dos fragmentos nos nós. Pode-se armazenar uma única cópia de cada fragmento ou pode-se optar pela replicação de alguns ou de todos eles (ÖZSU e VALDURIEZ, 1999). Caso haja apenas uma cópia de cada fragmento em toda a rede, temos um banco de dados denominado não-replicado.

Para aumentar a disponibilidade e a confiabilidade do sistema, além de melhorar o desempenho de operações de consulta, pode-se optar pela replicação.

A replicação pode ser total ou parcial. Denominamos a replicação "total" quando cada nó armazenar uma cópia de toda a base de dados. A replicação total representa o máximo em termos de disponibilidade e flexibilidade para a distribuição de tarefas entre nós no que diz respeito ao processamento de consultas. Normalmente, encontramos essa alternativa de replicação em trabalhos sobre paralelismo inter-consultas devido a essa flexibilidade. AKAL *et al.* (2002), porém, apresentam uma proposta para a obtenção de paralelismo intra-consulta com bancos de dados totalmente replicados. Descrevemos mais detalhadamente essa proposta na seção II.2.2.

A utilização da replicação total pode não ser possível para algumas classes de problemas. Se o tamanho do banco de dados exceder a capacidade de armazenamento de cada nó, a replicação total não pode ser adotada. Nesse caso, deve-se optar pela não-replicação ou, se possível, pela replicação parcial. Esta última consiste em se replicar alguns fragmentos (ou relações inteiras) do banco de dados em alguns nós enquanto outros fragmentos não são replicados.

Uma comparação entre a utilização da replicação total e da replicação parcial em um cluster de bancos de dados foi feita no trabalho de RÖHM *et al.* (2000). O banco de dados utilizado foi o definido no benchmark TPC-R (TPC, 2003b), específico para relatórios OLAP, ou seja, consultas sem natureza *ad-hoc*. Nos experimentos com replicação total, como esperado, todas as relações (tabelas de fatos e dimensões) foram replicadas em todos os nós do cluster. Cada consulta foi executada inteiramente em apenas um nó, ou seja, foi empregado apenas paralelismo inter-consultas. Nos experimentos com replicação parcial, denominada "projeto híbrido", apenas a maior relação, tabela de fatos Lineitem, foi fragmentada. Cada fragmento recebeu $1/n$ tuplas da relação original, onde n corresponde ao número de nós do cluster, e foi armazenado em um nó diferente do cluster, sem réplicas. Todas as demais relações foram replicadas em todos os nós. Nessa série de experimentos, cada consulta utilizando a relação Lineitem foi executada por todos os nós simultaneamente, ou seja, foi empregado o paralelismo intra-consulta.

Nos experimentos com consultas utilizando replicação total, foi alcançada aceleração linear ao se aumentar o número de nós presentes no cluster, mostrando que bons resultados podem ser obtidos com essa estratégia. Porém, os experimentos

utilizando replicação parcial alcançaram aceleração superlinear, o que revelou ser essa estratégia mais eficiente. Na replicação parcial, a relação Lineitem possuía menor cardinalidade em cada nó do que no caso da replicação total. Devido a isso, os SGBDs geraram planos de execução mais eficientes para as sub-consultas. Isso leva a crer que o paralelismo intra-consulta tem desempenho superior ao inter-consultas no caso de consultas de alto custo como as das aplicações OLAP.

Outro trabalho que também explora fragmentação em bancos de dados para aplicações OLAP é o de STÖHR *et al.* (2000). Porém, nesse caso, os experimentos simulam a utilização de uma arquitetura de disco compartilhado, onde cada nó da máquina paralela possui processador e memória próprios e tem acesso direto a uma unidade de armazenamento secundário compartilhada. O trabalho propõe uma estratégia de fragmentação da tabela de fatos denominada "Fragmentação Herárquica Multi-Dimensional (MDHF – *Multi-Dimensional Hierarchical Fragmentation*). A MDHF é baseada na observação de que a maioria das tabelas de dimensão de um esquema estrela possui atributos que formam hierarquias. A fragmentação da tabela de fatos é então feita a partir da escolha de atributos pertencentes a hierarquias de diferentes dimensões. Atributos localizados em posições superiores das hierarquias geram fragmentos maiores. Atributos localizados em posições inferiores geram fragmentos menores. Todos os fragmentos são armazenados na unidade de armazenamento compartilhada, composta, na simulação, por vários discos. Cada fragmento possui seu próprio índice do tipo *bitmap*.

Nos experimentos, os autores analisam diversas opções de fragmentação para a tabela de fatos "Sales" do benchmark APB-1, juntamente com diferentes consultas do *benchmark*. As diferentes fragmentações propostas são baseadas nos perfis das consultas do *benchmark*. Os resultados mostram ganho de desempenho quase linear na maioria dos casos.

Um ponto que gostaríamos de ressaltar no trabalho de STÖHR *et al.* (2000) é o bom desempenho obtido com a arquitetura de disco compartilhado. Ela se assemelha à combinação da arquitetura de memória distribuída com a replicação total na medida em que permite flexibilidade na escolha dos nós para o processamento de consultas, pois todos os nós podem acessar diretamente quaisquer conjuntos de dados. Outro ponto que gostaríamos de destacar é o fato de que a fragmentação considerada ideal foi obtida a partir da análise do perfil de consultas pré-estabelecidas. Em ambientes com consultas

ad-hoc, dificuldades poderiam ser encontradas para a realização de tal análise, o que pode limitar o uso da MDHF.

Concluímos que a replicação parcial pode ser uma boa alternativa para alocação de dados. Ela reduz porém a flexibilidade na escolha dos nós utilizados no processamento de uma consulta, ao mesmo tempo em que aumenta a complexidade do algoritmo de escolha. Caso as consultas de uma aplicação acessem alguns fragmentos mais freqüentemente do que outros, o problema do desbalanceamento de carga pode surgir. A alternativa mais simples para a arquitetura de memória distribuída é a replicação total. Ela também é a que proporciona mais facilidades para a expansão do sistema, uma vez que não exige uma nova fragmentação de dados caso novos nós sejam adicionados ao sistema. Na próxima seção, descrevemos uma técnica que combina as facilidades dessa arquitetura com a eficiência do paralelismo intra-consulta.

II.2.2. Fragmentação Virtual

A estratégia de replicação total aumenta a flexibilidade no processo de alocação de tarefas durante o processamento de uma consulta. Como todos os nós possuem acesso local a todos os dados, qualquer tarefa pode ser atribuída a qualquer um deles. No caso de clusters de bancos de dados com replicação total, uma estratégia para a obtenção de paralelismo intra-consulta interessante por sua simplicidade foi proposta por AKAL *et al.* (2002). Tal estratégia é baseada no conceito de fragmento virtual definido a seguir.

Definição 2 (Fragmento Virtual) *Dada uma relação R e um predicado P da forma* $((A \geq a_1) \wedge (A < a_2))$, $a_2 > a_1$, *onde A é escolhido como o atributo de fragmentação de R e* a_1 *e* a_2 *são valores do domínio de A, um fragmento virtual de R, denominado* R_{VP}, *é a relação resultante da seleção em R com o predicado P, onde* $R_{VP} = \sigma_P(R)$.

A denominação "fragmento virtual" é dada em contraposição aos fragmentos reais produzidos por técnicas que realizam fragmentação física da base de dados.

Descrevemos agora a Fragmentação Virtual (VP – *Virtual Partitioning*). Suponha que uma consulta C, que acessa dados da relação R, é recebida pelo cluster de bancos de dados. O processador de consultas escolhe um conjunto de n nós (n_0, n_1, ..., n_{n-1}) para processar C. A idéia básica da VP é fazer com que cada nó processe a mesma consulta C em um ou mais fragmentos virtuais diferentes de R. Para isso, por exemplo, n sub-

consultas (C_0, C_1, ..., C_{n-1}) poderiam ser produzidas. Cada sub-consulta C_i é obtida substituindo-se R em C por um fragmento virtual R_{VPi} diferente. Em seguida, cada C_i é enviada para um nó n_i diferente. O processo é ilustrado na Figura 2 para um cluster com quatro nós.

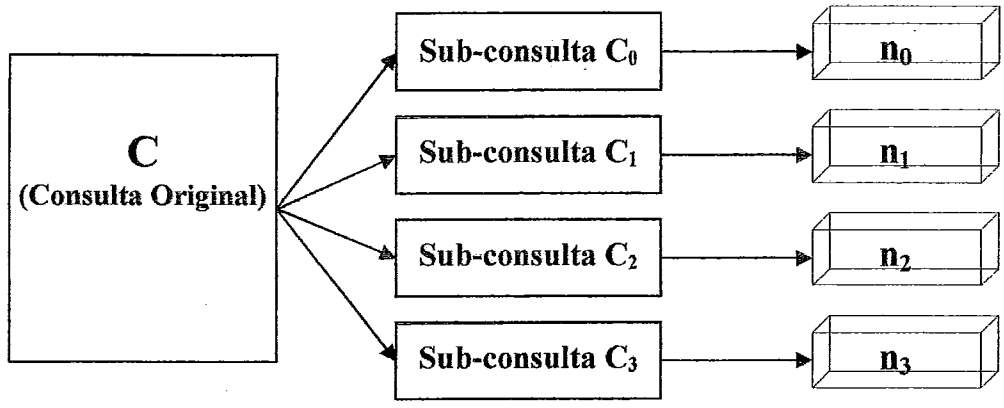

Figura 2 - Esquema da VP em um cluster com quatro nós

Após o processamento de todas as sub-consultas, é necessário que se faça a coleta dos seus resultados para envio à aplicação cliente. Caso C envolva agregações, pode ser necessária uma fase de pós-processamento para a produção do resultado final. Vale ressaltar ainda que, nesse caso, a elaboração das sub-consultas pode envolver a substituição de certas funções de agregação da consulta original e a inclusão de algumas outras a fim de que o resultado final possa ser corretamente obtido.

Na proposta de AKAL *et al.* (2002), a determinação dos tamanhos dos fragmentos virtuais, ou seja, dos intervalos de valores do atributo de fragmentação em cada R_{VPi}, é feita de maneira bem simples. Seja $[a_j, a_k]$, $a_k > a_j$, o intervalo que define o domínio do atributo de fragmentação A, o intervalo de cada fragmento virtual terá o tamanho $t_f = \lceil |(a_k - a_j)|/n \rceil$, considerando valores numéricos contínuos. Em geral, num ambiente de paralelismo em bancos de dados, atributos chave são os escolhidos como atributo de fragmentação, sendo numéricos na maioria das vezes. Denominamos essa abordagem "fragmentação virtual simples" (SVP – *Simple Virtual Partitioning*).

Vamos ilustrar a estratégia de paralelismo SVP com uma consulta baseada em uma relação do *benchmark* TPC-H (TPC, 2003a) denominada Lineitem. Trata-se de uma das tabelas de fatos definidas no *benchmark*, que é voltado para consultas *ad-hoc* em bancos de dados para aplicações OLAP. Suponha um cluster de bancos de dados

com quatro nós: n_0, n_1, n_2 e n_3. A seguinte consulta, que corresponde à consulta Q6 do *benchmark*, é recebida por esse cluster:

```
Q6: select  sum( l_extendendprice * l_discount ) as revenue
    from    lineitem
    where   l_shipdate >= date '1994-01-01'
      and   l_shipdate < date '1994-01-01' + interval '1 year'
      and   l_discount between .06 - 0.01 and .06 + 0.01
      and   l_quantity < 24;
```

Como o cluster possui quatro nós, quatro sub-consultas $Q6_i$, $0 \leq i \leq 3$, devem ser geradas, uma para cada nó. Suponha que o atributo l_orderkey é escolhido como atributo de fragmentação e que os valores para esse atributo nas tuplas de Lineitem pertençam ao intervalo [1, 6.000.000]. De acordo com a SVP, em cada n_i, $Q6_i$ processará o fragmento virtual $Lineitem_i$ contendo 1.500.000 tuplas. Assim, a substituição de Lineitem por $Lineitem_i$ se dá, conforme a definição 2, a partir da inclusão do predicado de seleção que restringe as tuplas de Lineitem a serem percorridas em n_i. As sub-consultas de Q6 seriam então as seguintes:

```
Q60: select  sum( l_extendendprice * l_discount ) as revenue
     from    lineitem
     where   l_shipdate >= date '1994-01-01'
       and   l_shipdate < date '1994-01-01' + interval '1 year'
       and   l_discount between .06 - 0.01 and .06 + 0.01
       and   l_quantity < 24
       and   l_orderkey >= 1 and l_orderkey < 1500001;

Q61: select  sum( l_extendendprice * l_discount ) as revenue
     from    lineitem
     where   l_shipdate >= date '1994-01-01'
       and   l_shipdate < date '1994-01-01' + interval '1 year'
       and   l_discount between .06 - 0.01 and .06 + 0.01
       and   l_quantity < 24
       and   l_orderkey >= 1500001 and l_orderkey < 3000001;

Q62: select  sum( l_extendendprice * l_discount ) as revenue
     from    lineitem
     where   l_shipdate >= date '1994-01-01'
       and   l_shipdate < date '1994-01-01' + interval '1 year'
       and   l_discount between .06 - 0.01 and .06 + 0.01
       and   l_quantity < 24
       and   l_orderkey >= 3000001 and l_orderkey < 4500001;

Q63: select  sum( l_extendendprice * l_discount ) as revenue
     from    lineitem
     where   l_shipdate >= date '1994-01-01'
       and   l_shipdate < date '1994-01-01' + interval '1 year'
       and   l_discount between .06 - 0.01 and .06 + 0.01
       and   l_quantity < 24
       and   l_orderkey >= 4500001 and l_orderkey < 6000001;
```

A consulta Q6 executa uma função de agregação (um somatório, nesse caso). Para que o resultado correto de Q6 seja produzido, é necessário que os resultados produzidos pela sub-consultas sejam somados em uma etapa posterior ao seu processamento.

Alguns requisitos são necessários para que a SVP atinja seu objetivos. O primeiro diz respeito ao atributo de fragmentação. O predicado adicionado às sub-consultas tem como objetivo fazer com que cada nó acesse apenas um subconjunto das tuplas da relação virtualmente fragmentada. Para isso, é necessário que as tuplas que compõem um fragmento virtual (VP) estejam agrupadas no disco rígido, ou seja, as tuplas da relação devem se encontrar ordenadas fisicamente de acordo com os valores do atributo de fragmentação. Caso isso não ocorra, podemos ter casos em que tuplas de um mesmo VP se encontrem espalhadas por todos os blocos de disco ocupados pela relação. Com isso, não evitaríamos que todos os blocos de disco associados à relação fossem carregados para a memória principal.

O segundo requisito é a existência de um índice baseado no atributo de fragmentação. Sem esse índice, a busca pelas tuplas que compõem um VP pode exigir uma busca linear ou binária sobre a tabela. Na grande maioria dos SGBDs, consultas baseadas em atributos para os quais não existem índices definidos são realizadas através da busca linear. Uma busca desse tipo realizada por qualquer um dos nós sobre a tabela virtualmente fragmenta também frustraria o objetivo básico de fazer com que cada nó acesse apenas um subconjunto dos dados.

O fato de a relação estar agrupada de acordo com o atributo de fragmentação e de haver um índice baseado nele não garantem a utilização do índice durante o processamento da consulta. Em vários SGBDs, um índice é escolhido pelo otimizador de consultas como meio de acesso a uma relação apenas se o número estimado de tuplas a serem recuperadas for menor do que um certo limite, que é estabelecido pelo fabricante ou ajustado pelo administrador do sistema. Se o número estimado de tuplas for maior do que esse limite, os índices são desconsiderados e a busca linear é executada. Como estamos supondo um cluster com SGBDs considerados "caixas-pretas", não há como modificar esse comportamento, a menos que o administrador do sistema ajuste o limite para a totalidade das tuplas da relação. Isso pode porém ter um resultado negativo no desempenho global do sistema, pois afetaria todas as consultas, realizadas sobre todas as relações. Isso se torna um problema para a SVP se os

tamanhos dos fragmentos virtuais forem muito grandes ou se, para pelo menos um deles, o SGBD estimar que um número grande de tuplas será recuperado.

Outro fator importante para o sucesso da SVP é a distribuição uniforme de valores do atributo de fragmentação entre as tuplas da relação virtualmente fragmentada. Para reduzir as chances de desbalanceamento de carga entre os nós do cluster, é importante que os VPs tenham tamanhos não muito diferentes, ou seja, que o número de tuplas correspondente a cada intervalo seja aproximadamente o mesmo. Uma distribuição não uniforme dos valores do atributo de fragmentação poderá resultar em severo desbalanceamento de carga entre os nós. Infelizmente, em várias aplicações reais, a hipótese de uniformidade nessa distribuição é irrealista. Como a SVP envia apenas uma consulta a cada nó, e cada SGBD é um componente do tipo "caixa-preta", não é possível a adoção de uma técnica de balanceamento dinâmico entre os nós através da redistribuição da carga de trabalho após o início do processamento das sub-consultas.

Por fim, para que o resultado da consulta executada através da SVP seja correto, é necessário que a definição da fragmentação virtual executada sobre a relação escolhida atenda a critérios de correção conforme definido em SGBDs distribuídos (ÖZSU e VALDURIEZ, 1999). Para fragmentação virtual a correção implica na definição de uma fragmentação completa e disjunta, conceitos que definimos a seguir.

Definição 3 (Fragmentação Virtual Completa de uma Relação) *A fragmentação virtual de uma relação R em n fragmentos é completa se* $\forall r \in R, \exists R_{VPi}, 0 \leq i \leq n-1, r \in R_{VPi}$, *onde r representa as tuplas de R.*

Definição 4 (Fragmentação Virtual Disjunta de uma Relação) *A fragmentação virtual de uma relação R em n fragmentos é disjunta se* $\forall i,j\ (i \neq j, 0 \leq i,j \leq n-1), R_{VPi} \cap R_{VPj} = \varnothing$.

Uma fragmentação virtual completa garante que todas as tuplas da relação fragmentada serão processadas. Uma fragmentação virtual disjunta garante que cada tupla pertence a apenas um fragmento virtual e assim não será processada mais do que uma vez.

II.3. Trabalhos Correlatos

Nessa seção, descrevemos alguns trabalhos relacionados à nossa pesquisa. Na seção II.3.1 são descritos trabalhos que utilizam a arquitetura de clusters de bancos de dados, ou seja, clusters que utilizam SGBDs como componentes "caixas-pretas". A seção II.3.2 descreve algumas extensões proprietárias de SGBDs para clusters.

II.3.1. Sistemas de Clusters de Bancos de Dados

Há algumas iniciativas no sentido de se utilizar clusters de bancos de dados como definidos nesta tese, ou seja, como um conjunto de SGBDs considerados componentes "caixas-pretas", sem capacidade para processamento paralelo, sendo executados em nós de um cluster de PCs e orquestrados por uma camada de software responsável por implementar algoritmos para prover paralelismo nas suas variadas formas. Descrevemos brevemente três dessas iniciativas nessa seção: PowerDB, Leg@Net e C-JDBC.

II.3.1.1 PowerDB

O objetivo do projeto PowerDB é "construir um cluster de SGBDs de alto desempenho utilizando, tanto quanto possível, hardware convencional e componentes de software" (POWERDB, 2004). A arquitetura proposta no projeto está ilustrada na Figura 3, adaptada do trabalho de RÖHM *et al.* (2001) e do sítio do projeto (POWERDB, 2004).

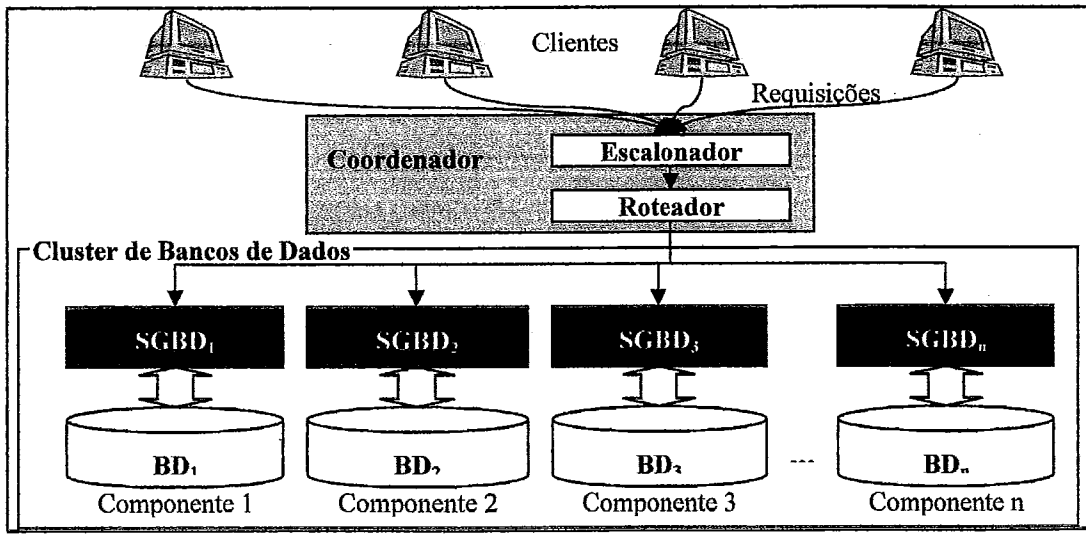

Figura 3 - Arquitetura PowerDB, adaptada de (POWERDB, 2004)

A arquitetura PowerDB propõe o uso de SGBDs Relacionais como componentes "caixa-pretas". Cada SGBD gerencia sua própria instância do banco de dados. O paralelismo é implementado e conduzido por um módulo denominado "coordenador", que serve como ponto de entrada do sistema, ou seja, é a ele que os clientes se conectam para a submissão de requisições. Entre os componentes principais do coordenador se encontram o "escalonador", responsável por gerenciar a fila de requisições recebidas, e o "roteador", responsável pela escolha dos nós que irão executar cada requisição e pelo envio das mesmas a eles.

Vários são os trabalhos no contexto desse projeto. Destacaremos os mais relevantes para o nosso problema. Primeiramente, o trabalho de RÖHM *et al.* (2000) investiga a utilização de esquemas simples de organização de dados e técnicas elementares para roteamento de consultas. O cenário utilizado para testes inclui consultas OLAP, OLTP e operações de atualização de dados obtidas do benchmark TPC-R (TPC, 2003b), específico para relatórios OLAP. Os esquemas de organização de dados analisados são a replicação total e o denominado "projeto híbrido", no qual a maior relação do banco de dados é fragmentada e distribuída entre os nós enquanto as demais são replicadas. Uma discussão mais detalhada sobre esse esquema é apresentada na seção II.2.1. Como conclusão geral, os autores destacam que o esquema de roteamento baseado em um coordenador possui boa *escalabilidade* para cenários com consultas analíticas complexas. Vale destacar que esse trabalho explora paralelismo intra-consulta nos testes realizados com a abordagem híbrida para o projeto de distribuição da base de dados.

O segundo trabalho (RÖHM *et al.*, 2001) apresenta uma estratégia para roteamento de consultas baseada em estimativas a respeito do conteúdo do *cache* de cada SGBD componente. Cada consulta é associada a uma "assinatura", que tenta representar os dados acessados por ela. Essa assinatura é utilizada para a realização de estimativas a respeito do conteúdo do *cache* de um SGBD após a execução da consulta. Quando uma nova consulta é recebida pelo roteador, sua assinatura é produzida. O roteador a utiliza então para avaliar qual seria o melhor nó para processá-la, baseado no conteúdo estimado dos *caches* dos nós. A replicação total é adotada como estratégia de projeto de distribuição para a base de dados e operações de atualização não são consideradas. Os resultados obtidos mostram que a técnica proposta supera as até então

consideradas como sendo o estado da arte para roteamento de consultas. Apenas o paralelismo inter-consultas é explorado nesse trabalho.

Atualizações de dados e consultas analíticas sendo executadas simultaneamente em um ambiente OLAP com replicação total são consideradas no trabalho de RÖHM *et al.* (2002). A idéia básica é a de que consultas OLAP nem sempre precisam ser realizadas sobre conjuntos de dados os mais atualizados possíveis. Ao submeter uma consulta, o usuário pode especificar um "limite de frescor" (*freshness limit*), que estabelece o grau de tolerância em relação à diferença entre os dados mais atualizados e os utilizados para a resolução da consulta. Um novo protocolo – denominado FAS (*Freshness-Aware Scheduling*) – é proposto e tem o objetivo de garantir que o limite de frescor estabelecido será sempre obtido pelas consultas. Ele direciona as consultas para o nó que atenda ao limite solicitado. Isso possibilita um relaxamento na sincronização das réplicas, aumentando o desempenho do sistema em operações de atualização e consultas complexas. O paralelismo inter-consultas é o único explorado nesse trabalho.

Por fim, destacamos o trabalho de AKAL *et al.* (2002), que serve como base para esta tese. Nele, encontramos a definição de cluster de bancos de dados (*database cluster*) que utilizamos aqui, bem como a proposta da fragmentação virtual. Seu objetivo é obter paralelismo intra-consulta para aplicações OLAP utilizando a fragmentação virtual em um cluster de bancos de dados com replicação total. Os testes realizados são feitos com a base de dados e consultas do benchmark TPC-R (TPC, 2003b). O SGDB utilizado é o MS-SQL Server 2000. Os resultados obtidos são excelentes, sendo a aceleração linear alcançada em vários casos, mostrando a fragmentação virtual como uma técnica promissora. Um problema apontado pelos autores é a utilização da técnica com relações que apresentam distribuição não-uniforme para os valores do atributo de fragmentação. Uma solução para esse problema é proposta no nosso trabalho.

II.3.1.2 Leg@Net

O objetivo do projeto Leg@Net (LEGNET, 2001) é "demonstrar a viabilidade do modelo ASP (*Application Service Provider* – provedor de serviços para aplicações) para aplicações farmacêuticas na França" (GANÇARSKI *et al.*, 2002a). De acordo com GANÇARSKI *et al.* (2002b), um provedor de serviços para aplicações hospeda, em seu sítio, aplicações e bancos de dados (dados e SGBDs) dos seus clientes, que precisam

estar disponíveis normalmente através da Internet, de maneira tão eficiente como se estivessem hospedados localmente nos seus próprios sítios. As aplicações devem ser migradas do ambiente do cliente para o ASP de maneira praticamente transparente, sem necessidade de alterações em seus códigos-fonte. Esse é o denominado modelo ASP.

O desafio do projeto Leg@Net é propor técnicas que permitam a obtenção de bom desempenho para as aplicações em um cluster de computadores com bancos de dados autônomos, sem requerer mudanças nas mesmas ou nos esquemas dos seus bancos de dados. Isso é relativamente fácil com aplicações que realizam apenas operações de consultas, caso em que, normalmente, a replicação total pode ser adotada. Porém, as aplicações estudadas se caracterizam por realizarem muitas operações de atualização no banco de dados. Isso requer o desenvolvimento de técnicas para evitar inconsistências que possam ser originadas por operações de atualização simultâneas em diferentes réplicas.

GANÇARSKI *et al.* (2002a) e GANÇARSKI *et al.* (2002b) descrevem algumas dessas técnicas. Ambos os trabalhos consideram um cluster de bancos de dados com nós similares, cada um com um ou mais processadores, memória principal e disco. Os elementos da sua arquitetura conceitual estão descritos na Figura 4. Os processos de autenticação e autorização de usuários são realizados utilizando-se um diretório onde são armazenadas informações a respeito dos usuários e aplicações. Se o processo é bem sucedido, o usuário é conectado à aplicação. Então, a aplicação pode se conectar ao SGBD.

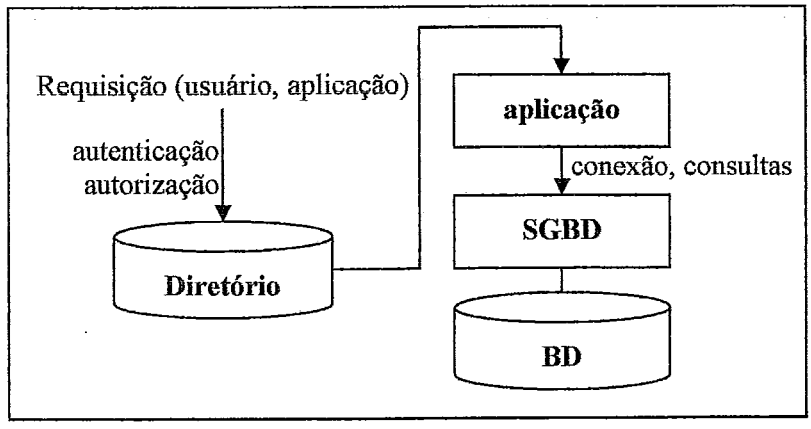

Figura 4 - Leg@Net - Arquitetura Conceitual

GANÇARSKI *et al.* (2002b) propõem três organizações principais para os elementos da arquitetura. Na organização "conexão cliente-servidor ao SGBD" (Figura 5a), uma aplicação cliente em um nó se conecta ao SGBD em outro nó. Na organização

"conexão ponto-a-ponto ao SGBD" (Figura 5b), cada aplicação se conecta a um SGBD local. Esse SGBD possui funcionalidades de distribuição e se conecta, de maneira transparente para o usuário, ao mesmo SGBD em outro nó. Na última organização, denominada "bancos de dados replicados" (Figura 5c), o SGBD e o banco de dados são replicados em todos os nós do cluster.

GANÇARSKI *et al.* (2002a) propõem uma solução para processamento paralelo com bancos de dados autônomos utilizando a organização "bancos de dados replicados". Essa organização é escolhida porque é a "mais genérica e a mais eficiente, uma vez que possibilita tanto a execução paralela de aplicações como de operações de acesso ao banco de dados". O foco do trabalho são aplicações que realizam atualizações intensivas de dados, típicas do contexto ASP. Eles propõem também uma arquitetura para balanceamento de carga durante a execução de transações e discutem sua implementação utilizando um cluster Linux com SGBD Oracle 8i. Uma proposta para balanceamento carga entre bancos de dados e aplicações também é proposta em GANÇARSKI et al. (2002a).

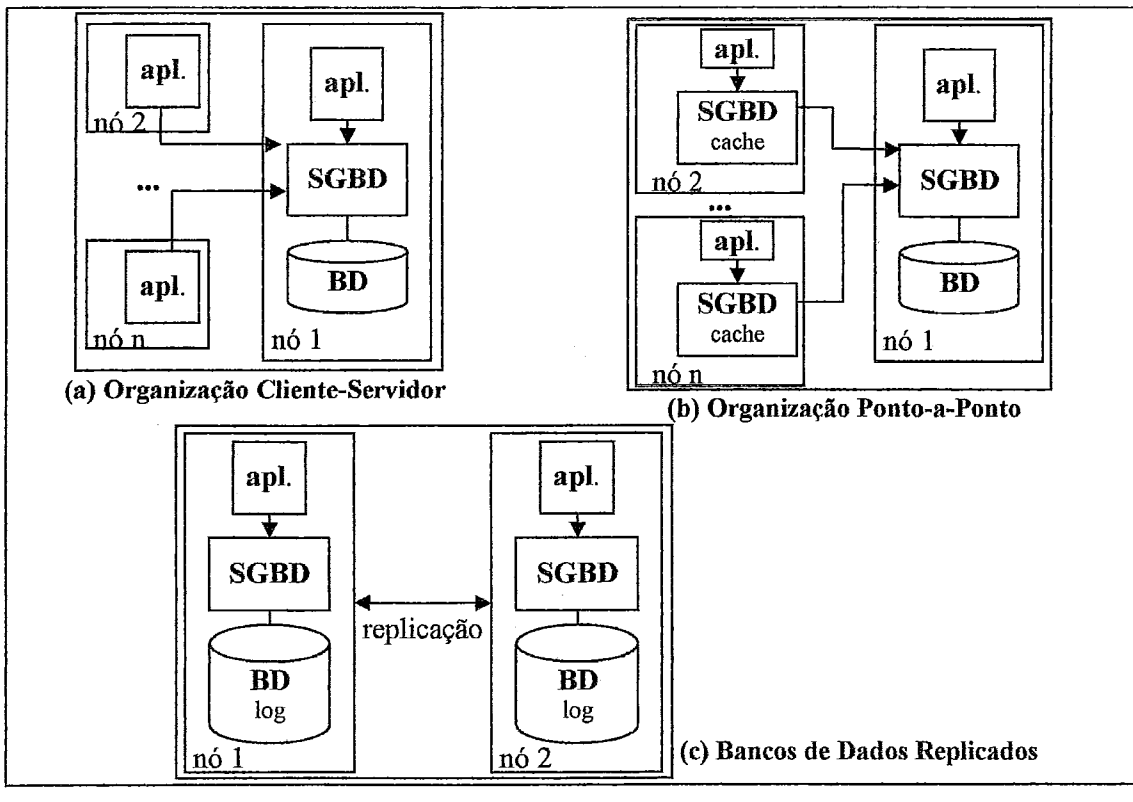

Figura 5 - Organizações alternativas para os elementos da arquitetura Leg@Net

II.3.1.3 C-JDBC

Outro projeto que utiliza clusters de SGBDs "caixas-pretas" é o C-JDBC – *Clustered* JDBC (CECCHET *et al.*, 2004). Trata-se de um *middleware* de código aberto cujo objetivo é possibilitar a construção de clusters de bancos de dados utilizando arquitetura de memória distribuída construída com hardware comum, ou seja, sem processadores paralelos. Os SGBDs utilizados também não possuem características de paralelismo. Trata-se portanto de uma solução que se enquadra no conceito de clusters de computadores que aqui exploramos.

JDBC™ (WHITE *et al.*, 2001) é uma API Java utilizada para acesso a dados tabulares. A grande maioria dos fabricantes de SGBDs (comerciais ou não) fornecem *drivers* JDBC para seus produtos. Muitas aplicações Java desenvolvidas utilizam essa tecnologia para se conectar a SGBDs relacionais. Isso as torna independentes do SGBD, possibilitando a sua utilização com qualquer gerenciador de banco de dados para a qual exista um *driver* JDBC disponível.

Na sua forma mais simples, o C-JDBC trabalha com replicação total das bases de dados. Em cada nó do cluster, há uma réplica gerenciada por um SGBD. O módulo principal do C-JDBC, denominado *C-JDBC Controller* e que se encontra no nó de entrada do cluster, se conecta aos SGBDs via *drivers* JDBC. Além disso, ele próprio implementa um *driver* JDBC, que as aplicações clientes utilizam para realizarem suas conexões. Uma vez conectada, a aplicação passa a submeter suas transações ao C-JDBC, que as repassa aos SGBDs. A Figura 6 ilustra a arquitetura proposta.

O *C-JDBC Controller* provê funções de gerência de requisições, escalonamento de transações, balanceamento de carga, *cache* de resultados de consultas, tolerância a falhas (pontos de checagem e *log* para recuperação) e sincronização de réplicas. O balanceamento de carga é feito no momento da escolha do nó a ser utilizado para o processamento de cada transação. O nó com menor quantidade de tarefas a executar é escolhido.

O C-JDBC provê paralelismo inter-consultas no cluster de bancos de dados. Não há, no entanto, suporte a paralelismo intra-consulta. Cada consulta é enviada a apenas um nó, que a executa isoladamente. Isso pode ser uma desvantagem para sua utilização com sistemas OLAP.

Outras características relevantes do C-JDBC são o suporte a replicação parcial dos bancos de dados e a possibilidade de se utilizar vários *controllers* em um mesmo

cluster, com diferentes tipos de combinação entre eles. A opção de vários *controllers* pode ser utilizada para aumentar tanto a escalabilidade do sistema quanto a sua tolerância a falhas.

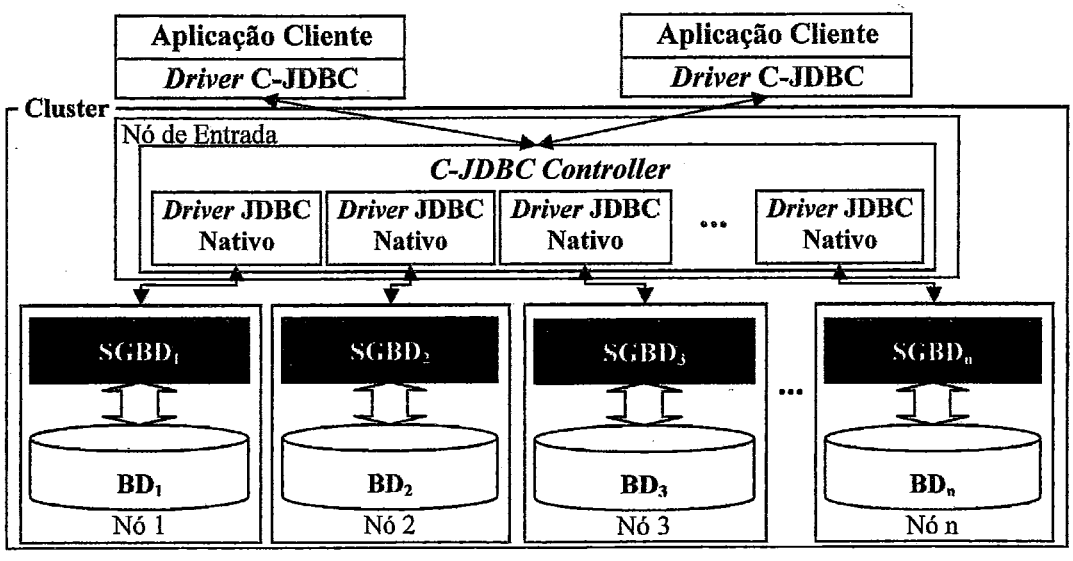

Figura 6 - C-JDBC - arquitetura básica

O C-JDBC é uma proposta bastante interessante, pois se baseia num padrão amplamente aceito ao mesmo tempo em que oferece uma solução não intrusiva para aumentar o desempenho de aplicações com muitos acessos a bancos de dados. Além disso, é de código aberto e não dependente de nenhum SGBD específico. Entretanto, de acordo com consulta realizada em novembro de 2004, o C-JDBC não oferece o paralelismo intra-consulta.

II.3.2. Extensões Proprietárias de SGBDs para Clusters

As soluções que descrevemos na seção anterior se baseiam em SGBDs "caixas-pretas", ou seja, são projetadas para funcionarem com qualquer SGBD. Nessa seção, descrevemos extensões proprietárias de SGBDs para clusters, ou seja, SGBDs que foram modificados para executarem suas funções em clusters.

II.3.2.1 MySQL Cluster

O SGBD MySQL (MYSQL, 2004b) é um software livre, de código aberto, disponível na Internet. Dadas as suas características de alto desempenho e simplicidade de uso, ele é hoje um dos SGBDs mais utilizados, inclusive em aplicações comerciais.

Recentemente, foi disponibilizada uma versão desse software para clusters, o MySQL Cluster (MYSQL, 2004a).

A proposta do MySQL Cluster é oferecer serviços de acesso a bancos de dados, com alta disponibilidade, em clusters de computadores. A arquitetura utilizada é a de memória distribuída, apesar de a documentação citar que há suporte para arquiteturas do tipo memória compartilhada e disco compartilhado.

O MySQL cluster define três tipos de nós: nó de banco de dados, nó de aplicação e nó de gerência. Os nós de banco de dados são os principais. Todos os dados são replicados e neles armazenados. São também os responsáveis pelas transações. A replicação faz com que a queda de um nó não inviabilize a utilização do sistema. Os nós de aplicação são responsáveis pela execução das aplicações que se conectam aos bancos de dados. Em caso de falha de um nó de banco de dados, a aplicação pode continuar sua execução se conectando à sua réplica. Os nós de gerência são responsáveis pela configuração do sistema. São utilizados apenas durante a inicialização e para reconfigurações. Não precisam estar ativos durante todo o tempo.

O MySQL cluster garante alto desempenho no processamento de transações por trabalhar exclusivamente com dados em memória principal e escritas assíncronas dos logs de transações. Para isso, exige que todo o banco de dados possa ser alocado nas memórias principais de todos os nós. Isso impõe um limite ao tamanho do banco, que deve ser igual à soma das capacidades das memórias.

O paralelismo inter-consultas é implementado no MySQL cluster. Não há porém implementação de paralelismo intra-consulta. Além disso, a atualização das réplicas é feita de maneira síncrona. Uma transação só é efetivada quando todos os nós de banco de dados que contêm réplicas dos dados atualizados a realizam completamente. Isso reduz a vazão do sistema.

II.3.2.2 Oracle Real Application Cluster 10g

O SGBD comercial Oracle (ORACLE, 2004), hoje na versão 10g, também possui uma extensão para clusters, denominada *Real Application Cluster* (RAC). Disponível como um produto à parte na versão anterior (9i), essa extensão é agora parte integrante do SGBD.

A arquitetura recomendada pela Oracle é a de disco compartilhado, onde os dados são armazenados em uma *Storage Area Network* (SAN) acessada por todos os nós do

cluster. Para conseguir alto desempenho, o SGBD aproveita a alta velocidade das redes de interconexão típicas de clusters de computadores para reduzir o número de acessos à SAN. Isso é conseguido através da tecnologia de "fusão de *caches*" (*cache fusion*), desenvolvida na versão 9i. Quando um nó necessita de uma página de disco que não se encontra em seu *cache*, envia uma solicitação aos outros nós antes de ir à SAN. Se algum nó possuir a página em seu *cache*, envia-a ao solicitante. Caso isso não ocorra, é necessária a solicitação da página à SAN.

Quanto ao processamento de consultas, o Oracle implementa tanto paralelismo inter-consultas quanto intra-consulta. Essa característica faz dele uma opção atraente tanto para aplicações OLTP quanto para aplicações OLAP. A possibilidade de utilizá-lo em clusters torna possível a obtenção de alto desempenho sem a necessidade de se investir em hardware paralelo. O SGBD Oracle porém ainda é um produto de alto custo, se comparado com outros SGBDs, principalmente os disponibilizados como software livre. A necessidade de se adquirir uma SAN também contribui para o alto custo de soluções baseadas nessa ferramenta.

II.3.2.3 IBM DB2 Integrated Cluster Environment

O IBM DB2 Integrated Cluster Environment (ICE) (DB2, 2004) é um produto integrado da empresa IBM constituído pelo SGBD IBM DB2 UDB (Universal Database) e pelo cluster IBM eServer Linux Cluster 1350. Outros produtos como gerente de armazenamento e servidor de aplicação (proprietários) podem ser facilmente integrados à solução, segundo o fabricante.

Trata-se de uma solução proprietária para a obtenção de alto desempenho com um SGBD adaptado para o ambiente de cluster. O IBM DB2 UDB provê suporte tanto a paralelismo inter-consultas como a paralelismo intra-consultas, o que o torna atraente para a aplicações OLAP e OLTP. No entanto, assim como o Oracle 10g, é uma solução de alto custo. Sua vantagem é não demandar a utilização de uma *Storage Area Network*, o que contribui para a redução do custo do investimento em *hardware*.

II.3.2.4 PGCluster

PGCluster (PGCLUSTER, 2004) é uma extensão para clusters do SGBD livre PostgreSQL (POSTGRESQL, 2004). Também é disponibilizada sob a forma de software livre.

A proposta dessa ferramenta é a de obtenção de paralelismo inter-consultas através da replicação total do banco de dados nos nós do cluster. A consistência das réplicas é feita de maneira síncrona e fica sob a responsabilidade de um componente denominado Servidor de Replicação (*Replication Server*). Quando uma operação de atualização é recebida por esse servidor, ele a encaminha a todos os nós. Se algum problema é detectado em um deles, as operações subseqüentes não são enviadas ao nó defeituoso até que o problema seja resolvido. Aumenta-se, assim, a disponibilidade do sistema.

Um Servidor de Balanceamento de Carga também faz parte do sistema e é o responsável por direcionar consultas aos nós com menor carga. Ao detectar problemas em algum dos nós, passa a desconsiderá-lo até a resolução do problema.

O PG cluster não implementa paralelismo intra-consulta, o que pode torná-lo pouco atraente para aplicações de processamento analítico.

II.3.2.5 Clusgres

O produto Clusgres (CLUSGRES, 2004) é um conjunto de bibliotecas que permitem a utilização do SGBD PostgreSQL em clusters de computadores. Trata-se de um produto comercial de propriedade do Linux Labs.

As principais características dessa ferramenta são a implementação de paralelismo inter-consultas e a transparência para as aplicações, que não precisam sofrer modificações. Aparentemente, a arquitetura utilizada é a de memória distribuída com replicação total do banco de dados, apesar de as informações no sítio do produto (CLUSGRES, 2004) não especificarem isso de maneira direta.

Uma restrição importante desse produto é a exigência de hardware especial (Dolphin *interconnect system*) para interconexão entre os nós a fim de simular ambiente de memória compartilhada. Além disso, ele também não implementa paralelismo intra-consulta.

II.4. Conclusão

Entre os tipos de paralelismo tradicionalmente oferecidos em sistemas paralelos de bancos de dados, o intra-consulta é o que consegue melhor desempenho para o processamento de consultas de alto custo, como as típicas de aplicações OLAP. Quanto

às arquiteturas, as que se mostram mais flexíveis e com menos propensão a problemas de balanceamento de carga são a de disco compartilhado e a que combina memória distribuída com replicação total. Sua flexibilidade as torna atraentes em ambientes para processamento de consultas *ad-hoc*. A primeira exige porém hardware específico para armazenamento de dados com alta velocidade de acesso. Além disso, para operações de modificação na base de dados, algoritmos complexos de coerência entre os *caches* dos nós e o disco são necessários. A segunda opção parece ser então menos dispendiosa e mais genérica.

A técnica de fragmentação virtual simples se mostra, nesse contexto, bastante atraente, na medida em que tenta aliar os benefícios do paralelismo intra-consulta à flexibilidade da arquitetura de memória distribuída com replicação total no processamento de consultas OLAP. Ela possui porém algumas limitações que a tornam vulnerável a idiossincrasias do SGBD utilizado no cluster.

Analisando os trabalhos relacionados, podemos notar que o interesse pela utilização de clusters de bancos de dados é crescente e excelentes ganhos de desempenho têm sido relatados. Porém, o único que apresenta proposta para paralelismo intra-consulta é o PowerDB, onde surgiu a proposta da SVP. Entre os produtos especialmente desenvolvidos para clusters, temos paralelismo intra-consulta implementado no Oracle 10g. No caso do Oracle, a arquitetura de disco compartilhado é utilizada e é exigida a utilização de uma SAN para armazenamento dos dados no cluster. SANs são normalmente produtos de elevado custo, o que, aliado ao custo do próprio SGBD, faz com que essa solução seja bastante dispendiosa. Isso contribui ainda mais para tornar a fragmentação virtual em clusters de bancos de dados uma solução atraente do ponto de vista custo/benefício.

Finalmente, considerando a necessidade de uma solução de alto desempenho para consultas de processamento intenso e demorado, como OLAP, sem incorrer em um investimento dispendioso, consideramos o uso de clusters de banco de dados com paralelismo intra-consulta a solução mais indicada. A análise dos trabalhos relacionados nos mostra que apenas o PowerDB oferece tal solução, que denominamos SVP. Entretanto, a proposta do PowerDB falha em diversas situações onde o comportamento dos SGBDs sobre fragmentos de grande tamanho foge ao controle do cluster de BD tornando a solução paralela pior do que a solução seqüencial. Assim, surge a motivação para esta tese, onde apresentamos uma solução de paralelismo intra-consulta sobre

clusters de BD sem os problemas de falta de controle da SVP, além de não ser dependente da distribuição uniforme de valores do atributo da fragmentação virtual.

No próximo capítulo, descrevemos a Fragmentação Virtual Adaptativa, nossa extensão para SVP que, aliada a uma estratégia de redistribuição dinâmica de trabalho, elimina várias das suas limitações.

III. Fragmentação Virtual Adaptativa com Redistribuição Dinâmica de Carga

A fragmentação virtual simples (SVP), proposta por AKAL *et al.* (2002), é uma técnica interessante para o processamento de consultas de alto custo porque tenta aliar a eficiência do paralelismo intra-consulta à flexibilidade da arquitetura de memória distribuída com replicação total de dados. Alguns requisitos são, no entanto, necessários para o sucesso da SVP. Neste capítulo, analisamos esses requisitos e algumas limitações provenientes dos mesmos no contexto do paralelismo intra-consulta. Em seguida, propomos técnicas para superar essas limitações.

III.1. Requisitos e Problemas da SVP

Consideremos uma consulta C a ser processada em n nós de um cluster através da SVP. Essa consulta acessa, entre outras, a relação R, escolhida para ser virtualmente fragmentada. O atributo de fragmentação virtual escolhido é r_{VP}. Cada sub-consulta de C (C_1, C_2, ..., C_n) acessará um fragmento virtual diferente de R (R_{VP1}, R_{VP2}, ..., R_{VPn}). O objetivo básico da SVP é fazer com que cada nó acesse apenas o subconjunto da relação virtualmente fragmentada definido pelo intervalo do seu fragmento virtual, que seria o ocorrido caso a fragmentação física (ou real) tivesse sido empregada. Além disso, é necessário que o resultado da execução paralela seja o mesmo da execução seqüencial da consulta. Isso será conseguido se os requisitos a seguir forem satisfeitos.

Requisito 1 - A fragmentação virtual de R deve ser completa e disjunta.

Os conceitos de fragmentação virtual de relações completa e disjunta foram definidos na seção II.2.2 (Definição 3 e Definição 4, respectivamente). A fragmentação completa garante que todas as tuplas de R estão presentes em algum fragmento. A fragmentação disjunta, por sua vez, garante que cada tupla pertence a apenas um fragmento. Com isso, não há a possibilidade de que alguma tupla seja processada mais de uma vez, o que, se ocorresse, poderia provocar erros no resultado final.

Requisito 2 - As tuplas da relação virtualmente fragmentada devem estar fisicamente ordenadas de acordo com o atributo de fragmentação.

Um dos objetivos básicos da fragmentação virtual é, assim como a fragmentação física, fazer com que apenas parte da relação original seja acessada. No caso da fragmentação física isso é trivial, uma vez que os dados já se encontram divididos em subconjuntos distintos. Na fragmentação virtual, porém, não há divisão física. Todos os dados pertencem à mesma relação. As tuplas que formam um mesmo fragmento virtual podem se encontrar espalhadas em todos os blocos de disco alocados para a relação. Caso isso ocorra, o nó responsável pelo processamento de tal fragmento deveria ler do disco todos esses blocos, reduzindo a eficiência da estratégia.

Para que isso não ocorra, é necessário que as tuplas estejam ordenadas fisicamente de acordo com o atributo de fragmentação. Isso reduziria o número de blocos lidos por cada nó ao acessar seu fragmento virtual. Além disso, a maioria dos blocos lidos conteria apenas dados relevantes. As exceções são o primeiro e o último blocos de cada fragmento, que podem conter tuplas dos fragmentos anterior e posterior, respectivamente.

Requisito 3 - Deve haver um índice para a relação virtualmente fragmentada baseado no atributo de fragmentação.

A ordenação física das tuplas contribui para que, uma vez encontrada a primeira tupla do seu fragmento virtual, cada nó do cluster leia o número mínimo de blocos necessários para a obtenção das restantes. É preciso porém que haja um meio eficiente para a localização da primeira tupla. Uma busca binária poderia ser empregada, uma vez que os dados se encontram ordenados. A maioria dos SGBDs porém não utiliza essa técnica para localizar tuplas em uma relação. Por isso, enumeramos entre os requisitos a necessidade da existência de um índice baseado no atributo de fragmentação.

Todos os SGBDs implementam índices como estruturas auxiliares de acesso. E, na sua ausência, optam por executar uma busca linear (GRAEFE, 1993) sobre a relação a ser consultada. Uma busca linear executada pelo nó responsável pelo primeiro fragmento virtual não traria tanto problema, uma vez que o SGBD poderia abandonar o processo tão logo a primeira tupla não pertencente ao fragmento fosse encontrada, o que é possível se os dados já se encontrarem ordenados. No entanto, o nó responsável pelo último fragmento deveria percorrer todos os blocos da relação até encontrar a primeira

tupla do seu fragmento. Percorreria então os blocos restantes para obter as demais tuplas. Como resultado, toda a relação seria lida do disco.

Requisito 4 - O SGBD deve obrigatoriamente utilizar o índice para a obtenção das tuplas pertencentes a cada fragmento virtual.

A ordenação física de tuplas segundo o atributo de fragmentação e a existência de um índice sobre o mesmo são condições necessárias mas não suficientes para garantir que cada nó acessará apenas os blocos da relação que contêm dados relativos ao seu fragmento virtual. Os otimizadores de consultas de vários SGBDs seguem uma heurística que os faz optar pela utilização do índice para acesso a uma relação apenas se o número estimado de tuplas a recuperar estiver abaixo de um certo limite. Caso esse limite seja ultrapassado, ou seja, caso o predicado que determina o fragmento virtual não seja suficientemente seletivo, a busca linear é escolhida para a obtenção dos dados. Esse limite vem normalmente pré-determinado pelo fabricante e a maioria dos SGBDs permite a sua modificação pelo administrador do banco de dados.

Essa característica dos SGBDs pode se tornar um grande problema para a utilização da SVP, principalmente se o objetivo é a construção de um processador de consultas para clusters que funcione com qualquer SGBD como componente "caixa-preta". Isso se dá pelo fato de que o tamanho do fragmento virtual é determinado pelo número de nós empregados no processamento da consulta. Se esse número não for suficiente para gerar fragmentos pequenos o bastante (dentro do limite adotado pelo otimizador do SGBD), a busca linear pode ser adotada em algum ou até mesmo em todos os nós. Uma opção seria mudar o limite descrito acima no momento da execução da consulta. A forma pela qual isso é feito, no entanto, varia de acordo com o SGBD, o que reduziria a independência da SVP. Além disso, essa mudança afetaria todas as consultas, inclusive aquelas que poderiam se beneficiar da busca linear.

Requisito 5 - Os valores do atributo de fragmentação devem ser uniformemente distribuídos entre as tuplas da relação virtualmente fragmentada.

Como descrito na seção II.2.2, a determinação dos fragmentos virtuais é feita pela SVP de maneira bem simples. A SVP divide o tamanho do intervalo de valores do atributo de fragmentação pelo número de nós do sistema utilizados para o processamento da consultas a fim de que se obtenha o tamanho de cada fragmento. Caso

os valores do atributo de fragmentação sejam distribuídos uniformemente entre as tuplas da relação fragmentada, os fragmentos possuirão todos praticamente o mesmo tamanho e um bom balanceamento de carga inicial poderá ser obtido. No entanto, esse requisito pode ser bastante difícil de se satisfazer.

Em muitas aplicações, é comum a existência de distorção de dados (*data skew*), em particular, distorção relacionada a valores de atributos, que ocorre quando os valores de alguns atributos não se encontram distribuídos uniformemente entre as tuplas de uma relação (WALTON *et al.*, 1991). A distorção de valores de atributos pode levar a SVP a produzir fragmentos virtuais de tamanhos muito diferentes no que diz respeito ao número de tuplas. Apesar dos tamanhos dos seus intervalos serem iguais, um número diferente de tuplas pode pertencer a cada um deles, gerando desbalanceamento inicial de carga entre os nós do cluster. Quanto maior a distorção, mais severo o desbalanceamento.

Alguns dos requisitos descritos podem ser satisfeitos com relativa facilidade. Outros, no entanto, só o podem se o SGBD utilizado no cluster e os dados por eles gerenciados possuírem certas características. Os requisitos 1, 2 e 3 se encontram no primeiro caso. O requisito 1 é trivial e a estratégia de determinação de fragmentos virtuais proposta pela SVP o satisfaz. Qualquer variação da SVP que determine intervalos para os fragmentos da mesma maneira também o satisfará. Os requisitos 2 e 3 podem ser satisfeitos utilizando-se recursos providos pela maioria dos SGBDs. Vários SGBDs possuem recursos para a ordenação física de tuplas de uma relação segundo um (ou vários) atributo(s). Além disso, possuem ainda facilidades para a criação de índices.

Os requisitos 4 e 5 porém pertencem ao segundo caso. Quanto ao requisito 4, não há maneira padronizada de se forçar a utilização de um índice por parte do SGBD. Até onde sabemos, isso só é possível através da utilização de funções específicas, que variam de SGBD para SGBD. Assim, caso algum fragmento virtual não seja pequeno o suficiente, o SGBD pode optar por processá-lo através da busca linear. Isso foge ao controle do cluster de bancos de dados e torna a SVP vulnerável às características do SGBD, ao número de nós utilizados para processar a consulta e à distribuição de tuplas entre os fragmentos.

A satisfação do requisito 5 também não é trivial e deixa a SVP vulnerável a eventuais problemas de distorção de dados. Caso a distribuição de tuplas entre os

fragmentos não seja uniforme, alguns nós receberão maior carga de trabalho do que outros, o que pode afetar negativamente o desempenho.

Notamos então que há basicamente dois problemas não resolvidos pela SVP, que especificamos expressamente a seguir para facilitar referências futuras:

Problema 1 - Grandes Fragmentos Virtuais: esse problema pode levar à não utilização de índice por parte do SGBD para a obtenção das tuplas do fragmento virtual.

Problema 2 - Distorção de Dados: a distorção de valores de atributos pode levar à atribuição de cargas iniciais desiguais para os nós do cluster. Outros tipos de distorção (WALTON *et al.*, 1991) podem levar a problemas desbalanceamento de carga mesmo que a carga inicial seja igualmente distribuída.

Na próxima seção, apresentamos uma abordagem inovadora para a fragmentação virtual que mantém a flexibilidade e a simplicidade da SVP. Essa abordagem serve como base para a proposta de uma nova estratégia de processamento de consultas, que denominamos fragmentação virtual adaptativa.

III.2. Uma Nova Abordagem para a Fragmentação Virtual

Ambos os problemas da SVP provêm do fato de que, ao processar uma consulta, ela utiliza um número de fragmentos virtuais equivalente ao número de nós do cluster. Conseqüentemente, se n nós são usados, n sub-consultas são geradas e, a cada nó, é atribuída apenas uma delas. Essa abordagem de uma consulta por nó torna difícil a resolução dos problemas da SVP. Primeiramente, o tamanho dos fragmentos virtuais fica dependente do número de nós. Para evitar o problema de grandes fragmentos, deve-se utilizar um número de nós suficientemente grande. Isso pode não ser possível. Não é difícil imaginar casos em que mesmo a utilização de todos os nós disponíveis não geraria fragmentos pequenos o suficiente para evitar a varredura completa da relação virtualmente fragmentada em algum dos nós. Nesse caso, a solução do problema só é possível se o sistema for expandido com a aquisição de novos nós.

Na presença de distorção de dados, fragmentos com quantidades de tuplas bastante diferentes podem ser gerados, ocasionando problemas de desbalanceamento de carga, como visto anteriormente. Uma solução possível para o desbalanceamento é a

redistribuição dinâmica de carga, que consiste em retirar parte das tarefas de um nó e designá-las a outro, que já tenha finalizado seu próprio trabalho e se encontre disponível. Na SVP, isso não pode ser feito, pois cada nó processa apenas uma sub-consulta. Até onde nos é dado saber, não é possível (utilizando um SGBD como componente "caixa-preta"), interromper uma consulta, analisar seu status, modificá-la e solicitar ao SGBD que continue a execução a partir da nova versão. Sendo assim, a redistribuição dinâmica de carga não pode ser realizada.

Outra possível solução para os problemas causados por distorção de dados é utilizar uma estratégia para a determinação inicial dos fragmentos virtuais mais eficiente. No lugar de produzir fragmentos com intervalos de tamanhos iguais, poder-se-ia considerar estatísticas do banco de dados relativas à distribuição real de valores do atributo de fragmentação entre as tuplas. Assim, fragmentos com quantidades de tuplas aproximadamente iguais poderiam ser obtidos. O inconveniente dessa alternativa é o aumento da complexidade do algoritmo de fragmentação virtual inicial. Como essa fase é executada seqüencialmente, é desejável que ela seja bastante simples, a fim de não causar impacto no desempenho do sistema. Além disso, a obtenção de estatísticas de dados a partir dos catálogos dos SGBDs não é padronizada. Isso reduziria a interoperabilidade do processador de consultas que utilizasse a SVP, que deveria possuir um módulo coletor de estatísticas específico para cada SGBD que se desejasse utilizar. Alternativamente, estatísticas mais simples, obtidas através de consultas aos bancos de dados, poderiam ser obtidas pelo próprio processador de consultas. A atualização periódica dessas estatísticas seria necessária, como ocorre nos próprios SGBDs.

Dada a complexidade das soluções apresentadas, decidimos propor uma nova abordagem para a fragmentação virtual que não utiliza a estratégia de uma sub-consulta por nó proposta pela SVP. Nossa proposta é executar uma fragmentação virtual de granularidade mais fina do que a empregada pela SVP ou, em outras palavras, utilizando pequenos fragmentos, em quantidade tipicamente maior do que o número de nós utilizados para o processamento da consulta. Esquematicamente, a estratégia está representada na Figura 7, para um cluster de quatro nós.

A idéia da utilização de pequenos fragmentos é simples. Inicialmente, cada nó do cluster recebe uma sub-consulta e um intervalo correspondente a um fragmento virtual. O tamanho desse intervalo é calculado da mesma maneira proposta pela SVP. A sub-

consulta é parametrizada, ou seja, os valores correspondentes ao início e ao final do seu fragmento virtual são parâmetros. Todos os nós, então, recebem a mesma sub-consulta, cada uma com seu intervalo na forma de parâmetros. A partir daí, o processamento do fragmento virtual se dá de maneira diversa da proposta pela SVP.

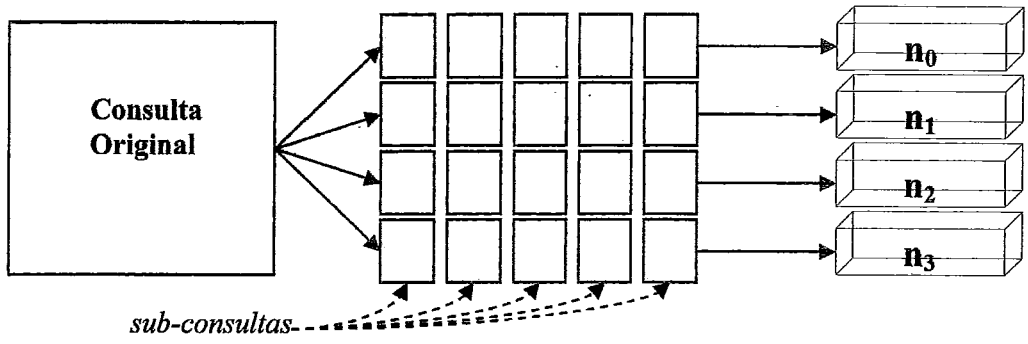

Figura 7 - Fragmentação virtual com granularidade fina

Suponha que um nó *i* tenha recebido um intervalo I_i e uma sub-consulta parametrizada C_i para processar o seu fragmento virtual. No lugar de processar inteiramente I_i através de uma única submissão da sub-consulta, o nó subdivide esse intervalo em *p* intervalos menores (I_{i0}, I_{i1}, I_{i2}, ..., I_{ip-1}). O valor de *p* deve ser tal que os sub-intervalos I_{ij} sejam pequenos o bastante. A sub-consulta C_i é então submetida seqüencialmente ao SGBD *p* vezes. Em cada uma delas, seus parâmetros são substituídos de forma a fazer com que cada sub-intervalo I_{ij} seja processado. Obviamente, a fragmentação virtual de I_i deve ser completa e disjunta. O processo é ilustrado na Figura 8.

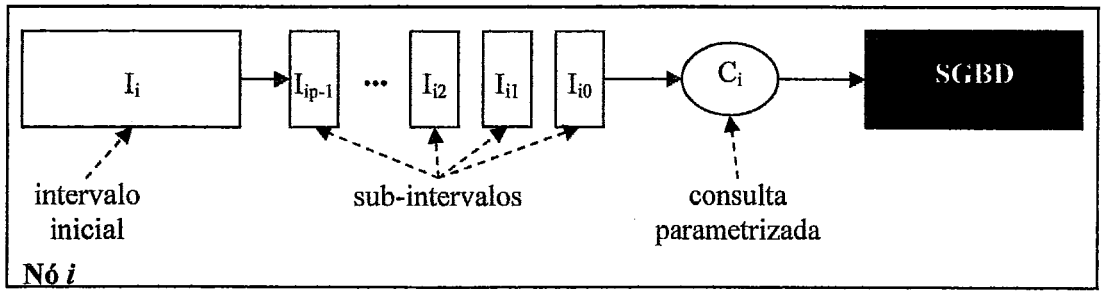

Figura 8 – Estratégia básica da fragmentação virtual com granularidade fina em um nó

Nessa nova abordagem, o fragmento virtual recebido por um nó não é processado através de uma única sub-consulta, mas sim através de várias. Não há mais a abordagem

de uma sub-consulta por nó, como na SVP. Analisemos como essa nova estratégia pode simplificar a resolução dos problemas da SVP.

Tratemos primeiramente do problema relacionado a grandes fragmentos virtuais. Ao dividir, em cada nó, o intervalo inicial em partes menores, encerramos a relação entre o tamanho do fragmento virtual e o número de nós do cluster. Os sub-intervalos I_{ij} podem ser suficientemente pequenos a fim de evitar seu processamento através de buscas lineares realizadas pelo SGBD sobre a relação fragmentada. O acréscimo de nós ao cluster para a resolução desse problema se torna desnecessário.

Nossa abordagem fornece ainda uma base para a resolução dos problemas de desbalanceamento de carga causados pela distorção de dados, sem tornar mais complexa a fase de determinação inicial dos fragmentos virtuais. Na verdade, essa fase permanece como na SVP. Se houver distorção de dados, a carga inicial de trabalho atribuída a cada nó ainda será desigual. Porém, a divisão do intervalo inicial em pequenos sub-intervalos torna possível a redistribuição dinâmica de carga. Caso algum nó processe inteiramente seu intervalo inicial e haja algum outro nó com sub-intervalos ainda a processar, um ou vários desses sub-intervalos poderiam ser retirados do nó ocupado e passados ao nó livre.

Como maneira de analisar a efetividade da nossa proposta, realizamos alguns testes experimentais utilizando algumas consultas do TPC-H em um cluster de bancos de dados gerenciados pelo PostgresSQL. Usamos um número arbitrário de sub-intervalos para cada consulta, uma vez que não possuíamos ainda um meio eficiente para determiná-lo. Além disso, utilizamos um banco de dados sem problemas de distorção, o que eliminou a necessidade de implementação de mecanismos para balanceamento dinâmico de carga. Os resultados dos experimentos são mostrados em LIMA *et al.* (2004a). A nova estratégia mostrou melhor desempenho do que a SVP na grande maioria dos casos considerados, chegando a ter desempenho seis vezes melhor em alguns casos. A utilização de pequenos fragmentos conseguiu evitar a realização de buscas lineares. Além disso, fez com que algoritmos de junção mais eficientes fossem utilizados nas consultas. Em muitos casos, diminuiu o tamanho das estruturas temporárias utilizadas para o processamento de certas operações, como agrupamentos e ordenações, o que reduziu o número de operações de entrada e saída. Esses resultados indicaram que a abordagem de fragmentação com granularidade fina era bastante promissora.

Os resultados obtidos nos incentivaram a tentar utilizar essa estratégia em um processador de consultas para cluster. Havia porém um problema em aberto: o que é um fragmento "pequeno o bastante"? No trabalho mencionado, utilizamos tamanhos arbitrários para os fragmentos, pois só pretendíamos provar nossos conceitos. A implementação de uma técnica para processamento de consultas exige a resolução desse problema.

Nesse sentido, continuamos o trabalho e chegamos à Fragmentação Virtual Adaptativa (AVP), uma técnica que usa fragmentação virtual de granularidade fina e uma abordagem adaptativa para determinar e mudar dinamicamente os tamanhos dos fragmentos virtuais processados por cada sub-consulta. Na próxima seção, descrevemos a AVP.

III.3. Fragmentação Virtual Adaptativa

Uma das principais questões na criação de um algoritmo que implemente a fragmentação virtual com granularidade fina é a determinação dos tamanhos dos fragmentos virtuais de cada sub-consulta. No caso extremo, pode-se pensar em utilizar fragmentos de tamanho unitário, pois eles seriam os menores possíveis. No entanto, o resultado pode ser ruim. O custo total de processamento de uma consulta é determinado por uma série de componentes (ÖZSU e VALDURIEZ, 1999). Antes de iniciar o acesso aos dados, há o custo de geração do plano de execução da consulta, que inclui a sua compilação e otimização. Em seguida, há o custo de iniciar o processo, envolvendo tarefas como, por exemplo, alocação de memória e criação de estruturas de dados temporárias. Só então, os dados podem ser lidos e a consulta, processada. Quando não há mais dados a processar, os resultados devem ser enviados ao requisitante e os recursos do sistema, liberados. De modo geral, essas são as fases de processamento de uma consulta.

A quantidade de dados acessados por uma consulta não afetam essas fases da mesma maneira. A geração do plano de execução, por exemplo, está mais relacionada à complexidade das operações realizadas pela consulta do que ao número de bytes lidos do disco. Porém, a fase de liberação de recursos pode ser mais ou menos custosa, dependendo da quantidade de dados acessados e das operações executadas.

Vejamos como isso afeta a nossa estratégia. Se produzirmos um grande número de pequenos intervalos (o que aumenta o número de sub-consultas), operações cujo custo não é afetado pelo número de tuplas processadas (como a geração do plano de execução) irão predominar e o desempenho será prejudicado. Por outro lado, um pequeno número de grandes intervalos (menos sub-consultas) pode acarretar em todos os problemas descritos anteriormente (buscas lineares, estruturas temporárias armazenadas em disco, etc). No caso da nossa abordagem, esses problemas podem prejudicar o desempenho ainda mais do que na SVP, pois o número de sub-consultas geradas pela primeira é maior do que o da segunda.

Uma solução possível é tentar calcular o tamanho de cada sub-intervalo utilizando informações contidas no catálogo do SGBD, como cardinalidade das relações envolvidas na consulta, distribuições de valores dos atributos utilizados, índices existentes, etc. Porém, o fato de estarmos utilizando SGBDs como componentes do tipo "caixa-preta" dificulta a implementação dessa alternativa. Geralmente, SGBDs diferentes possuem estruturas de catálogos diferentes. O acesso às informações aí contidas exigiria então o desenvolvimento de módulos específicos para cada SGBD que se desejasse utilizar com o nosso processador de consultas. Mesmo que se opte por essa solução, os otimizadores de consultas de diferentes SGBDs costumam trabalhar de maneiras diferentes. Seria difícil determinar com precisão quais estruturas de acesso e algoritmos seriam efetivamente utilizados durante o processamento de cada sub-consulta. Teríamos que trabalhar com suposições e esperar que essas suposições estivessem corretas. Além disso, antes de iniciar o investimento em uma solução desse tipo, algumas questões deveriam ser respondidas. Não estamos certos a respeito da existência de um tamanho de intervalo "ideal", que deva ser utilizado em todas as sub-consultas de uma consulta. Talvez sub-consultas acessando sub-conjuntos de tuplas diferentes devam utilizar intervalos de tamanhos diferentes. Há uma série de fatores que devem ser investigados nesse caso, como seletividade do predicado da consulta no intervalo considerado, número de relacionamentos de cada tupla, etc. Essas considerações aumentariam bastante a complexidade de um algoritmo que tentasse calcular os tamanhos dos intervalos ou fragmentos virtuais, o que poderia prejudicar o desempenho do sistema.

No lugar de tentar responder a todas essas questões, optamos por uma abordagem mais simples e dinâmica para o problema dos tamanhos dos fragmentos virtuais. Nós a

denominamos "Fragmentação Virtual Adaptativa" (AVP – *Adaptive Virtual Partitioning*) e a propusemos em LIMA *et al.* (2004b). Para evitar comunicação entre os nós para determinação dos tamanhos dos fragmentos, implementamos a AVP de maneira que cada nó a execute localmente, de maneira independente dos demais. Descrevemos a seguir o funcionamento do algoritmo em um nó. Na próxima seção, daremos uma visão geral do processo, combinando a AVP com as demais fases de processamento, incluindo um algoritmo para redistribuição dinâmica de carga.

O processo se inicia quando um nó recebe uma sub-consulta parametrizada e um intervalo para ser processado através dela, o que corresponde a um fragmento virtual. O intervalo inicial enviado para cada nó é calculado como na SVP. Então, o nó executa os seguintes passos:

1. Inicie utilizando um tamanho de fragmento muito pequeno que comece com o primeiro valor do intervalo designado para o nó. No caso extremo, pode-se iniciar com um intervalo de valor unitário. Na prática, podemos utilizar intervalos maiores;
2. Execute a sub-consulta utilizando esse tamanho de fragmento e meça o tempo de execução;
3. Aumente o tamanho do fragmento;
4. Execute a sub-consulta utilizando esse novo tamanho de fragmento e meça o tempo de execução. O novo fragmento deve começar a partir do limite final do fragmento anterior;
5. Se o aumento no tempo de execução com o novo tamanho for proporcionalmente menor do que o tempo obtido com o tamanho anterior, retorne ao passo 3;
6. Pare de aumentar o tamanho dos fragmentos. Um tamanho estável foi encontrado;
7. Continue utilizando o mesmo tamanho e medindo o tempo de execução até que ocorra uma deterioração de desempenho ou até que o limite final do fragmento original seja atingido. Nesse último caso, interrompa o processo, pois o fragmento foi totalmente processado;
8. Se a deterioração for confirmada, i.e., se há execuções consecutivas com acréscimo no tempo, diminua o tamanho do fragmento e retorne ao passo 2. Senão, retorne ao passo 7.

Os passos acima ilustram os princípios básicos da AVP. Ao iniciarmos com um intervalo muito pequeno, evitamos buscas lineares no início do processo. Para agirmos de maneira diferente, deveríamos saber o limite a partir do qual o SGBD opta por buscas lineares no lugar da utilização de índice para acesso à relação virtualmente fragmentada.

Quando aumentamos o tamanho do fragmento virtual, monitoramos o tempo de execução da sub-consulta. Nosso objetivo é determinar o tamanho de intervalo a partir do qual a quantidade de tuplas acessadas passa a influenciar de maneira significativa o custo de processamento das sub-consultas. Com isso, diminuímos a influência das fases do processamento da sub-consulta cujos custos não são influenciados pela quantidade de dados manipulados (geração do plano de execução, por exemplo). Se, por exemplo, ao dobrarmos o tamanho do fragmento, obtivermos um tempo de execução para a sub-consulta próximo do dobro do tempo obtido com o tamanho anterior, há uma boa chance de que tenhamos encontrado esse ponto. Interrompemos então o aumento no tamanho do fragmento. Continuar aumentando o tamanho a partir daí pode levar-nos a fragmentos muito grandes e aos problemas de execução de buscas lineares e/ou criação de estruturas temporárias baseadas em disco.

Poderíamos supor que o tamanho de fragmento encontrado nesse ponto do algoritmo é o ideal e não deve sofrer alterações. Essa suposição, no entanto, poderia não estar correta. O bom tempo de execução da sub-consulta com o tamanho encontrado pode ter sido conseqüência da utilização exclusiva de dados presentes no *cache* do SGBD. Ou ainda, o fragmento virtual pode corresponder a um intervalo para o qual a relação não possui tuplas, ou as possui em pouca quantidade, o que provocaria uma execução muito rápida. Em resumo, o tamanho "ideal" poderia ter sido considerado como tal por mero acaso, ou seja, porque as condições do sistema sob as quais foi avaliado lhe favoreceram.

Por esse motivo, não interrompemos o monitoramento dos tempos de execução das sub-consultas. Se as execuções subseqüentes apresentam deterioração de desempenho, há a possibilidade de que o tamanho utilizado seja maior do que o recomendado. Daí a importância do passo 8. Ele representa uma oportunidade para que tamanhos menores sejam experimentados. Em caso de deterioração, diminuímos o tamanho do fragmento e reiniciamos todo o processo.

Como pode ser notado, a AVP não utiliza estatísticas ou quaisquer outras informações de catálogo do SGBD durante sua execução. Ela é totalmente baseada em observações feitas a respeito do tempo de execução das sub-consultas, que utiliza para adaptar os tamanhos dos fragmentos virtuais. Com essa abordagem, não há necessidade da determinação de um tamanho de fragmento ideal, único para todas as sub-consultas, o que elimina a necessidade de comunicação entre os nós para a execução do processo. Além disso, ela permite que o processamento de consultas se adapte às condições gerais de carga do nó em que está sendo executada. Normalmente, há vários processos simultâneos sendo executados em cada nó, especialmente os do sistema operacional. Com isso, tamanhos utilizados em um nó podem não ser ideais para outros, o que é mais um fator contra a tentativa de determinação de um tamanho global a ser utilizado por todos os nós participantes da execução da consulta.

A AVP, como implementação da fragmentação virtual com granularidade fina, resolve o problema de grandes fragmentos virtuais da SVP. Na próxima seção, descrevemos como a AVP pode ser utilizada para o processamento de consultas, e como sua utilização em conjunto com um algoritmo de redistribuição dinâmica de carga resolve os problemas de desbalanceamento ocasionado pela distorção de dados.

III.4. Processamento de Consultas com Fragmentação Virtual Adaptativa

Nessa seção, mostramos nossa estratégia para execução de consultas de alto custo em clusters de bancos de dados. Ela utiliza a AVP em conjunto com uma técnica para redistribuição dinâmica de trabalho, tornada possível devido ao abandono da abordagem de "uma sub-consulta por nó" da SVP.

Quando uma consulta é recebida pelo processador de consultas do cluster, um nó é escolhido como o "coordenador" do seu processamento. O coordenador é o responsável pela distribuição de carga inicial entre os nós envolvidos no processo, denominados nós "participantes". O processamento da consulta se dá então em quatro fases:

1. **Fragmentação virtual inicial**: atribui a cada nó um fragmento virtual. O processo ocorre do mesmo modo proposto pela SVP, ou seja, todos os nós recebem fragmentos virtuais com intervalos de tamanhos idênticos.

2. **Ajuste dos tamanhos dos fragmentos**: fase durante a qual as sub-consultas são processadas pelos nós utilizando a AVP. Ela se destina a resolver o problema de grandes fragmentos virtuais.
3. **Redistribuição de carga**: utiliza uma técnica dinâmica para a redistribuição de carga entre os nós. Seu objetivo é resolver problemas ocasionados pela distorção de dados.
4. **Encerramento**: finaliza a execução paralela da consulta.

O processo se inicia com o coordenador executando seqüencialmente a fase um. Para torná-lo o mais rápido possível, utilizamos a mesma estratégia da SVP. Em seguida, as fases dois e três são executadas paralelamente pelos nós participantes, de forma que a técnica de redistribuição dinâmica de carga possa se beneficiar dos pequenos fragmentos produzidos pela AVP. A fase quatro é executada ao final.

As técnicas utilizadas nas fases um e dois já foram discutidas anteriormente. Por isso, no restante dessa seção, descreveremos o processo de redistribuição dinâmica de carga e o processo de encerramento da consulta.

III.4.1. Redistribuição de Carga

Em caso de distorção de dados, a simples utilização da AVP pode levar aos mesmos problemas de desbalanceamento de carga da SVP. Se os fragmentos iniciais forem de tamanhos desiguais quanto ao número de tuplas, alguns nós poderão ter uma carga de trabalho maior do que a de outros. Para lidar com esses problemas, adotamos a estratégia de redistribuir dinamicamente a carga dos nós durante o processamento da consulta.

Na AVP, o fragmento virtual associado a um nó é progressivamente processado através de uma seqüência de sub-consultas, cada uma correspondendo a uma pequena parte do fragmento inicial. Isso nos dá oportunidade de interferir e modificar a carga de trabalho do nó antes que seu fragmento inicial seja completamente processado. Podemos fazer isso dando simplesmente parte do intervalo não processado de um nó ocupado para um nó livre, i.e., que já processou inteiramente o seu fragmento inicial.

Há várias abordagens possíveis para a execução dessa tarefa. A fim de evitar um possível ponto de contenção causado por uma abordagem centralizadora (onde um nó seria responsável por todo o processo), decidimos adotar uma alternativa

completamente descentralizada e executada simultaneamente por todos os nós envolvidos no processamento da consulta. Ela se baseia em uma organização lógica dos nós e em um mecanismo de difusão de mensagens. Ambos são descritos nas seções a seguir. Após as descrições, detalhamos o algoritmo de redistribuição.

III.4.1.1 Organização Lógica dos Nós

A organização lógica que propomos considera que o cluster utilizado possui arquitetura e protocolos de rede que permitam que qualquer nó envie diretamente mensagens a qualquer outro nó. Se roteamentos são necessários, eles são transparentes para as aplicações que desejam enviar as mensagens.

No nosso algoritmo de redistribuição, cada nó deve difundir mensagens pelo cluster. Por "difusão" entendemos a capacidade de fazer com que a mesma mensagem seja recebida por todos os outros nós. No lugar de exigirmos a presença de serviços de rede para difusão de mensagens no cluster, nós propomos uma solução mais genérica (e portável) baseada no conceito de "grafo d-dimensional em malha" (WOJCIECHOWSKA, 1999).

Definição 5 (Grafo D-Dimensional em Malha) *Um grafo d-dimensional em malha* $G_{n0,n1,\ldots,nd-1}$ *é um grafo com* $n_0 * n_1 * \ldots * n_{d-1}$ *vértices, cada um identificado através de um, e somente um, elemento de* $\{(i_0,i_1,\ldots,i_{d-1}) \mid 0 \leq i_j < n_j,\ 0 \leq j < d\}$, *e com arestas conectando vértices que diferem em um, e exatamente em um, componente dos seus identificadores.*

Os n vértices designados para o processamento de uma consulta C são organizados logicamente como se fossem os vértices de um grafo bidimensional em malha $G_{p,q}$, onde $p = \lceil n / \lceil \sqrt{n} \rceil \rceil$ e $q = \lceil \sqrt{n} \rceil$. Se n possui raiz quadrada inteira, obtemos um grafo em malha quadrado. Se não, temos um grafo em malha retangular onde (p-q) = 1, ou seja, uma malha com formato próximo ao de um quadrado. Nosso interesse por malhas quadradas se dá porque nelas o diâmetro é minimizado. Como não simulamos barramentos para difusão de mensagens, essa é a configuração mais eficiente, se comparada a de outras malhas bidimensionais (BAR-NOY, PELEG, 1991). A Figura 9 ilustra como um conjunto de 25 nós seria organizado.

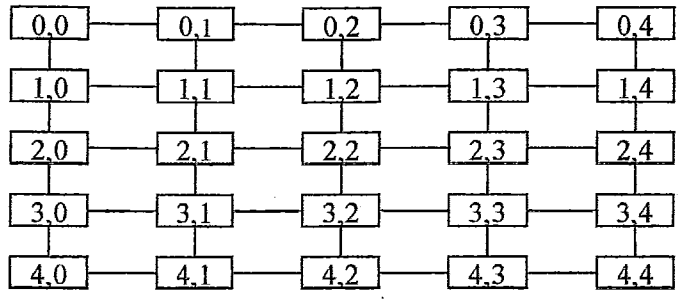

Figura 9 - Grafo bidimensional em malha com 25 nós

Quando organizados em uma malha, cada nó possui, no máximo, quatro vizinhos diretos, a saber: norte, sul, leste e oeste, baseados nas suas posições relativas.

III.4.1.2 Difusão de Mensagens

Para realizar a difusão de mensagens em nossa malha, é necessário um algoritmo que garanta as seguintes propriedades:

(1) Nenhuma mensagem será recebida mais do que uma vez por um mesmo nó; e

(2) O nó que origina uma mensagem consegue saber quando ela já foi difundida para todos os outros.

A propriedade (1) evita trabalho redundante. A propriedade (2) é útil para que o nó saiba quando reiniciar a difusão de uma mensagem.

Quando um nó deseja difundir uma mensagem, ele a envia a todos os seus vizinhos, juntamente com o seu identificador (do nó). Quando um nó recebe uma mensagem que deve ser difundida, ele a propaga utilizando o seguinte algoritmo:

1. Se a mensagem é recebida do vizinho norte ou sul, retransmita-a ao vizinho do lado oposto e aos vizinhos leste e oeste.
2. Se a mensagem é recebida do vizinho leste ou oeste, retransmita-a pelo lado oposto.
3. Se o nó se encontra na extremidade leste (oeste) da malha e recebe uma mensagem do vizinho oeste (leste), notifique o nó que originou a mensagem de que a mensagem não pode mais ser propagada.

O algoritmo é ilustrado na Figura 10. As notificações ao nó de origem foram omitidas para simplificar a figura.

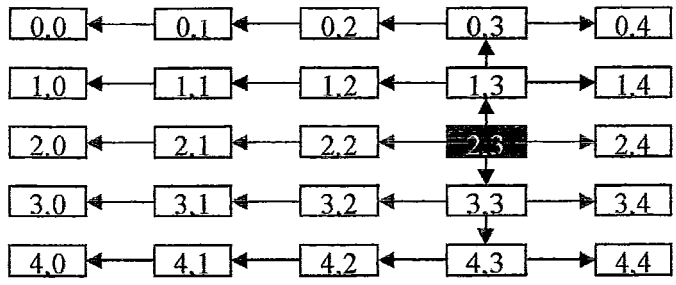

Figura 10 - Difusão de mensagem originada pelo nó (2,3) (as notificações ao nó de origem foram omitidas)

Vamos agora mostrar informalmente que esse algoritmo de difusão garante que cada nó recebe cada mensagem apenas uma vez (propriedade (1)). Nós iniciamos mostrando que nenhum nó pode receber uma mesma mensagem de vizinhos diferentes. Denominemos "fonte" o nó que inicia um processo de difusão. A identificação desse nó é (p,q).

A Figura 11 mostra todas as possíveis situações pelas quais uma mensagem pode chegar a um nó e ser propagada por ele. Observando-a, podemos estimar a região da malha na qual cada mensagem foi originada. Por exemplo, suponha que um nó (i,j) receba uma mensagem através do seu vizinho norte (i-1,j). Isso só pode ocorrer se (i-1,j) é a fonte da mensagem ou se ele também houver recebido a mensagem do seu vizinho norte (i-2,j). O mesmo raciocínio pode ser aplicado para o nó (i-2,j). Então, podemos concluir que a mensagem recebida por (i,j) através do seu vizinho norte tem como fonte o nó (L,j), onde $0 \leq L < i$. A regra para propagação de mensagens recebidas através do vizinho sul é simétrica a das recebidas através do vizinho norte. Então, uma mensagem recebida por (i,j) através do seu vizinho sul tem como fonte o nó (L,j), onde $i < L < p$.

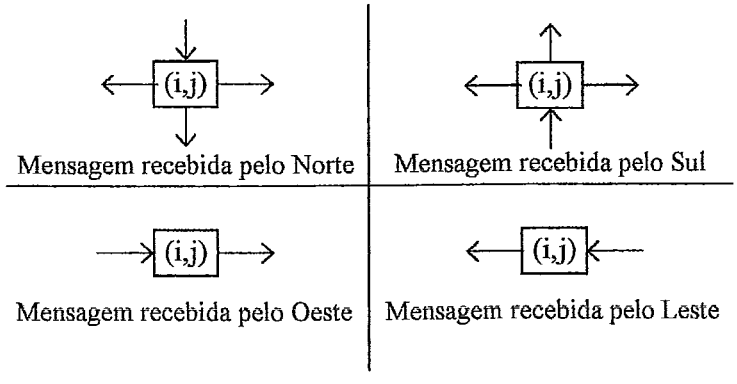

Figura 11 - Regras de propagação de mensagens

Suponha agora que o nó (i,j) receba uma mensagem através do seu vizinho oeste (i,j-1). (i,j-1) propaga uma mensagem para (i,j) se é a fonte, ou se a recebeu de (i,j-2), (i-1,j-1) ou (i+1,j-1). O mesmo raciocínio pode ser aplicado para (i,j-2). Para os outros dois nós, os casos de mensagens recebidas do norte e do sul podem ser aplicados. Concluímos então que uma mensagem recebida por (i,j) através do seu vizinho leste tem como fonte o nó (L,C), onde $0 \leq L < p$, $0 \leq C < j$. A regra para propagação de mensagens recebidas através do vizinho leste é simétrica a de propagação de mensagens recebidas através do vizinho oeste. Então, uma mensagem recebida por (i,j) através do vizinho leste tem como fonte o nó (L,C), onde $0 \leq L < p$, $j < C < q$.

Resumindo, dado um nó (i,j) e a direção através da qual uma mensagem *m* é por ele recebida, a fonte de *m* pertence a um dos seguintes conjuntos:

$S_{ij1} = \{(L,j) \mid 0 \leq L < i\}$, para mensagens vindas do norte;

$S_{ij2} = \{(L,j) \mid i < L < p\}$, para mensagens vindas do sul;

$S_{ij3} = \{(L,C) \mid 0 \leq L < p, 0 \leq C < j\}$, para mensagens vindas do oeste;

$S_{ij4} = \{(L,C) \mid 0 \leq L < p, j < C < q\}$, para mensagens vindas do leste.

Observando os conjuntos, podemos notar que

$$S_{ijm} \cap S_{ijn} = \varnothing,\ 1 \leq m,n \leq 4,\ m \neq n. \qquad \textbf{(1)}$$

e

$$(i,j) \notin S_{ijm},\ 1 \leq m \leq 4. \qquad \textbf{(2)}$$

Suponha agora que o nó (i,j) tenha recebido a mesma mensagem de dois vizinhos diferentes, i.e., de direções diferentes. Como cada mensagem possui apenas uma fonte, isso implica na presença simultânea da fonte em dois conjuntos S_{ij} diferentes, o que contradiz a equação (1). Sendo assim, um nó não pode receber uma mesma mensagem através de dois vizinhos diferentes.

Podemos também concluir que um nó fonte (i_s,j_s) não pode receber uma mensagem difundida por ele mesmo. Se isso ocorresse, então $\{\exists m,\ 1 \leq m \leq 4 \mid (i_s,j_s) \in S_{isjsm}\}$, o que contradiz a equação (2).

Se uma mensagem *m* é recebida por cada nó através de um único vizinho, podemos concluir que existe um único caminho possível para *m* da fonte até qualquer outro nó. O algoritmo de difusão determina que um nó fonte envie *m* apenas uma vez a cada vizinho. Como uma fonte nunca recebe suas próprias mensagens, *m* nunca poderá

percorrer duas vezes um mesmo caminho. Então, concluímos que cada mensagem é recebida por cada nó apenas uma vez.

O algoritmo também permite que o nó fonte saiba quando sua mensagem já foi difundida por toda a malha (propriedade (2)). Quando, de acordo com o algoritmo de propagação, um nó não pode propagar uma mensagem, ele envia uma notificação direta à fonte. Como o formato da malha não se modifica e cada nó conhece sua própria posição na mesma, é trivial para um nó calcular o número de notificações recebidas necessárias para se concluir que a mensagem foi completamente difundida. Por exemplo, na Figura 10, o nó fonte (2,3) deve receber dez notificações para concluir que sua mensagem foi difundida. Porém, se o nó (0,4) se torna uma fonte, precisa receber apenas cinco notificações. Isso ocorre porque ele se situa em uma extremidade da malha. Todos os nós situados na mesma coluna que (0,4) receberão sua mensagem através de um vizinho norte ou sul e sempre conseguirão propagá-la, de acordo com o algoritmo. Eles, então, não enviam notificações.

Podemos ter situações em que o número de nós utilizados para o processamento de uma consulta não permita a formação de uma malha perfeita, ou seja, com todas as linhas e colunas preenchidas. Nesses casos, a última linha pode não apresentar nós em todas as suas colunas. Essa situação é ilustrada pela Figura 12. Ela mostra um caso em que vinte e três nós são utilizados para o processamento de uma consulta. Podemos notar que a sua última linha não se encontra completamente preenchida.

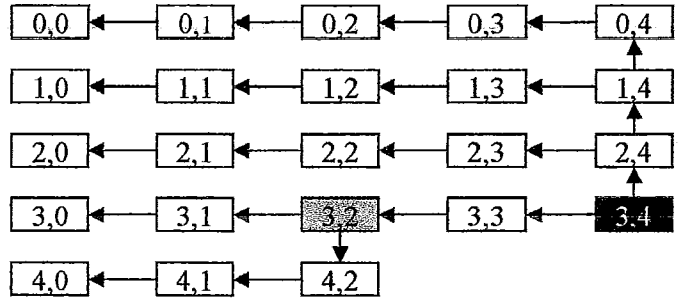

Figura 12 - Difusão de mensagem com fonte (3,4) em uma malha imperfeita

Em casos de malha imperfeita, a difusão de mensagens exige uma modificação no algoritmo. Suponha uma malha desse tipo com L linhas e C colunas, sendo que a última linha só possui nós para as primeiras *c* colunas. Os nós da última linha não receberiam mensagens que tivessem como fontes os nós (L-2,k), onde $c \leq k \leq C-1$. Na malha da

Figura 12, os nós da última linha não receberiam mensagens originadas por (3,3) e (3,4).

A resolução desse problema é simples. Como cada nó sabe sua identificação, o número total de nós utilizados e o número de linhas e colunas da malha, ele pode identificar facilmente que sua malha é imperfeita e se o seu vizinho sul é o último nó utilizado. Nesse caso, ele fica responsável por propagar também para esse vizinho as mensagens recebidas do leste. Na Figura 12, por exemplo, o nó (3,2) se encontra nessa situação. Ele passa a ser então o responsável por propagar para (4,2) as mensagens recebidas através de (3,3). O passo 2 do algoritmo deve ser então reescrito da seguinte forma:

2. Se a mensagem é recebida do vizinho leste ou oeste, retransmita-a pelo lado oposto. Se a malha é imperfeita e o vizinho sul é o último nó, retransmita a mensagem para o sul.

O número de mensagens de notificação a serem esperadas para se concluir que uma mensagem foi completamente difundida só sofre modificação nas malhas com apenas um nó na última linha. Nesse caso, esse nó só recebe mensagens pelo norte e, normalmente, não enviaria nenhuma notificação à fonte. O passo 1 deve ser então reescrito da seguinte maneira:

1. Se a mensagem é recebida do vizinho norte ou sul e o nó não é o único em sua linha de malha, retransmita-a ao vizinho do lado oposto e aos vizinhos leste e oeste. Se a mensagem é recebida do vizinho norte e o nó é o único presente na sua linha de malha, notifique o nó fonte de que a mensagem não pode mais ser propagada.

Obviamente, os nós deverão estar cientes da imperfeição da malha a fim de calcular o número correto de notificações esperadas.

III.4.1.3 Algoritmo de Redistribuição de Carga

Quando um nó participante recebe seu fragmento virtual, inicia o seu processamento utilizando a AVP. Quando seu fragmento foi completamente processado, notifica o coordenador e se torna "livre", continuando, porém, no conjunto de nós participantes. Nesse momento, ele começa a executar o algoritmo para redistribuição de carga, que é baseado em "ofertas de ajuda".

A idéia básica do algoritmo é: se um nó se torna livre, ele oferece ajuda aos outros nós. Denominamos esse nó "ofertante". O primeiro nó que aceita a ajuda dá parte da sua carga, i.e., parte do fragmento virtual restante, ao ofertante, que se torna então "ocupado" e não ajuda nenhum outro nó até que o novo fragmento recebido seja completamente processado.

O algoritmo utiliza três tipos de mensagens: "oferta de ajuda", enviada por um ofertante para indicar que ele se encontra disponível para ajudar outros nós; "aceitação de ajuda", enviada por um nó ocupado para indicar que aceita a oferta; e "extremidade de malha atingida", enviada por nós localizados nas extremidades leste e oeste da malha para indicar que não podem continuar propagando uma mensagem (favor ver algoritmo de difusão de mensagens).

O algoritmo de redistribuição funciona da seguinte maneira: quando um nó se torna livre, envia uma mensagem de oferta de ajuda para todos os seus vizinhos. Se um nó recebe uma oferta e também se encontra livre, propaga a oferta utilizando o algoritmo de difusão de mensagens. Caso contrário, se o nó está ocupado e não se encontra processando o último intervalo do seu fragmento virtual, ele envia ao ofertante uma mensagem de aceitação de ajuda e não continua a difusão da oferta. Vale notar que o nó ocupado não interrompe o processamento de sub-consultas enquanto espera uma resposta do ofertante. Ele envia a aceitação e continua processando o seu fragmento virtual.

Enquanto sua oferta está sendo propagada, o nó ofertante entra em estado de espera por alguma resposta. Quando a primeira aceitação é recebida, ele interrompe a espera e solicita ao nó que a enviou parte do intervalo do fragmento virtual ainda não processado. Ao recebê-lo, reinicia a execução da sub-consulta parametrizada utilizando a AVP e o novo intervalo. Outras mensagens de aceitação que cheguem nesse período são descartadas. Como os nós que as enviaram não se encontram em espera nem modificaram seus fragmentos virtuais, não há risco de que alguma parte deles não seja processada.

Há casos em que uma oferta pode ser completamente difundida na malha sem que o ofertante receba nenhuma mensagem de aceitação, apesar de ainda haver nós ocupados. Essa situação é indesejável, uma vez que representa desperdício do poder de processamento do cluster. Para evitar que isso ocorra, utilizamos a propriedade do algoritmo de difusão que permite aos nós saberem quando suas mensagens foram

completamente difundidas. Quando um ofertante detecta que sua oferta já foi recebida por todos os nós da malha, ele envia uma nova oferta a seus vizinhos. O processo se repete até que uma aceitação seja recebida ou que uma notificação seja recebida do coordenador indicando que a consulta foi completamente processada e o nó deve encerrar suas atividades. As mensagens de oferta de ajuda são numeradas seqüencialmente pelos seus ofertantes. Isso os ajuda a processar apenas as mensagens de extremidade de malha atingida relativa à última oferta propagada. Todas as demais são descartadas.

III.4.2. Encerramento

O encerramento do processamento de uma consulta é de responsabilidade do seu coordenador. Para que ele possa ser executado, é necessário que o coordenador tenha condições de saber se todos os fragmentos virtuais por ele produzidos na fase 1 foram processados. Para isso, definimos o conceito de nó "proprietário" de um fragmento virtual. Cada nó é proprietário do fragmento inicialmente recebido do coordenador. Ele é responsável por notificar o coordenador quando o seu fragmento for totalmente processado. Porém, devido à redistribuição dinâmica de carga, parte do seu fragmento pode ter sido processado por outros nós. Resolvemos esse problema da seguinte maneira: quando um nó ocupado envia parte do seu trabalho a um nó ofertante, ele envia também o identificador do proprietário do fragmento virtual que está sendo processado. Quando um nó termina o processamento de algum fragmento obtido dessa forma, envia uma mensagem direta ao proprietário, indicando o intervalo correspondente ao fragmento processado. O proprietário elimina então esse intervalo do fragmento de sua propriedade. Se ele detecta que não há mais intervalos desse fragmento a processar, notifica o coordenador. Quando o coordenador recebe notificações desse tipo de todos os proprietários, conclui que a consulta foi completamente processada. Ele então envia mensagens a todos os participantes a fim de que eles encerrem suas atividades.

Essa é a estratégia adotada pelo processador de consultas para clusters de bancos de dados que desenvolvemos. Na próxima seção, descrevemos a arquitetura desse processador.

IV. Arquitetura do Processador de Consultas

Nesse capítulo, descrevemos nossa proposta de arquitetura para um processador de consultas em cluster de bancos de dados. Primeiramente, descrevemos os principais componentes da arquitetura na seção IV.1. Em seguida, na seção IV.2, descrevemos o funcionamento dos módulos responsáveis pelas etapas do processamento de consultas em nível global. Para finalizar, descrevemos, na seção IV.3, os módulos responsáveis por esse processamento em nível local, ou seja, em cada nó.

IV.1. Visão Geral da Arquitetura

A Figura 13 mostra os principais componentes da arquitetura do processador de consultas para clusters que propomos nesta tese. Globalmente, existem dois objetos principais: o Processador de Consultas de Cluster (CQP – *Cluster Query Processor*) e o Intermediador. Para executar suas funções, o CQP utiliza metadados presentes no Catálogo, que possui informações do esquema do banco de dados necessárias à AVP: nomes das relações e dos seus atributos de fragmentação, com seus respectivos intervalos de valores. "Nó 1", ..., "Nó n" representam, na Figura 13, os *n* nós do cluster de PCs que hospeda nossa arquitetura. Em cada nó, é executada uma instância do Processador de Consultas de Nó (NQP – *Node Query Processor*), além do SGBD local, que gerencia uma réplica da base de dados.

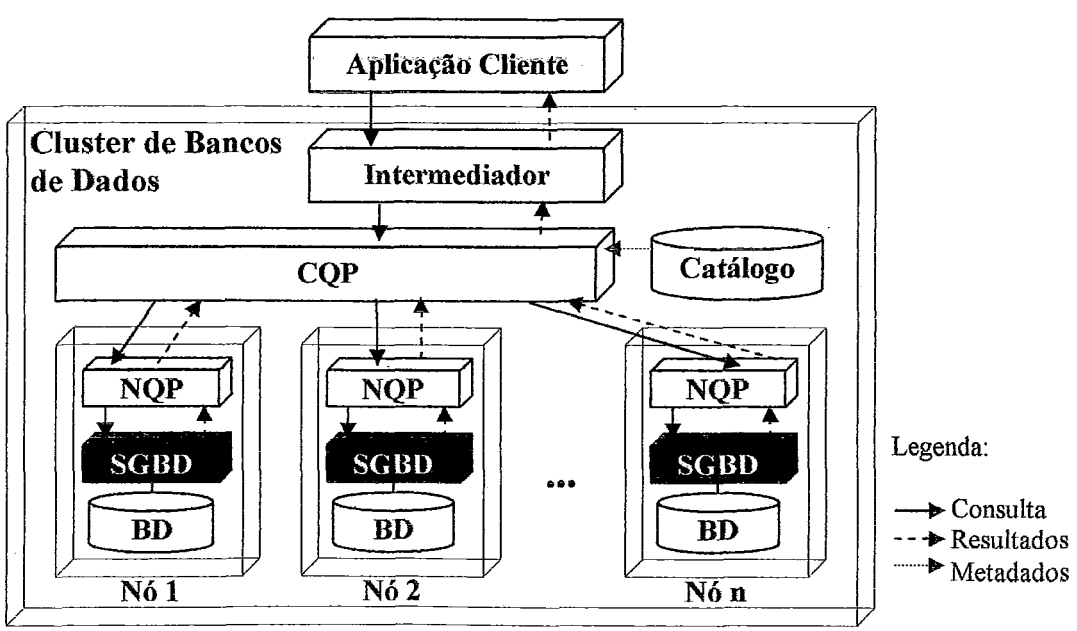

Figura 13 - Visão geral da arquitetura do processador de consultas para clusters

O Intermediador é o componente ao qual as aplicações clientes se conectam para submeter suas consultas. Normalmente, os clusters não permitem acesso direto a todos os seus nós. Eles possuem um nó para o qual é permitida a realização de conexões através de computadores externos. É nesse nó que o Intermediador deve ser executado, a fim de poder se comunicar com as aplicações. Os demais componentes se localizam nos nós internos do cluster.

A arquitetura do cluster representado na Figura 13 é a de memória distribuída. No entanto, os mesmos componentes poderiam ser utilizados em uma arquitetura de disco compartilhado. Não há exigência de que a base de dados esteja situada no mesmo nó do SGBD.

As fases do processamento de consultas da AVP – fragmentação virtual inicial, ajuste dos tamanhos dos fragmentos, redistribuição de carga e encerramento (favor ver seção III.4) – são executadas pelos componentes da arquitetura. O funcionamento desses componentes é detalhado nas seções a seguir.

IV.2.Processamento Global

Nessa seção, descrevemos as etapas globais do processamento de consultas e os componentes que as desempenham. A Figura 14 ilustra esse processo.

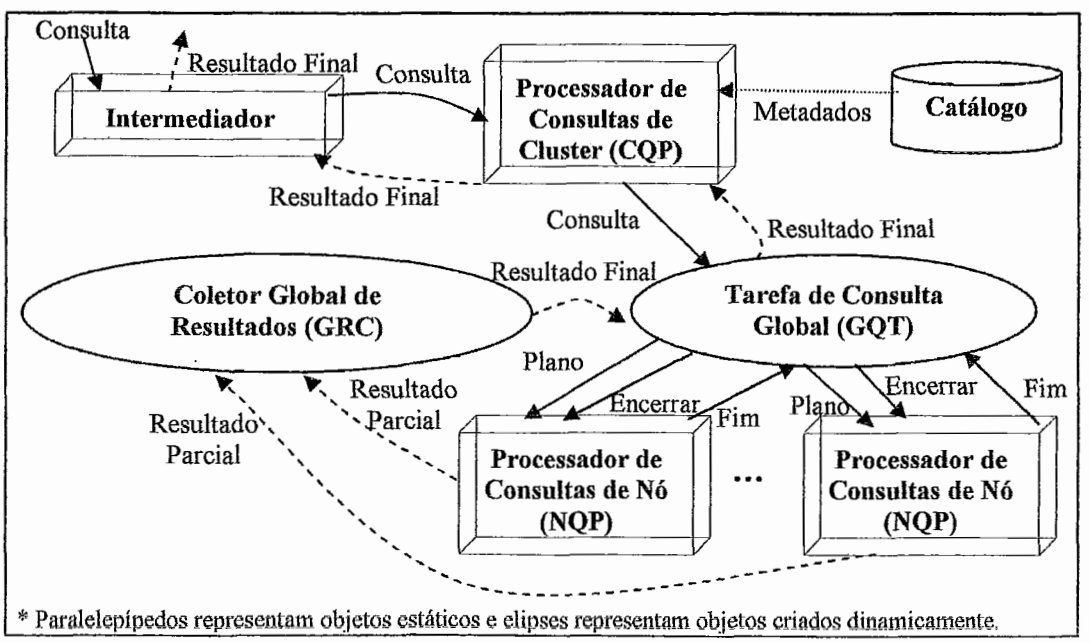

Figura 14 - Etapas globais do processamento de uma consulta

O processo se inicia quando o Intermediador recebe uma consulta de uma aplicação cliente. Ele a repassa ao CQP. Para cada consulta recebida, o CQP cria uma nova Tarefa de Consulta Global (GQT – *Global Query Task*), que é executada em uma linha de execução (*thread*) em separado. Fisicamente, a GQT pode ser criada em qualquer nó do cluster. O CQP a cria no nó que estiver executando menos tarefas, reduzindo assim problemas de sobrecarga dos nós. A função da GQT é coordenar a execução da consulta. Ela é responsável pela fase de Fragmentação Virtual Inicial. A escolha dos atributos de fragmentação e o cálculo dos limites dos intervalos dos fragmentos iniciais são feitos a partir dos metadados presentes no Catálogo. Além da fragmentação inicial, é atribuição da GQT executar a fase de Encerramento do processamento da consulta.

As sub-consultas executadas pela AVP produzem resultados parciais da consulta submetida pelo usuário. A produção do resultado final exige um trabalho de composição global desses resultados parciais, principalmente em casos de agregação de valores. Para executar essa tarefa, a GQT cria um Coletor Global de Resultados (GRC – *Global Result Collector*), também em uma linha de execução à parte.

Após a fragmentação virtual inicial e a criação do GRC, a GQT envia aos NQPs dos nós envolvidos no processamento da consulta os seus Planos de Execução. Cada plano é constituído por uma nova versão da consulta inicial (que inclui os predicados para fragmentação virtual), pelos limites do fragmento virtual inicial e por uma referência para o GRC.

Os NQPs são os responsáveis pela execução das sub-consultas através das fases de ajuste do tamanho do fragmento virtual e redistribuição de carga. Seu funcionamento é descrito na seção IV.3. Durante essas etapas, resultados parciais são enviados ao GRC. Ao encerrar a execução das sub-consultas relativas ao seu fragmento inicial, o NQP notifica a GQT. Ao receber notificações de todos os NQPs (fase de Encerramento), a GQT solicita ao GRC o resultado final da consulta. Ao terminar de produzir esse resultado, o GRC o envia à GQT, que o repassa ao CQP. O CQP repassa o resultado ao Intermediador que, por fim, o envia à aplicação cliente.

IV.3. Processamento Local

Nessa seção, descrevemos as etapas executadas em cada nó para o processamento de uma consulta. Os componentes envolvidos estão ilustrados na Figura 15. Os componentes estáticos presentes em cada nó são o NQP, o Intermediador do SGBD (DBG – *DBMS Gateway*) e o SGBD. Para simplificar a figura, omitimos o banco de dados.

O DBG possui duas funções. Primeiramente, ele provê um conjunto de conexões ao SGBD, reduzindo o tempo de início das consultas. Além disso, torna os demais componentes independentes do SGBD, que não precisam se conectar diretamente a ele.

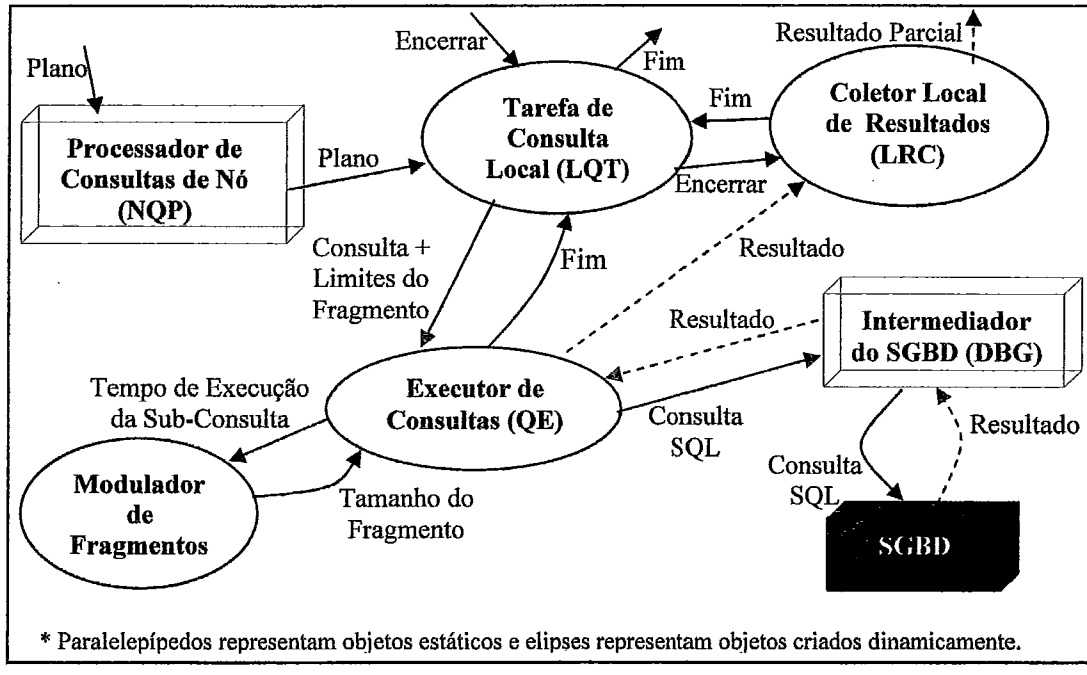

Figura 15 - Etapas locais do processamento de uma consulta

Ao receber um plano de execução, o NQP cria uma Tarefa de Consulta Local (LQT – *Local Query Task*), que gerencia o processamento da consulta em nível local. Cada consulta é gerenciada pela sua própria LQT, que é executada em uma linha de execução exclusiva. A LQT recebe o seu plano de execução e cria os seguintes objetos: Executor de Consultas (QE – *Query Executor*) e o Coletor Local de Resultados (LRC – *Local Result Collector*), cada um em uma nova linha de execução. A utilização de linhas de execução diferentes para alguns objetos tem o objetivo de aproveitar ao máximo os recursos de concorrência do sistema operacional. Muitos clusters possuem

nós com mais de um processador, o que permite a obtenção de paralelismo na execução de tarefas.

O QE é o objeto responsável pela submissão das sub-consultas. Ele cria um objeto denominado Modulador de Fragmentos. A interação entre o QE e o Modulador implementa o algoritmo de fragmentação virtual adaptativa. Depois de criar o Modulador, o QE solicita um tamanho de intervalo para utilizar na primeira sub-consulta. Ele a submete então ao DBG, que a repassa ao SGBD. O resultado produzido é devolvido ao DBG, que o repassa ao QE. O QE envia então ao Modulador o tempo gasto na execução da sub-consulta com o tamanho de fragmento utilizado e solicita um novo tamanho. O Modulador o calcula de acordo com a AVP e o informa ao QE, que o utiliza em uma nova sub-consulta. O processo continua até que o fragmento inicial tenha sido completamente processado.

O resultado de cada sub-consulta é enviado pelo QE ao LRC. O LRC desempenha o mesmo papel do GRC, porém em um nível local. Ele recebe os resultados das sub-consultas para a produção do resultado parcial do nó. Nosso objetivo com a criação do LRC é evitar que o GRC se torne um ponto de contenção na arquitetura, o que poderia ocorrer, caso todos os nós enviassem os resultados de todas as suas sub-consultas diretamente a ele. Assim, tornamos o processo de composição de resultados distribuído.

Ao terminar de processar todo o intervalo inicial, o QE notifica a LQT, que, por sua vez, notifica a GQT. O QE entra então em estado de espera por um novo intervalo. Isso se dá devido ao algoritmo de redistribuição dinâmica de carga. São os LQTs dos diferentes nós que interagem para a implementação desse algoritmo. São eles os responsáveis pelo envio das mensagens de oferta e aceitação de ajuda, pela propagação das ofertas e pela notificação de que uma extremidade do cluster foi atingida (favor ver algoritmo de propagação na seção III.4.1.3). Quando notificada pelo QE de que o fragmento inicial foi totalmente processado, a LQT inicia a propagação de sua oferta de ajuda. Se algum nó aceitar a oferta, ela solicita um intervalo ao nó e o repassa ao QE, que reinicia o Modulador e recomeça a submissão de sub-consultas. Enquanto o QE executa consultas, a LQT tem como responsabilidade propagar ofertas de outros nós e aceitar ofertas de ajuda, caso haja fragmentos a processar.

Quando todos os fragmentos iniciais foram processados, a GQT envia mensagens a todas as LQTs ordenando o encerramento de suas atividades. Cada LQT, ao receber essa mensagem, ordena que o LRC envie seu resultado parcial ao GRC. Esse envio, na

verdade, pode ocorrer de tempos em tempos por iniciativa do próprio LRC, de acordo com o tamanho do resultado, a fim de reduzir o tempo de produção do resultado final. Ao enviar o resultado, o LRC notifica a LQT, que, por sua vez, notifica a GQT. O processo é então encerrado.

V. Validação Experimental

Para validar nossa solução, implementamos o protótipo de um processador de consultas em um cluster de PCs e executamos experimentos com o *benchmark* TPC-H. Na seção V.1, descrevemos brevemente nossa implementação e o ambiente de testes. As consultas do TPC-H utilizadas, as estratégias para fragmentação virtual empregadas e a técnica que usamos para simular distorção com os dados do TPC-H são apresentadas na seção V.2.

Nas seções seguintes, descrevemos os resultados dos experimentos realizados. Nessas seções, representamos a fragmentação virtual simples pela sigla SVP; a fragmentação virtual adaptativa sem redistribuição dinâmica de carga pela sigla AVP; e a fragmentação virtual adaptativa com a redistribuição pela sigla AVP_WR (WR – *Workload Redistribution*). Realizamos basicamente duas séries de experimentos. Na primeira, testamos e comparamos os desempenhos da SVP, da AVP e da AVP_WR durante a execução de consultas isoladas, ou seja, com apenas uma consulta sendo executada no cluster. Os resultados desses experimentos estão descritos na seção V.3. Na segunda série de experimentos, avaliamos o desempenho das técnicas propostas durante a execução de lotes de consultas concorrentes. Seus resultados estão descritos na seção V.4.

V.1. Ambiente de Testes e Detalhes de Implementação

Implementamos nosso protótipo em um cluster de PCs (arquitetura de memória distribuída) com 64 nós, cada um apresentando dois processadores Intel Xeon com 2,4 GHz, 1 Gb de memória RAM, e 20 Gb de disco rígido. Os nós se encontram interconectados por uma rede Ethernet 100. Utilizamos o SGBD PostgreSQL v. 7.3.4 e sistema operacional Linux. Utilizamos dados e consultas do *benchmark* TPC-H, que representa genericamente aplicações de suporte à decisão e que "consiste em um conjunto de consultas *ad-hoc* orientadas a negócios" (TPC, 2003a). Geramos a base de dados TPC-H conforme especificado por TPC (2003a), com fator de escala igual a 5, produzindo um banco de dados de aproximadamente 11 Gb. Replicamos esse banco de dados em todos os nós do cluster. As tabelas de fatos foram fisicamente ordenadas de acordo com os atributos de fragmentação e índices foram construídos com base nos

mesmos. Construímos ainda índices para todas as chaves estrangeiras das tabelas de fatos e de todas as tabelas de dimensão. Após a geração dos dados e a criação dos índices, as estatísticas do banco de dados foram atualizadas a fim de que o otimizador de consultas do SGBD as utilizasse. Como nosso objetivo é lidar com consultas *ad-hoc*, nenhuma outra otimização foi feita, como orienta o *benchmark*.

Durante os experimentos, as consultas foram executadas utilizando-se diferentes números de nós. Para cada configuração, cada consulta foi executada três vezes, a fim de minimizar a influência de flutuações de carga do próprio sistema operacional. Tomamos então o tempo médio de execução. Nos gráficos onde mostramos esses tempos, nós os normalizamos dividindo cada tempo médio de execução pelo maior tempo de execução da consulta correspondente.

Nosso protótipo foi implementado em Java. Alguns componentes são objetos Java RMI: CQP, NQP, GQT, LQT, GRC, LRC e DBG. Nossa implementação utiliza *multi-threading* a fim de aproveitar ao máximo as características de multitarefa do sistema operacional. A composição final de resultados é feita em um único nó para cada consulta.

A implementação do algoritmo da AVP demanda a utilização de uma série de parâmetros, como o tamanho inicial utilizado para os intervalos que determinam os fragmentos virtuaispor exemplo. Os parâmetros utilizados na nossa implementação e seus valores estão descritos na Tabela 1.

Tabela 1 - Parâmetros de Execução da AVP

Parâmetro	Valor	Descrição
tam_frag_ini	1024	Tamanho inicial de intervalo para fragmentos virtuais.
tx_ini_aum_tam	100 %	Taxa inicial para aumento no tamanho do intervalo que define um fragmento virtual.
tx_red_tam	5 %	Taxa utilizada para redução do tamanho do fragmento quando deterioração de desempenho é detectada.
tx_reini_aum_tam	20 %	Taxa de aumento no tamanho do fragmento utilizada após detecção de deterioração.
tol_aum_t_exec	25 %	Tolerância do aumento do tempo de execução durante modulação dos tamanhos dos fragmentos. Se o tamanho do fragmento é aumentado por um fator t_i, um aumento de (tol_aum_tempo_exec * t_i) no tempo de execução é tolerado.
tol_aum_t_deter	10 %	Tolerância do aumento do tempo de execução para suspeita de deterioração de desempenho. Se, após a determinação de um tamanho de fragmento ideal, o tempo de execução sofrer um acréscimo superior a tol_aum_tempo_deter, há suspeita de deterioração do desempenho.
num_exec_deter	3	Número de execuções consecutivas com tempo superior ao determinado por tol_aum_tempo_deter para que seja concluído

Parâmetro	Valor	Descrição
		que há deterioração de desempenho. Se num_exec_deter execuções consecutivas apresentarem tempo acima do tolerado, o processo de busca por um novo tamanho de fragmento é iniciado.

V.2. Consultas do Benchmark TPC-H

Para facilitar o entendimento das consultas utilizadas, descrevemos brevemente a estrutura da base de dados do *benchmark* TPC-H. Oito tabelas são especificadas: seis tabelas de dimensão (Region, Nation, Supplier, Part, Customer e Partsupp) e duas tabelas de fatos (Orders e Lineitem). Region e Nation são bastante pequenas, possuindo cinco e vinte e cinco tuplas, respectivamente. Esses números são invariáveis, independem do fator de escala utilizado para a geração dos dados. Chamando *SF* o fator de escala, as cardinalidades das outras tabelas são: : |Supplier| = *SF**10.000; |Customer| = *SF**150.000; |Part| = *SF**200.000; |Partsupp| = *SF**800.000; |Orders| = *SF**1.500.000; e |Lineitem| = *SF**6.000.000. Lineitem é a maior tabela em número de tuplas, seguida por Orders, com 25% da sua cardinalidade. As tuplas de Lineitem referenciam as tuplas de Orders através de uma chave estrangeira que é também o primeiro atributo da sua chave primária. A Figura 16 traz um esquema de dados com as tabelas do TPC-H, seus atributos e relacionamentos.

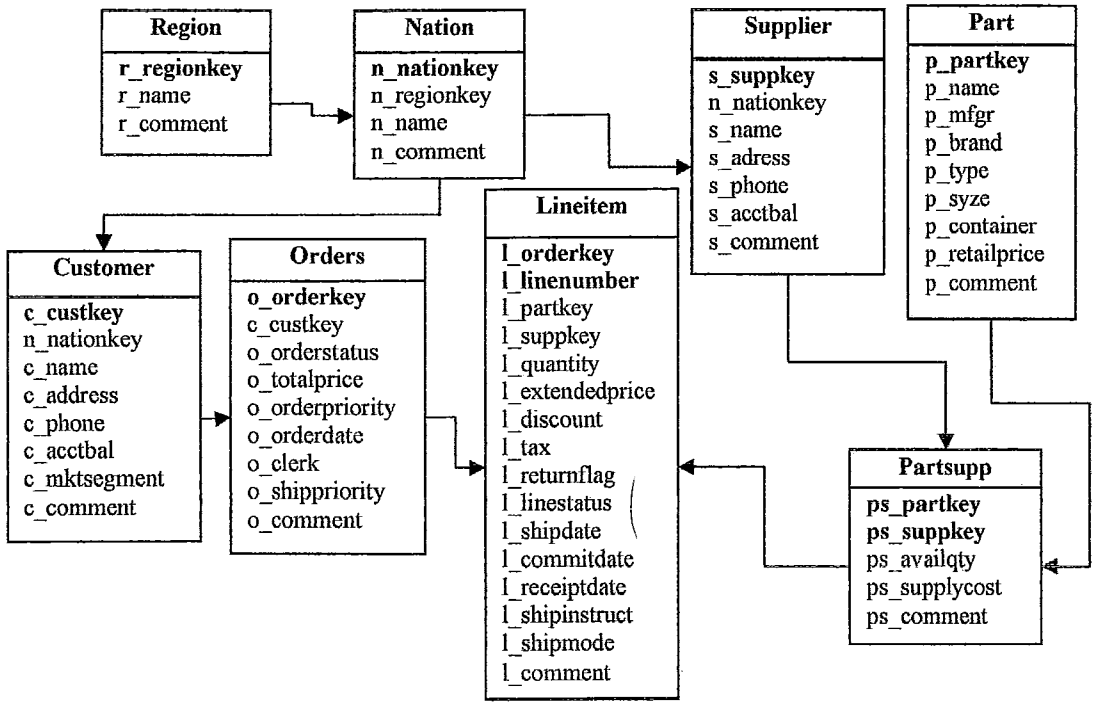

Figura 16 - Esquema do benchmark TPC-H (TPC, 2003a)

Utilizamos as consultas do TPC-H Q1, Q4, Q5, Q6, Q12, Q14, Q18 e Q21, que possuem diferentes complexidades. Elas se encontram listadas abaixo e foram geradas com auxílio do aplicativo *qgen*, encontrado no sítio da organização TPC (TPC, 2004).

Consulta Q1:

```
select l_returnflag, l_linestatus,
       sum(l_quantity) as sum_qty, sum(l_extendedprice) as sum_base_price,
       sum(l_extendedprice * (1-l_discount)) as sum_disc_price,
       sum(l_extendedprice * (1-l_discount) * (1+l_tax)) as sum_charge,
       avg(l_quantity) as avg_qty,
       avg(l_extendedprice) as avg_price,
       avg(l_discount) as avg_disc,
       count(*) as count_order
from   lineitem
where  l_shipdate <= date '1998-12-01' - interval '90 day'
group by l_returnflag, l_linestatus
order by l_returnflag, l_linestatus;
```

Consulta Q4:

```
select o_orderpriority, count(*) as order_count
from   orders
where  o_orderdate >= date '1993-07-01'
  and  o_orderdate < date '1993-07-01' + interval '3 month'
  and  exists (select * from lineitem
               where  l_orderkey = o_orderkey
                 and  l_commitdate < l_receiptdate )
group by o_orderpriority
order by o_orderpriority;
```

Consulta Q5:

```
select n_name, sum(l_extendedprice * (1 - l_discount)) as revenue
from   customer, orders, lineitem, supplier, nation, region
where  c_custkey = o_custkey
  and  l_orderkey = o_orderkey
  and  l_suppkey = s_suppkey
  and  c_nationkey = s_nationkey
  and  s_nationkey = n_nationkey
  and  n_regionkey = r_regionkey
  and  r_name = 'ASIA' and o_orderdate >= date '1994-01-01'
  and  o_orderdate < date '1994-01-01' + interval '1 year'
group by n_name
order by n_name;
```

Consulta Q6:

```
select sum(l_extendedprice * l_discount) as revenue
from   lineitem
where  l_shipdate >= date '1994-01-01'
  and  l_shipdate < date '1994-01-01' + interval '1 year'
  and  l_discount between .06 - 0.01 and .06 + 0.01
  and  l_quantity < 24;
```

Consulta Q12:

```
select l_shipmode,
       sum( case when o_orderpriority = '1-URGENT' or
                      o_orderpriority = '2-HIGH'
                 then 1 else 0
            end ) as high_line_count,
       sum( case when o_orderpriority <> '1-URGENT' and
                      o_orderpriority <> '2-HIGH'
                 then 1 else 0
            end ) as low_line_count
from   orders, lineitem
where  o_orderkey = l_orderkey
  and  l_shipmode in ('MAIL', 'SHIP')
  and  l_commitdate < l_receiptdate
  and  l_shipdate < l_commitdate
  and  l_receiptdate >= date '1994-01-01'
  and  l_receiptdate < date '1994-01-01' + interval '1 year'
group by l_shipmode
order by l_shipmode;
```

Consulta Q14:

```
select 100.00 * sum(case when p_type like 'PROMO%'
                         then l_extendedprice * (1 - l_discount)
                         else 0
        end) / sum(l_extendedprice * (1-l_discount)) as promo_revenue
from   lineitem, part
where  l_partkey = p_partkey
  and  l_shipdate >= date '1995-09-01'
  and  l_shipdate < date '1995-09-01' + interval '1 month';
```

Consulta Q18:

```
select c_name, c_custkey, o_orderkey, o_orderdate, o_totalprice,
       sum(l_quantity)
from   customer, orders, lineitem
where  o_orderkey in (select l_orderkey from lineitem
                      group by l_orderkey
                      having sum(l_quantity) > 300 )
and c_custkey = o_custkey and o_orderkey = l_orderkey
group by c_name, c_custkey, o_orderkey, o_orderdate, o_totalprice
order by o_totalprice desc, o_orderdate
limit 100;
```

Consulta Q21:

```
select s_name, count(*) as numwait
from   supplier, lineitem l1, orders, nation
where  s_suppkey = l1.l_suppkey and o_orderkey = l1.l_orderkey
  and  o_orderstatus = 'F' and l1.l_receiptdate > l1.l_commitdate
  and  exists (select * from lineitem l2
               where  l2.l_orderkey = l1.l_orderkey
                 and  l2.l_suppkey <> l1.l_suppkey )
  and  not exists ( select * from  lineitem l3
                    where  l3.l_orderkey = l1.l_orderkey
                      and  l3.l_suppkey <> l1.l_suppkey
                      and  l3.l_receiptdate > l3.l_commitdate )
  and s_nationkey = n_nationkey
  and n_name = 'SAUDI ARABIA'
group by s_name
order by unmwait desc, s_name
limit 100;
```

A seguir, descrevemos brevemente cada uma das consultas. Q1 acessa apenas a tabela Lineitem e executa várias operações de agregação. O predicado definido em Q1 não é muito seletivo, o que a faz utilizar 98% das tuplas da tabela. Q4 realiza uma agregação sobre a tabela de fatos Orders. Além disso, o predicado definido em sua cláusula *where* inclui uma sub-consulta sobre a tabela Lineitem. Nessa sub-consulta, porém, são utilizadas apenas tuplas relacionadas à tupla de Orders sendo processada na consulta externa, como em uma junção. Q5 executa uma junção entre quatro tabelas de dimensão (Region, Nation, Supplier e Customer) e as duas tabelas de fatos, executando apenas uma função de agregação. Q6, assim como Q1, acessa apenas a tabela Lineitem.

As principais diferenças entre elas estão no fato de que Q6 executa apenas uma operação de agregação e possui um predicado mais seletivo, satisfeito por apenas 1,9% das tuplas da tabela. Q12 acessa ambas as tabelas de fatos, executando uma junção entre elas. Q14 executa uma junção entre uma tabela de fatos (Lineitem) e uma dimensão (Part). Q18 executa uma junção entre uma dimensão (Customer) e ambas as tabelas de fatos. Sua principal característica é a presença de uma sub-consulta sobre a relação Lineitem. Essa sub-consulta executa uma agregação e não possui predicado, o que a faz acessar todas as tuplas de Lineitem. Q21 executa junções entre as duas tabelas de fatos e duas dimensões (Supplier e Nation). Além disso, possui duas sub-consultas sobre a tabela Lineitem. Tanto Q18 quanto Q21 limitam o tamanho do resultado em cem tuplas. Essa cláusula não foi utilizada em nossos experimentos, pois o Coletor Global e o Coletor Local de Resultados não possuem essa funcionalidade implementada. Consideramos as consultas escolhidas bastante representativas para aplicações OLAP. Elas possuem diferentes complexidades e alto custo de processamento.

A estratégia para fragmentação virtual utilizada é similar à encontrada no trabalho de AKAL *et al.* (2002). Para as consultas Q1 e Q6, ela é baseada no atributo l_orderkey, uma vez que esse é o primeiro atributo da chave primária da tabela Lineitem e possui poucas tuplas associadas a cada um de seus valores. Para as consultas Q5, Q12 e Q18, a fragmentação virtual da tabela Orders é baseada na chave primária o_orderkey e a da tabela Lineitem, na chave estrangeira l_orderkey. Para Q4, Q18 e Q21 acrescentamos um predicado para fragmentação virtual também nas sub-consultas, uma vez que isso não afetaria o seu resultado e traria ganho de desempenho. Para Q14, utilizamos a mesma estratégia de Q1 e Q6, deixando a tabela Part não fragmentada.

Não implementamos em nosso processador de consultas as funcionalidades de ordenação de tuplas e limitação da cardinalidade do resultado. Por esse motivo, predicados com esse objetivo presentes nas versões originais das consultas foram removidos das versões para fragmentação virtual. Além disso, certas operações de agregação precisam ser desmembradas nessas versões. A função para cálculo de médias (*avg*()) da SQL, por exemplo, precisa ser reescrita como um somatório (*sum*()) e uma operação de contagem de valores (*count*()). A seguir, listamos as versões das consultas para fragmentação virtual, com as modificações em relação à versão original destacadas em negrito. Os parâmetros :Ii e :If correspondem ao início e ao término dos intervalos que definem os fragmentos virtuais.

Consulta Q1:

```
select l_returnflag, l_linestatus,
       sum(l_quantity) as sum_qty, sum(l_extendedprice) as sum_base_price,
       sum(l_extendedprice * (1-l_discount)) as sum_disc_price,
       sum(l_extendedprice * (1-l_discount) * (1+l_tax)) as sum_charge,
       sum(l_quantity) as avg_qty_1, count(l_quantity) as avg_qty_2,
       sum(l_extendedprice) as price_1, count(l_extendedprice) as price_2,
       sum(l_discount) as avg_disc_1, count(l_discount) as avg_disc_1,
       count(*) as count_order
from   lineitem
where  l_shipdate <= date '1998-12-01' - interval '90 day'
  and  l_orderkey >= :Ii and l_orderkey < :If
group by l_returnflag, l_linestatus;
```

Consulta Q4:

```
select o_orderpriority, count(*) as order_count
from   orders
where  o_orderdate >= date '1993-07-01'
  and  o_orderdate < date '1993-07-01' + interval '3 month'
  and  o_orderkey >= :Ii and o_orderkey < :If
  and  exists (select * from lineitem
               where  l_orderkey = o_orderkey
                 and  l_commitdate < l_receiptdate
                 and  l_orderkey >= :Ii and l_orderkey < :If )
group by o_orderpriority;
```

Consulta Q5:

```
select n_name, sum(l_extendedprice * (1 - l_discount)) as revenue
from   customer, orders, lineitem, supplier, nation, region
where  c_custkey = o_custkey
  and  l_orderkey = o_orderkey
  and  l_suppkey = s_suppkey
  and  c_nationkey = s_nationkey
  and  s_nationkey = n_nationkey
  and  n_regionkey = r_regionkey
  and  r_name = 'ASIA'
  and  o_orderdate >= date '1994-01-01'
  and  o_orderdate < date '1994-01-01' + interval '1 year'
  and o_orderkey >= :Ii and o_orderkey < :If
  and l_orderkey >= :Ii and l_orderkey < :If
group by n_name;
```

Consulta Q6:

```
select sum(l_extendedprice * l_discount) as revenue
from   lineitem
where  l_shipdate >= date '1994-01-01'
  and  l_shipdate < date '1994-01-01' + interval '1 year'
  and  l_discount between .06 - 0.01 and .06 + 0.01
  and  l_quantity < 24
  and  l_orderkey >= :Ii and l_orderkey < :If;
```

Consulta Q12:

```
select l_shipmode,
       sum( case when o_orderpriority = '1-URGENT' or
                      o_orderpriority = '2-HIGH'
                 then 1 else 0
            end ) as high_line_count,
       sum( case when o_orderpriority <> '1-URGENT' and
                      o_orderpriority <> '2-HIGH'
                 then 1 else 0
            end ) as low_line_count
from   orders, lineitem
where  o_orderkey = l_orderkey
  and  l_shipmode in ('MAIL', 'SHIP')
  and  l_commitdate < l_receiptdate
  and  l_shipdate < l_commitdate
  and  l_receiptdate >= date '1994-01-01'
  and  l_receiptdate < date '1994-01-01' + interval '1 year'
  and  o_orderkey >= :Ii and o_orderkey < :If;
  and  l_orderkey >= :Ii and l_orderkey < :If;
group by l_shipmode;
```

Consulta Q14:

```
select 100.00 * sum(case when p_type like 'PROMO%'
                         then l_extendedprice * (1 - l_discount)
                         else 0
                    end) as promo_revenue_1,
        sum(l_extendedprice * (1-l_discount)) as promo_revenue_2
from   lineitem, part
where  l_partkey = p_partkey  and  l_shipdate >= date '1995-09-01'
  and  l_shipdate < date '1995-09-01' + interval '1 month'
  and  l_orderkey >= :Ii and l_orderkey < :If;
```

Consulta Q18:

```
select c_name, c_custkey, o_orderkey, o_orderdate, o_totalprice,
       sum(l_quantity)
from   customer, orders, lineitem
where  o_orderkey in (select l_orderkey
                      from   lineitem
                      where  l_orderkey >= :Ii and l_orderkey < :If
                      group by l_orderkey
                      having sum(l_quantity) > 300 )
  and  c_custkey = o_custkey
  and  o_orderkey = l_orderkey
  and  o_orderkey >= :Ii and o_orderkey < :If
  and  l_orderkey >= :Ii and l_orderkey < :If
group by c_name, c_custkey, o_orderkey, o_orderdate, o_totalprice;
```

Consulta Q21:

```
select s_name, count(*) as numwait
from   supplier, lineitem l1, orders, nation
where  s_suppkey = l1.l_suppkey and o_orderkey = l1.l_orderkey
  and  o_orderstatus = 'F' and l1.l_receiptdate > l1.l_commitdate
  and  exists (select * from lineitem l2
               where  l2.l_orderkey = l1.l_orderkey
                 and  l2.l_suppkey <> l1.l_suppkey
                 and  l2.l_orderkey >= :Ii and l2.l_orderkey < :If )
  and  not exists ( select * from lineitem l3
                    where  l3.l_orderkey = l1.l_orderkey
                      and  l3.l_suppkey <> l1.l_suppkey
                      and  l3.l_receiptdate > l3.l_commitdate
                      and  l3.l_orderkey >= :Ii and l3.l_orderkey < :If )
  and s_nationkey = n_nationkey
  and n_name = 'SAUDI ARABIA'
  and  o_orderkey >= :Ii and o_orderkey < :If
  and  l1.l_orderkey >= :Ii and l1.l_orderkey < :If
group by s_name;
```

A sumarização das principais características das consultas com as tabelas acessadas estão na Tabela 2.

Tabela 2 - Principais características das consultas utilizadas

Consulta	Tabela(s) Acessada(s)	Número de Agregações	Agrupamento	Tabela(s) Fragmentada	Sub-consulta
Q1	Lineitem	8	Sim	Lineitem	Não
Q4	Lineitem Orders	1	Sim	Lineitem Orders	Sim (Lineitem)
Q5	Customer Lineitem Nation Orders Region Supplier	1	Sim	Lineitem Orders	Não
Q6	Lineitem	1	Não	Lineitem	Não
Q12	Lineitem Orders	2	Sim	Lineitem Orders	Não
Q14	Lineitem Part	2	Não	Lineitem	Não
Q18	Customer Lineitem Orders	1	Sim	Lineitem Orders	Sim (Lineitem)
Q21	Lineitem Orders Nation Supplier	1	Sim	Lineitem Orders	Sim (Lineitem – 2)

O banco de dados definido pelo TPC-H apresenta distribuição de valores quase uniforme para os atributos utilizados na fragmentação virtual. A fim de avaliar a eficiência da nossa técnica de redistribuição dinâmica de carga, necessitamos de distorção de dados. Devido a limitações de espaço para armazenamento em disco, não pudemos gerar dois bancos de dados diferentes com o mesmo tamanho. Por essa razão, simulamos uma distribuição de valores *zipf* (KNUTH, 1973) para os atributos de fragmentação utilizando diferentes tamanhos para os fragmentos na fase de fragmentação virtual inicial.

A distribuição de valores *zipf* é definida da seguinte maneira. Considere um intervalo de valores e divida-o em n intervalos distintos e não-sobrepostos. Denominando p_i a probabilidade de um determinado valor pertencer ao i-ésimo sub-intervalo, a distribuição de valores é considerada como sendo *zipf* se

$p_1 = c/1, p_2 = c/2, \ldots, p_n = c/n,$ **(3)**

onde $c = 1/H_n$, e H_n é o n-ésimo número harmônico de ordem 1, i.e., $H_n = 1 + 1/2 + 1/3 + \ldots + 1/n$.

Os atributos de fragmentação virtual utilizados em nossos experimentos (o_orderkey, para a tabela Orders, e l_orderkey, para a tabela Lineitem) possuem os mesmos intervalos de valores, variando de 1 a $3*10^7$. Para simular a distorção de dados, dividimos esse intervalo em 512 sub-intervalos e calculamos o tamanho de cada um fazendo $n = 512$ em (3). Em outras palavras, o tamanho do primeiro intervalo é p_1*3*10^7, o do segundo é p_2*3*10^7, e assim por diante até o qüingentésimo décimo segundo intervalo, com tamanho $p_{512}*3*10^7$. Obviamente, essa abordagem faz com que os maiores tamanhos sejam associados aos primeiros intervalos. Para simular uma distorção mais realista, redistribuímos aleatoriamente os tamanhos ao longo do intervalos. Finalmente, como executamos nossos testes em um cluster de 64 nós, agrupamos sub-intervalos consecutivos somando os seus tamanhos até obtermos 64 fragmentos virtuais. Dessa forma, o maior intervalo recebeu tamanho igual a 4.935.617, e o menor, 68.390. Os tamanhos resultantes foram utilizados na fase de fragmentação virtual inicial. Em experimentos que utilizassem menos de 64 nós, agrupamos intervalos consecutivos novamente, da mesma maneira que procedemos para obter os 64 intervalos iniciais.

A distribuição *zipf* causa extrema não-uniformidade na fragmentação virtual inicial. Nosso objetivo é testar a eficácia das técnicas propostas em condições totalmente opostas às proporcionadas pela distribuição uniforme. Situações mais favoráveis, ou seja, com distribuições intermediárias, podem ser obtidas através de variações no cálculo da distribuição *zipf* (KNUTH, 1973). No entanto, elas não foram avaliadas nesse trabalho.

V.3. Experimentos com Consultas Isoladas

Nesta série de experimentos, avaliamos a aceleração obtida no processamento de consultas isoladas de acordo com o número de nós utilizados no clusters. Primeiramente, analisamos os resultados dos experimentos com distribuição uniforme de dados (seção V.3.1). Em seguida, analisamos os resultados dos experimentos com distorção de dados simulada (seção V.3.2). Devido à grande disparidade no tempo de

execução das diferentes consultas, nossos gráficos mostram tempos normalizados. Eles foram obtidos da seguinte forma: para uma determinada consulta e um método de processamento (SVP, AVP ou AVP_WR), dividimos o seu tempo de execução pelo maior tempo por ela apresentado com o método considerado. Os tempos considerados são médias de várias execuções consecutivas, a fim de minimizar efeitos de flutuações momentâneas de carga do cluster devido a processos simultâneos inevitáveis, como os do sistema operacional. As análises foram baseadas nas várias medidas realizadas e nos planos de execução gerados pelo PostgreSQL.

V.3.1. Experimentos com Distribuição de Dados Uniforme

Faremos três tipos de análise. Primeiramente, na seção V.3.1.1, analisamos a aceleração obtida com o uso da AVP_WR. Na seção V.3.1.2, comparamos o desempenho da SVP e da AVP_WR. Por fim, analisamos o impacto do algoritmo de redistribuição dinâmica de carga na seção V.3.1.3.

V.3.1.1 Analisando a Aceleração com AVP_WR

Iniciamos analisando a aceleração obtida com a utilização da AVP_WR no cluster de banco de dados. A Tabela 3 mostra os tempos de execução obtidos em configurações do cluster com diferentes números de nós, variando de 1 a 64.

Tabela 3 - AVP_WR - tempos de execução de consultas com distribuição uniforme (em segundos)

Consulta	1 nó	2 nós	4 nós	8 nós	16 nós	32 nós	64 nós
Q1	802,77	375,72	194,64	95,25	46,56	23,03	11,99
Q4	358,77	182,39	92,23	53,48	4,74	2,91	2,07
Q5	360,72	184,22	95,08	26,28	14,10	8,06	4,88
Q6	200,35	102,15	50,59	16,02	7,98	4,44	2,66
Q12	361,57	183,27	88,48	37,57	17,23	8,86	5,05
Q14	387,60	193,23	106,54	78,98	8,55	4,79	2,94
Q18	300,03	143,05	73,88	37,59	13,87	7,30	4,12
Q21	532,07	262,64	135,45	66,64	37,60	18,76	9,79

A Figura 17 mostra um gráfico com os mesmos tempos de execução normalizados. Todas as consultas apresentaram redução no tempo de execução com o aumento do número de nós utilizados. As maiores acelerações obtidas foram para as consultas Q4 (redução de 91,14% ao passar de 8 para 16 nós) e Q14 (redução de 89,17% ao passar também de 8 para 16 nós).

Em configurações com 8 nós, as consultas Q1, Q5, Q6 e Q12 obtêm aceleração super-linear em relação à execução seqüencial. Com 16, 32 e 64 nós, todas as demais consultas, exceto a Q21, obtém aceleração super-linear em relação à execução seqüencial.

Com 2 e 4 nós torna-se difícil a obtenção de aceleração linear, apesar de ela ter sido conseguida em alguns casos. Diferente do que ocorre na SVP, os tamanhos dos fragmentos virtuais utilizados em cada sub-consulta da AVP_WR não depende do tamanho do intervalo inicial designado para cada nó. Assim, esperamos que intervalos iniciais maiores sejam processados por um número maior de sub-consultas. Quanto mais sub-consultas, maior o tempo total gasto pelo sistema com as fases do processamento de consultas que são independentes da quantidade de dados acessados (geração do plano de execução, por exemplo). Isso justifica em parte a aceleração sub-linear obtida com 2 e 4 nós. O outro motivo está ligado às operações de entrada e saída.

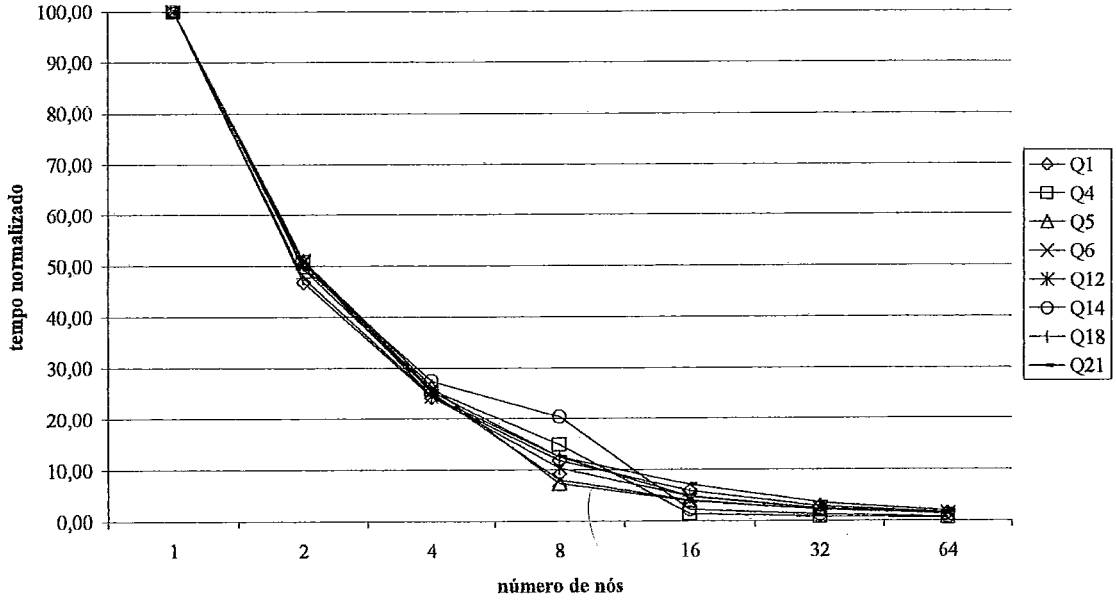

Figura 17 - Tempos de execução normalizados obtidos com AVP_WR

A Figura 18 mostra o número médio de operações de entrada e saída executadas por cada nó do cluster com a AVP_WR. O gráfico se encontra normalizado. Podemos notar que a queda é bastante acentuada conforme o número de nós aumenta. Com 8 nós, o número médio de operações é em torno de 10% do inicial. Exceto para as consultas Q1, Q5 e Q6, para as quais o número cai a menos de 0,1% do inicial, explicando sua aceleração super-linear. A partir de 16 nós, todas as consultas são processadas com

menos de 0,1% do número de operações de entrada e saída iniciais. Isso mostra que a AVP_WR resolve o problema de grandes fragmentos virtuais da SVP, sendo eficiente em manter os fragmentos com pequenos tamanhos apesar de ser uma técnica totalmente baseada em medidas de tempo obtidas no momento da execução da consulta, sem nenhum pré-processamento. Os planos de execução gerados pelo PostgreSQL mostram que todas as sub-consultas por ela geradas são processadas com a utilização do índice como meio de acesso à relação virtualmente fragmentada.

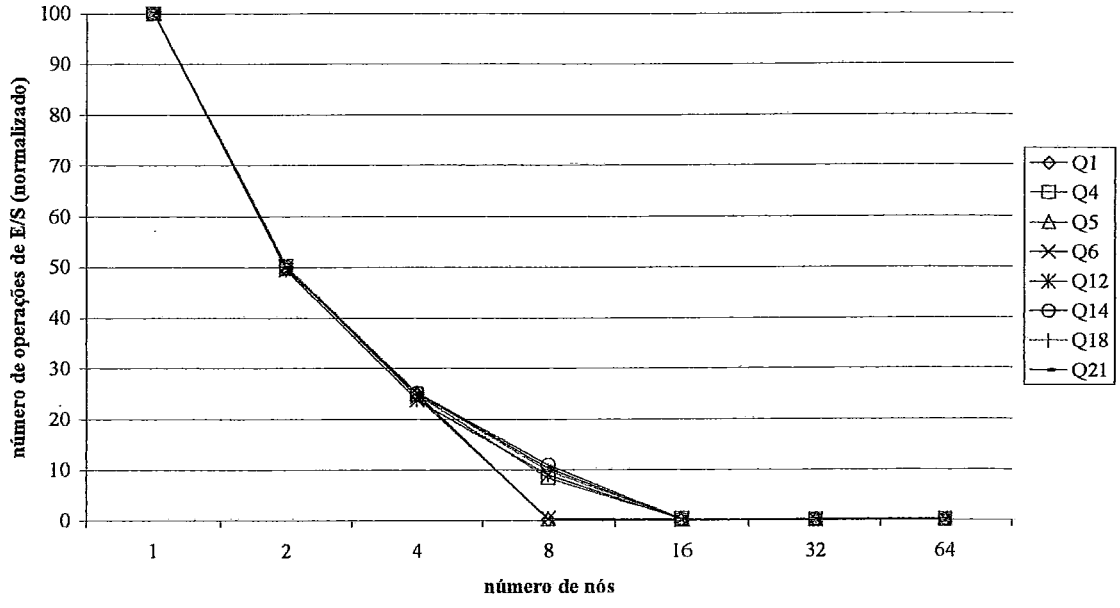

Figura 18 - Número de operações de entrada e saída com AVP_WR

Para a consulta Q21, só foi obtida aceleração superlinear na configuração com 2 nós. Nas demais, o ganho foi sublinear. Com 4 e 8 nós a aceleração foi muito próxima da linear, ficando, respectivamente, 1,83% e 0,2% abaixo dessa. Com 16, 32 e 64 nós, suas acelerações foram 13,05%, 12,84% e 17,72% abaixo da linear. Atribuímos esse fato à complexidade da consulta, que inclui duas sub-consultas. A maior complexidade ocasiona uma maior demanda de tempo para a geração do plano de execução. O elevado número de sub-consultas faz com que essa característica afete o desempenho final.

V.3.1.2 Comparação de Desempenho entre SVP e AVP_WR

Comparamos agora os desempenhos da SVP e da AVP_WR. A Tabela 4 mostra os tempos de execução das consultas com a utilização da SVP, em configurações com 1 a 64 nós.

Tabela 4 - SVP - Tempos de execução de consultas com distribuição uniforme (em segundos)

Consulta	1 nó	2 nós	4 nós	8 nós	16 nós	32 nós	64 nós
Q1	3.302,17	1.254,25	459,40	275,42	212,61	171,67	158,08
Q4	342,64	182,18	97,88	67,30	2,23	1,36	1,16
Q5	654,26	695,07	222,42	24,77	6,30	4,60	3,68
Q6	116,55	116,79	117,20	145,33	154,45	142,77	2,03
Q12	160,88	142,54	130,61	147,28	147,56	6,76	3,76
Q14	153,51	159,28	158,73	170,02	153,26	148,98	148,72
Q18	10.935,93	709,95	413,18	329,50	327,41	155,48	123,64
Q21	489,75	247,15	127,58	65,72	28,58	15,16	8,18

Para facilitar a comparação, utilizaremos os dados da Tabela 5 e o gráfico da Figura 19. Eles mostram o resultado da divisão do tempo de execução obtido com SVP pelo obtido com AVP_WR para cada consulta em cada uma das configurações utilizadas. Resultados acima de 1,0 representam melhor desempenho da AVP_WR. Abaixo de 1,0, ao contrário, representam melhor desempenho da SVP.

Tabela 5 - SVP / AVP_WR com distribuição de dados uniforme

Consulta	1 nó	2 nós	4 nós	8 nós	16 nós	32 nós	64 nós
Q1	4,11	3,34	2,36	2,89	4,57	7,45	13,19
Q4	0,96	1,00	1,06	1,26	0,47	0,47	0,56
Q5	1,81	3,77	2,34	0,94	0,45	0,57	0,75
Q6	0,58	1,14	2,32	9,07	19,35	32,18	0,76
Q12	0,44	0,78	1,48	3,92	8,57	0,76	0,74
Q14	0,40	0,82	1,49	2,15	17,92	31,12	50,65
Q18	36,45	4,96	5,59	8,77	23,61	21,29	30,04
Q21	0,92	0,94	0,94	0,99	0,76	0,81	0,84

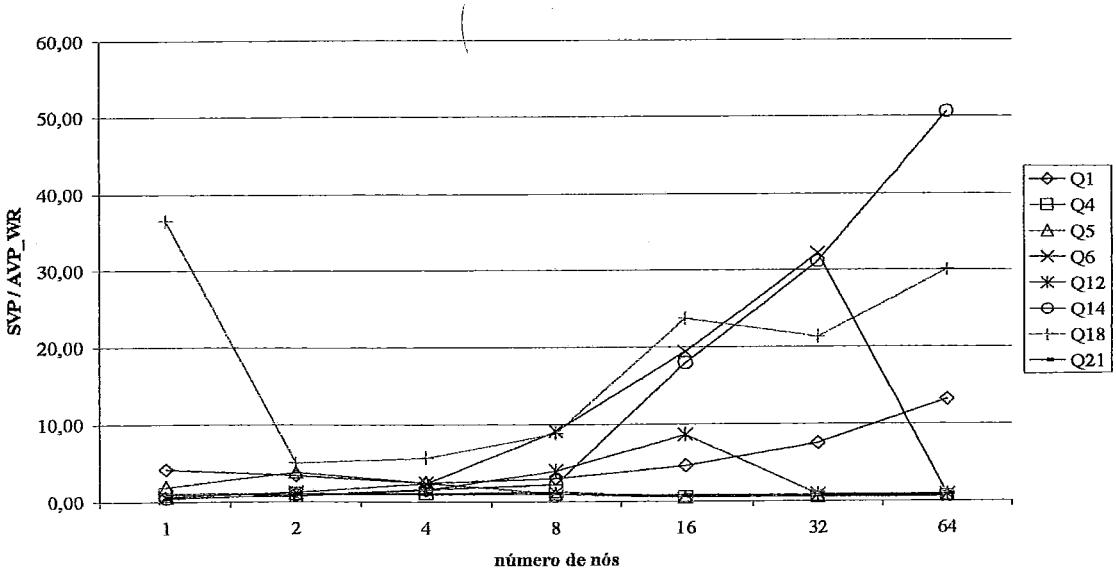

Figura 19 - Comparação dos tempos de execução obtidos com SVP e AVP_WR

Faremos uma análise consulta a consulta. Para a execução seqüencial de Q1, o plano de execução gerado pelo PostgreSQL determina uma busca linear sobre a tabela Lineitem. As tuplas recuperadas são então ordenadas e agrupadas. Como Q1 apresenta um predicado muito pouco seletivo, a operação de ordenação se torna muito dispendiosa. As sub-consultas geradas pela SVP para todos os casos (de 1 a 64 nós), são processadas através de buscas lineares, devido aos tamanhos dos fragmentos. A pouca seletividade do predicado da consulta contribui para uma redução no número de tuplas recuperadas proporcional à redução do fragmento, o que reduz o custo da operação de ordenação. Por isso, a SVP apresenta boa aceleração. No entanto, os planos gerados para as sub-consultas da AVP_WR nunca envolvem buscas lineares. O acesso à tabela Lineitem sempre é realizado com o auxílio do índice. O pequeno número de tuplas recuperadas por cada uma faz com que a ordenação seja sempre pouco dispendiosa. Daí, o seu melhor desempenho. Com 64 nós, AVP_WR consegue ser 13,19 vezes mais rápida do que a SVP para essa consulta. Vale destacar que, para essa consulta, AVP_WR apresenta melhor desempenho mesmo com apenas 1 nó. Q1 apresenta muitas operações de agregação, o que a faz consumir bastante tempo de CPU. Na SVP, todas as agregações de uma sub-consulta são realizadas pelo mesmo processo PostgreSQL, ou seja, não há paralelismo na sua execução, apesar de haver 2 processadores em cada nó. Na AVP_WR, ao contrário, há muitas sub-consultas mesmo com apenas um nó, cada uma lidando com poucos dados. Assim, o processo geral de agregação é dividido entre o PostgreSQL e o módulo da nossa arquitetura responsável pela composição local de resultados. Como há dois processadores em cada nó, essa operação é feita em paralelo. Daí o melhor desempenho da AVP_WR mesmo com 1 nó.

O plano de execução seqüencial gerado pelo PostgreSQL para a consulta Q4 estabelece busca linear sobre a tabela Orders e acesso à tabela Lineitem através de índice. Nessa consulta, o problema de fragmentos grandes não é significativo, uma vez que a tabela Orders possui apenas 25% do tamanho da tabela Lineitem. Os planos gerados para as sub-consultas da SVP também são eficientes. A partir de 2 nós, também a tabela Orders passa a ser acessada através do índice que possui sobre o atributo de fragmentação. De 2 a 8 nós, o número de operações de entrada e saída observados para o processamento com SVP contribui para que seu desempenho seja inferior ao da AVP_WR. Como Q1, ela também realiza ordenações. A partir de 16 nós, no entanto, o tempo de processamento dos fragmentos cai drasticamente. Ela então tem melhor

desempenho do que a AVP_WR. Essa, por sua vez, sofre com a complexidade e o número elevado de sub-consultas. Além disso, a redistribuição de carga naturalmente provoca um número maior de acessos ao disco devido aos fragmentos dinamicamente obtidos por cada nó e que possuem alta probabilidade de não estarem armazenados no *cache*. Com isso, apesar do número de operações desse tipo ser pequeno, é maior do que o apresentado pela SVP. Com 16, 32 e 64 nós, a SVP consegue, respectivamente, desempenhos 2,13, 2,14 e 1,78 vezes melhores dos que os obtidos com AVP_WR. Em termos absolutos, essas diferenças são de 2,5, 1,5 e 0,9 segundos, respectivamente.

Para a versão seqüencial da consulta Q5, o plano de execução gerado determina a busca linear como método de acesso a todas as tabelas. Esse é também o plano gerado para SVP com apenas 1 nó. Com 2 e 4 nós, a estratégia de fragmentação virtual da SVP passa a ter efeito e as tabelas virtualmente fragmentadas (Orders e Lineitem) são acessadas através de índices. Na verdade, constatamos que o predicado acrescentado para fragmentação virtual de Lineitem não tem efeito prático, uma vez que suas tuplas são recuperadas no momento da execução da sua junção com a tabela Orders, através do algoritmo de laço-aninhado indexado (GRAEFE, 1993). Todas as outras tabelas, no entanto, são acessadas via busca linear. Isso explica o melhor desempenho da AVP_WR em relação à SVP com 1, 2 e 4 nós. Para as sub-consultas da AVP_WR, todas as tabelas (exceto Nation e Region, de cardinalidades muito pequenas) são acessadas através de índices. A partir de 8 nós, a tabela Supplier começa a ser acessada através de índice para as sub-consultas da SVP. Com 16, 32 e 64 nós, todas as tabelas são acessadas dessa forma com a SVP, à exceção de Nation e Region. Isso provoca uma queda acentuada no número de operações de entrada e saída. Essas execuções sem buscas lineares associadas ao fato de que a SVP só executa uma sub-consulta em cada nó, explicam seu melhor desempenho em relação à AVP_WR. No pior caso, SVP consegue desempenho 2,24 vezes superior ao da AVP_WR. Com 1, 2 e 4 nós, no entanto, AVP_WR tem desempenho superior por fatores iguais a 1,81, 3,77 e 2,34, respectivamente.

De maneira similar à consulta Q1, Q6 acessa apenas a relação Lineitem. Ao contrário daquela, porém, esta possui um predicado muito mais seletivo, executa apenas uma função de agregação e não demanda ordenação, pois não possui cláusula de agrupamento de tuplas. Para a SVP, todos os planos de execução gerados para as configurações de 1 a 32 nós determinam a utilização de busca linear para o acesso à

tabela. Na AVP_WR, no entanto, todas as sub-consultas são executadas sem busca linear. Para 1 nó apenas, SVP apresenta melhor desempenho do que AVP_WR, pois toda a tabela deve ser varrida em ambos os casos. O maior número de sub-consultas da AVP_WR faz com que seu desempenho não seja tão bom. Nas configurações de 2 a 32 nós, no entanto, AVP_WR é melhor, chegando a obter tempo de execução 32,18 vezes menor do que o obtido com SVP. Porém, com 64 nós o plano de execução gerado para as sub-consultas da SVP não apresenta mais busca linear e a fragmentação virtual simples passa a ser eficiente. O número de operações de entrada e saída é substancialmente reduzido. O menor número de sub-consultas faz então a SVP apresentar desempenho 1,31 vezes melhor do que a AVP_WR.

A consulta Q12 utiliza apenas as tabelas de fatos. O plano de execução seqüencial determina a utilização do algoritmo junção por fusão (*merge-join*) (GRAEFE, 1993) entre ambas. Para isso, acessa Orders através de um índice (baseado no mesmo atributo usado posteriormente para fragmentação virtual) e executa uma busca linear sobre Lineitem. Esse é o mesmo plano gerado para as sub-consultas da SVP de 1 a 16 nós. Para a SVP com 32 e 64 nós e para a AVP_WR em todas as configurações, o plano de execução acessa a tabela Lineitem através do índice, que é baseado no atributo de fragmentação. Para 1 nó, SVP apresenta melhor desempenho. Como toda a tabela Lineitem deve ser percorrida, o maior número de sub-consultas da AVP_WR faz com que seu desempenho não seja tão bom. SVP também tem melhor desempenho com 2 nós mas, nesse caso, ele é apenas 29% superior. Com 4, 8 e 16 nós, a busca linear prejudica a SVP e AVP_WR obtém melhores desempenhos. Com 8 nós, seu desempenho é 3,92 vezes melhor e, com 16 nós, é 8,57 vezes melhor. Com 32 e 64 nós, a utilização do índice para sub-consultas da SVP faz com que o número de operações de entrada e saída decresça significativamente e ela supera a AVP_WR, sendo respectivamente 1,31 e 1,35 vezes mais rápida. Isso é explicado pelo maior número de sub-consultas da AVP_WR.

A consulta Q14 é aquela para a qual a AVP_WR apresenta melhor desempenho em relação à SVP. Ela executa uma junção entre a maior tabela de fatos (Lineitem) e a segunda maior dimensão (Part). O plano de execução seqüencial determina buscas lineares em ambas as tabelas e a utilização do algoritmo de *hash*-junção (GRAEFE, 1993). Para 1, 2, 4 e 8 nós, esse é o mesmo plano utilizado no processamento das sub-consultas da SVP. Para 16, 32 e 64 nós, a *hash*-junção é substituída pela junção por

fusão, a tabela Part passa a ser acessada através de índice e a busca linear sobre Lineitem é mantida. As sub-consultas da AVP_WR apresentam planos totalmente diferentes. Para elas, os planos gerados acessam ambas as tabelas através de índice e o algoritmo de junção por laços aninhados é utilizado. Com 1 nó, SVP é 2,52 vezes melhor do que AVP_WR. Com 2 nós, é apenas 1,21 vezes melhor. Atribuímos esse fato ao maior número de sub-consultas da AVP_WR. A partir daí, de 4 a 64 nós, AVP_WR é sempre melhor. Além da ausência de buscas lineares, a junção por laços aninhados, executada em memória, parece ser mais eficiente do que os algoritmos utilizados para SVP, que demandam operações extras de entrada e saída. Para 4 e 8 nós, o ganho é ainda modesto e a AVP_WR consegue superar a SVP por fatores iguais a 1,49 e 2,15 apenas. A partir de 16 nós, o ganho é mais acentuado, o que indica que o algoritmo de junção por fusão prejudica o desempenho da SVP, apesar de reduzir o número de operações de entrada e saída significativamente, de acordo com as observações realizadas. AVP_WR consegue então superar a SVP por fatores iguais a 17,92 (16 nós), 31,12 (32 nós) e 50,65 (64 nós).

A consulta Q18 é executada pelo PostgreSQL de maneira muito pouco eficiente. Observando a consulta, podemos notar que o resultado da sua sub-consulta é constante e não depende da tupla que está sendo processada na consulta externa. Seu resultado poderia ser obtido apenas uma vez, no início do processamento, e utilizado no restante da consulta. Não é o que acontece. O plano de execução mostra que a sub-consulta é re-executada uma vez para cada tupla da consulta externa. Esta, por sua vez, apresenta junções entre as tabelas Customer, Orders e Lineitem e nenhum predicado sobre as tuplas dessas relações. Com isso, a sub-consulta é re-executa para cada linha da tabela de fatos Orders. Além disso, o plano seqüencial determina buscas lineares sobre todas as tabelas, inclusive para a execução da sub-consulta. Para as sub-consultas da SVP, a tabela Lineitem é acessada através de busca linear nas configurações de 1 a 32 nós. Com 64 nós, o índice sobre o atributo de fragmentação é utilizado. Porém, na consulta externa, o algoritmo utilizado para a junção das duas tabelas é o de laço aninhado. Nas configurações anteriores, haviam sido usados o de *hash*-junção (1 a 4 nós) e o de junção por fusão (8 a 32 nós). A AVP_WR supera a SVP em todas as configurações por fatores que variam de 4,96 (2 nós) a 36,45 (1 nó), sendo igual a 30,04 com 64 nós. Suas sub-consultas não envolvem buscas lineares em nenhum caso. Vale observar que esse fator cai de 16 par 32 nós e volta a sofrer um aumento de 32 para 64 nós. Atribuímos esse

comportamento não a uma piora na aceleração da SVP mas à aceleração da AVP_WR, conseguida pelo baixo número de operações de entrada e saída com 64 nós.

A consulta Q21 é aquela para a qual a SVP apresenta melhor desempenho em relação à AVP_WR. Analisando os planos de execução gerados, verificamos que eles executam praticamente as mesmas operações, em ambos os casos. Nas sub-consultas, a tabela de fatos Lineitem sempre é acessada através de índices. Na consulta externa, a junção entre Lineitem e Orders é executada com o algoritmo de laço aninhado indexado. Com planos tão parecidos, SVP e AVP_WR obtêm acelerações muito semelhantes. O maior número de sub-consultas da AVP_WR faz com que a SVP obtenha melhor desempenho, principalmente pelo fato de essa ser uma consulta complexa, com duas sub-consultas, o que aumenta o tempo despendido na geração do plano de execução. Para todas as configurações, a SVP apresenta melhor desempenho, conseguindo ser 1,32 vezes melhor com 16 nós.

De acordo com o que foi apresentado, podemos concluir que a AVP_WR é uma técnica mais robusta do que a SVP. A SVP é bastante vulnerável às idiossincrasias do SGBD e apresenta grandes variações na sua aceleração. Quando seu desempenho é melhor do que o da AVP_WR, não o é por um fator muito elevado. Em configurações com mais de um nó, o maior fator foi igual a 2,24 (Q5, 16 nós), sendo que essa diferença em termos absolutos fica em 7,8s. Cabe ressaltar também que essa consulta não figura entre as mais pesadas do TPC-H .A AVP_WR, por sua vez, conseguiu superar a SVP por fatores bastante elevados, chegando a 50,65 para a consulta Q14 com 64 nós, sendo a diferença em termos absolutos por volta de 140s.

V.3.1.3 Comparação de Desempenho entre AVP e AVP_WR

Nessa seção, avaliamos o impacto da utilização do algoritmo de redistribuição dinâmica de carga em um ambiente em que ele não se faz muito necessário, uma vez que a distribuição dos dados do TPC-H é praticamente uniforme. Não é perfeitamente uniforme, vale destacar. Esperávamos que a utilização do algoritmo de redistribuição provocasse uma diminuição no desempenho do processador de consultas, uma vez que representa uma sobrecarga em relação à sua não utilização. Isso porém não ocorreu. Nos testes a seguir, AVP representa o algoritmo de fragmentação virtual adaptativa sem a redistribuição. Os tempos de execução obtidos com a AVP se encontram na Tabela 6.

Tabela 6 – AVP - Tempos de execução de consultas com distribuição uniforme (em segundos)

Consulta	1 nó	2 nós	4 nós	8 nós	16 nós	32 nós	64 nós
Q1	775,99	393,50	197,45	97,06	50,38	23,39	12,26
Q4	361,88	182,60	96,86	66,90	5,19	2,96	1,84
Q5	362,57	186,97	98,28	27,08	15,55	9,53	5,58
Q6	204,86	102,67	47,52	16,03	8,28	4,33	2,66
Q12	366,52	186,48	96,52	44,14	17,48	8,97	4,91
Q14	386,29	199,89	116,52	87,30	9,54	5,25	3,12
Q18	276,49	143,67	74,74	42,04	14,50	7,45	4,12
Q21	546,86	262,55	136,06	72,48	39,80	19,90	10,39

A comparação é feita com o auxílio dos dados da Tabela 7 e do gráfico da Figura 20, que mostram o resultado da divisão do tempo de execução obtido com AVP pelo obtido com AVP_WR para cada consulta em cada uma das configurações utilizadas.

Tabela 7 - AVP / AVP_WR com distribuição de dados uniforme

Consulta	1 nó	2 nós	4 nós	8 nós	16 nós	32 nós	64 nós
Q1	0,97	1,05	1,01	1,02	1,08	1,02	1,02
Q4	1,01	1,00	1,05	1,25	1,09	1,02	0,89
Q5	1,01	1,01	1,03	1,03	1,10	1,18	1,14
Q6	1,02	1,01	0,94	1,00	1,04	0,97	1,00
Q12	1,01	1,02	1,09	1,17	1,01	1,01	0,97
Q14	1,00	1,03	1,09	1,11	1,12	1,10	1,06
Q18	0,92	1,00	1,01	1,12	1,05	1,02	1,00
Q21	1,03	1,00	1,00	1,09	1,06	1,06	1,06

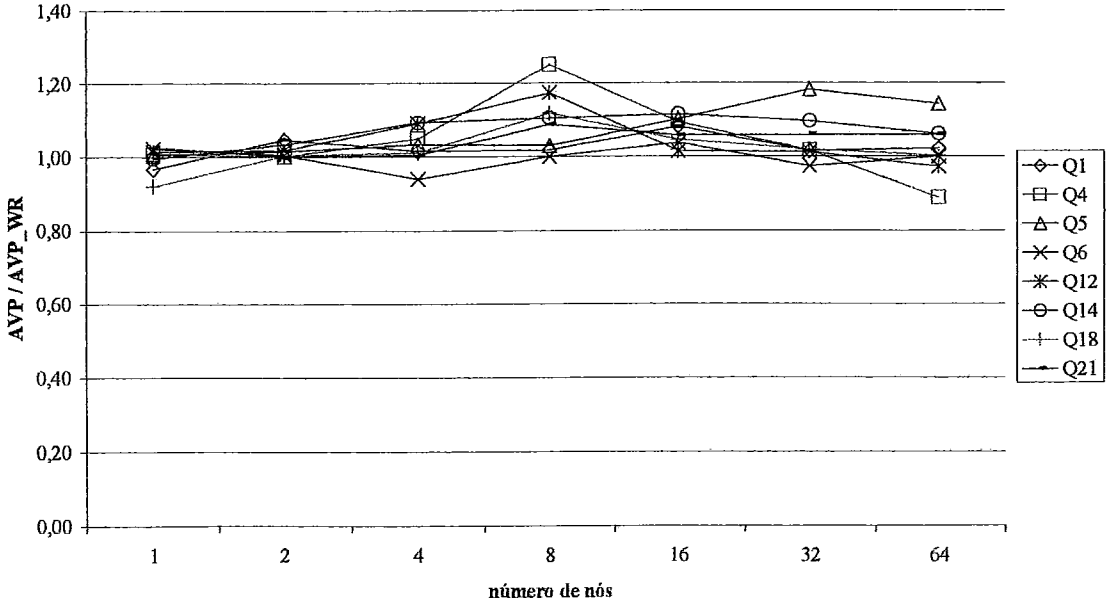

Figura 20- Comparação dos tempos de execução obtidos com AVP e AVP_WR

Como podemos observar, os desempenhos obtidos nas execuções com (AVP_WR) e sem (AVP) o algoritmo de redistribuição são praticamente os mesmos. Os fatores AVP/AVP_WR se situam em torno do valor 1,0 para todos os casos. Na verdade, notamos que, na maioria dos casos, AVP_WR tem melhor desempenho. Para a consulta Q4 com 8 nós, AVP_WR apresenta tempo de execução 1,25 vezes melhor do que a AVP. O melhor caso para a AVP é com a mesma consulta Q4 em 64 nós, onde seu desempenho é 1,13 vezes melhor. Em termos absolutos, esse fator representa uma diferença de 0,23 segundos e o atribuímos a flutuações nas cargas dos nós devido a outros processos em execução, principalmente os do sistema operacional.

O melhor desempenho obtido com a AVP_WR pode ser explicado por alguns fatores. Primeiramente, a uniformidade na distribuição dos valores dos atributos de fragmentação não é perfeita. Efetivamente, os fragmentos virtuais iniciais possuem pequenas diferenças de tamanho em relação ao número de tuplas. Além disso, o processador de consultas é executado pelos nós em simultaneidade com outros processos. Os experimentos foram feitos com utilização exclusiva dos nós do cluster. Porém, o sistema operacional continuava em execução. Esse fato pode levar a flutuações de carga de processamento. A AVP_WR permite que o processador de consultas se ajuste a essas flutuações e mantenha um bom desempenho. Além disso, cada nó do cluster utilizado possui dois processadores. Essa característica contribui para o baixo impacto do algoritmo de redistribuição no desempenho, mesmo em casos de distribuição quase uniforme.

V.3.2. Experimentos com Distorção de Dados Simulada

Como na seção anterior, faremos três tipos de análise. Primeiramente, na seção V.3.2.1, analisamos a aceleração obtida com o uso da AVP_WR. Na seção V.3.2.2, comparamos o desempenho da SVP e da AVP_WR. Por fim, analisamos o impacto do algoritmo de redistribuição dinâmica de carga na seção V.3.2.3.

V.3.2.1 Análise da Aceleração da AVP_WR com Distorção de Dados

A Tabela 8 mostra os tempos de execução obtidos com a utilização da AVP_WR em configurações do cluster com diferentes números de nós, variando de 1 a 64, e distorção de dados.

Tabela 8 - AVP_WR - tempos de execução de consultas com distorção de dados (em segundos)

Consulta	1 nó	2 nós	4 nós	8 nós	16 nós	32 nós	64 nós
Q1	820,40	387,33	189,11	106,81	61,03	47,66	23,65
Q4	357,16	184,44	84,50	47,14	13,95	8,47	5,45
Q5	437,31	287,16	97,44	54,39	24,68	20,63	13,55
Q6	204,43	101,66	48,15	26,78	14,31	8,77	5,76
Q12	367,43	182,83	89,58	50,70	25,38	18,37	10,44
Q14	490,30	289,37	94,78	54,03	34,29	28,18	7,41
Q18	300,45	146,28	75,58	41,62	28,49	15,58	9,27
Q21	545,01	271,79	131,94	80,21	43,26	32,75	18,60

A Figura 21 mostra um gráfico com os mesmos tempos de execução normalizados. Todas as consultas apresentaram redução no tempo de execução com o aumento do número de nós utilizados. As maiores acelerações obtidas foram para as consultas Q14 (redução de 73,69% ao passar de 32 para 64 nós) e Q4 (redução de 70,40% ao passar também de 8 para 16 nós).

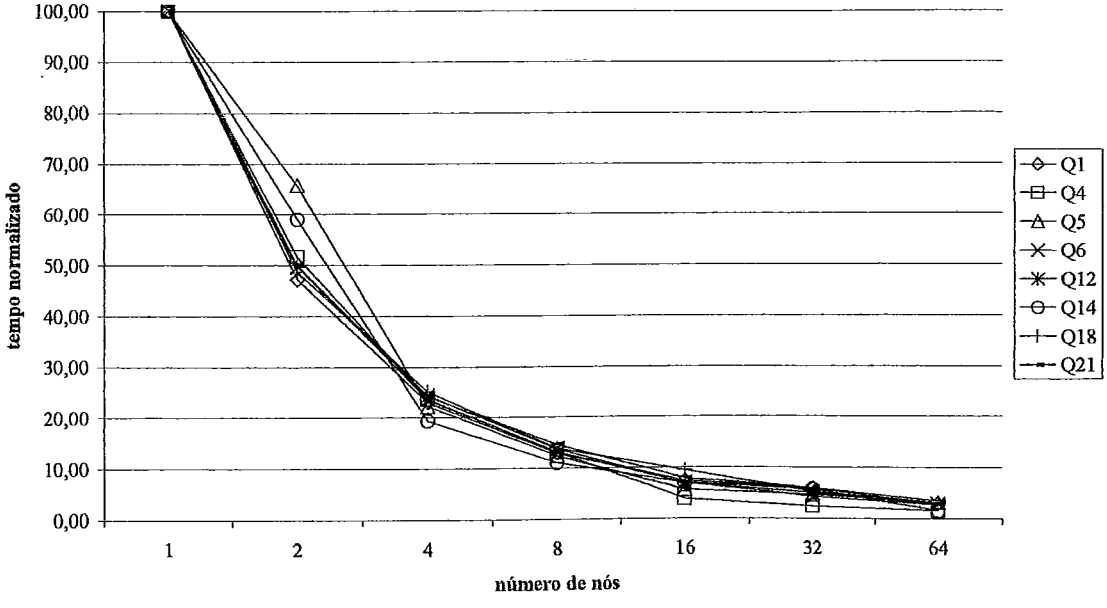

Figura 21 - Tempos de execução normalizados obtidos com AVP_WR e distorção de dados

A aceleração superlinear foi obtida em várias situações. Para as consultas Q1, Q6, Q12 e Q21, ela ocorreu nas configurações com 2 e 4 nós. Para a consulta Q4, com 4, 16, 32 e 64 nós. Para Q5, com 4, 8 e 16 nós. Para Q14, com 4, 8 e 64 nós. Para Q18, apenas com 2 nós. Os piores casos em termos absolutos foram os das consultas Q1 com 32 nós e Q5 com 2 nós. Q1 apresentou tempo de execução superior em 22,02 segundos ao

tempo esperado em caso de aceleração linear para 32 nós. Q5 ficou 68,05 segundos acima desse tempo com 2 nós.

Podemos notar que a AVP_WR apresenta ganho de desempenho consistente apesar da acentuada distorção de dados utilizada. Foram obtidas acelerações próximas da linear em todos os casos, sendo conseguida a superlinear em alguns deles.

Uma análise do número de difusões de ofertas de ajuda executadas pelos nós mostra que um número muito pequeno de ofertas não resulta em ajuda, mesmo em configurações com muitos nós. O gráfico da Figura 22 mostra o número de ofertas difundidas e o número de ofertas não aceitas no processamento de todas as consultas com 64 nós. Podemos notar que poucas são recusadas.

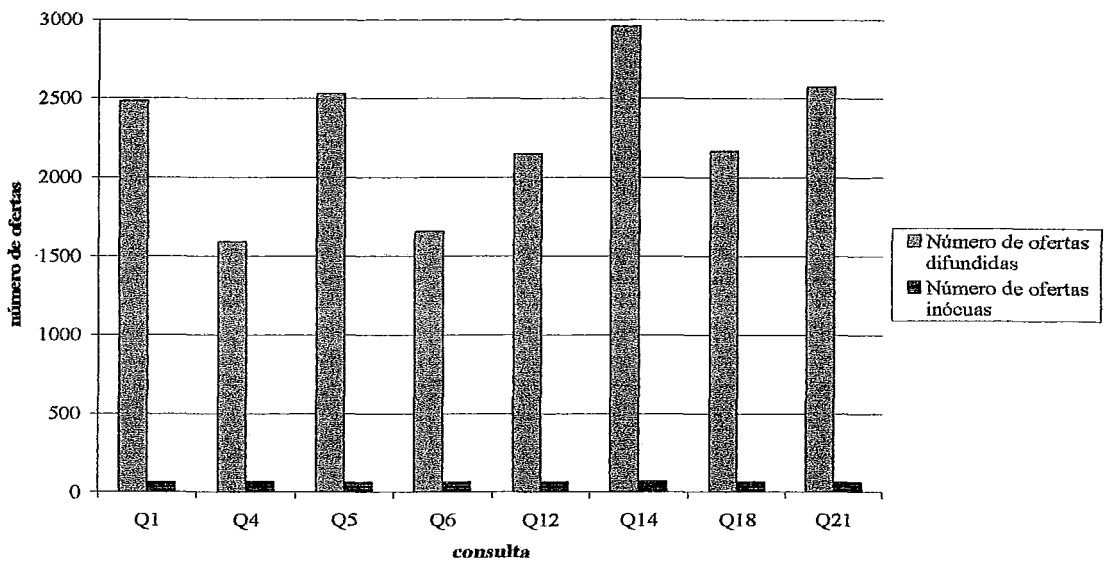

Figura 22 - Aproveitamento de ofertas de ajuda com 64 nós

O número de mensagens individuais propagadas pelo algoritmo, no entanto, é elevado. Os piores casos foram os das consultas Q1, Q14 e Q21 com 64 nós, durante o processamento das quais os nós trocaram, entre mensagens de oferta e aceitação de ajuda, de 40.000 a 45.000 mensagens. Pretendemos investigar modificações no algoritmo que reduzam esse número mantendo a eficiência.

V.3.2.2 Comparação de Desempenho entre SVP e AVP_WR

Comparamos agora os desempenhos obtidos com a utilização de SVP e AVP_WR em presença de distorção de dados. A Tabela 9 mostra os tempos de execução das consultas com a utilização da SVP, em configurações com 1 a 64 nós.

Tabela 9 - SVP - Tempos de execução de consultas com distorção de dados (em segundos)

Consulta	1 nó	2 nós	4 nós	8 nós	16 nós	32 nós	64 nós
Q1	3.299,33	1.818,09	1.379,01	1.022,44	603,28	348,47	330,46
Q4	344,50	236,37	197,03	199,87	96,53	55,19	54,67
Q5	641,80	972,37	766,46	645,48	325,28	145,24	132,69
Q6	115,94	115,59	116,98	140,92	133,80	123,93	125,69
Q12	160,93	147,30	145,47	159,71	134,22	127,57	126,02
Q14	153,09	150,43	149,06	175,33	149,86	150,37	144,49
Q18	10.981,44	849,22	750,61	636,84	448,25	346,88	339,97
Q21	495,50	314,58	269,95	244,92	159,98	87,85	80,95

Para facilitar a comparação, utilizaremos os dados da Tabela 10 e o gráfico da Figura 23. Eles mostram o resultado da divisão do tempo de execução obtido com SVP pelo obtido com AVP_WR para cada consulta em cada uma das configurações utilizadas, com distorção de dados.

Tabela 10 - SVP / AVP_WR com distorção de dados

Consulta	1 nó	2 nós	4 nós	8 nós	16 nós	32 nós	64 nós
Q1	4,02	4,69	7,29	9,57	9,89	7,31	13,97
Q4	0,96	1,28	2,33	4,24	6,92	6,52	10,03
Q5	1,47	3,39	7,87	11,87	13,18	7,04	9,80
Q6	0,57	1,14	2,43	5,26	9,35	14,13	21,81
Q12	0,44	0,81	1,62	3,15	5,29	6,94	12,07
Q14	0,31	0,52	1,57	3,25	4,37	5,34	19,49
Q18	36,55	5,81	9,93	15,30	15,73	22,27	36,67
Q21	0,91	1,16	2,05	3,05	3,70	2,68	4,35

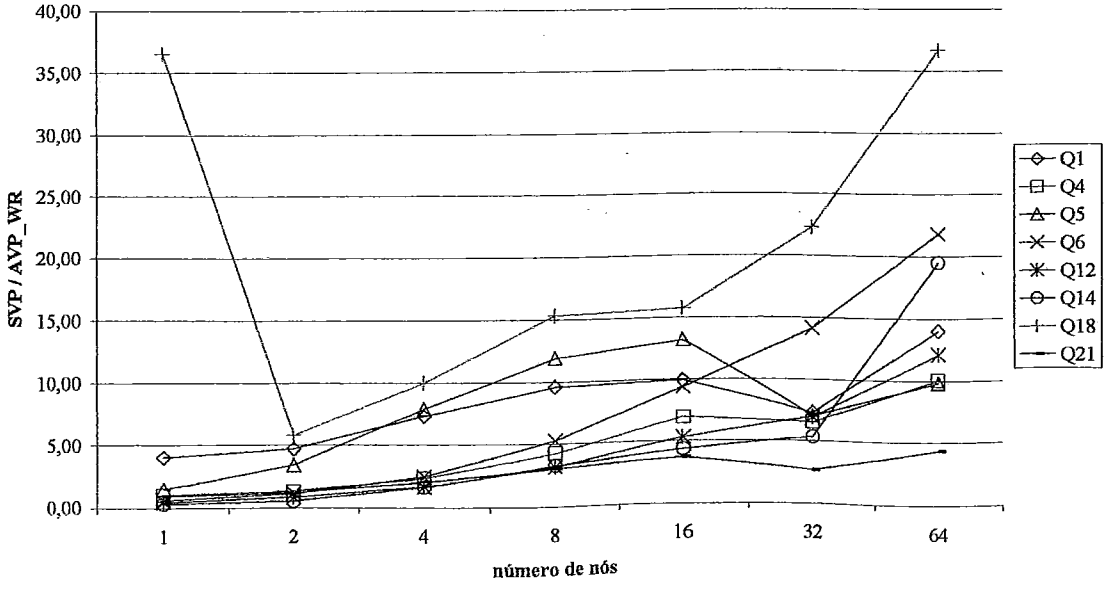

Figura 23 - Comparação dos tempos de execução obtidos com SVP e AVP_WR (distorção)

Não faremos nesse caso uma análise consulta a consulta, por a considerarmos desnecessária. Analisaremos o comportamento da SVP de maneira geral. Como a SVP não possui redistribuição de carga, seu tempo de execução é determinado pelo nó que processa o fragmento com intervalo de maior tamanho. A Figura 24 mostra os diferentes tamanhos dos intervalos que definem 64 fragmentos virtuais. Esses tamanhos serviram como base para a geração dos fragmentos virtuais das outras configurações. Por exemplo, para produzirmos tamanhos a serem utilizados em experimentos com 32 nós, somamos 2 tamanhos consecutivos; para 16 nós, 4 consecutivos; e assim sucessivamente.

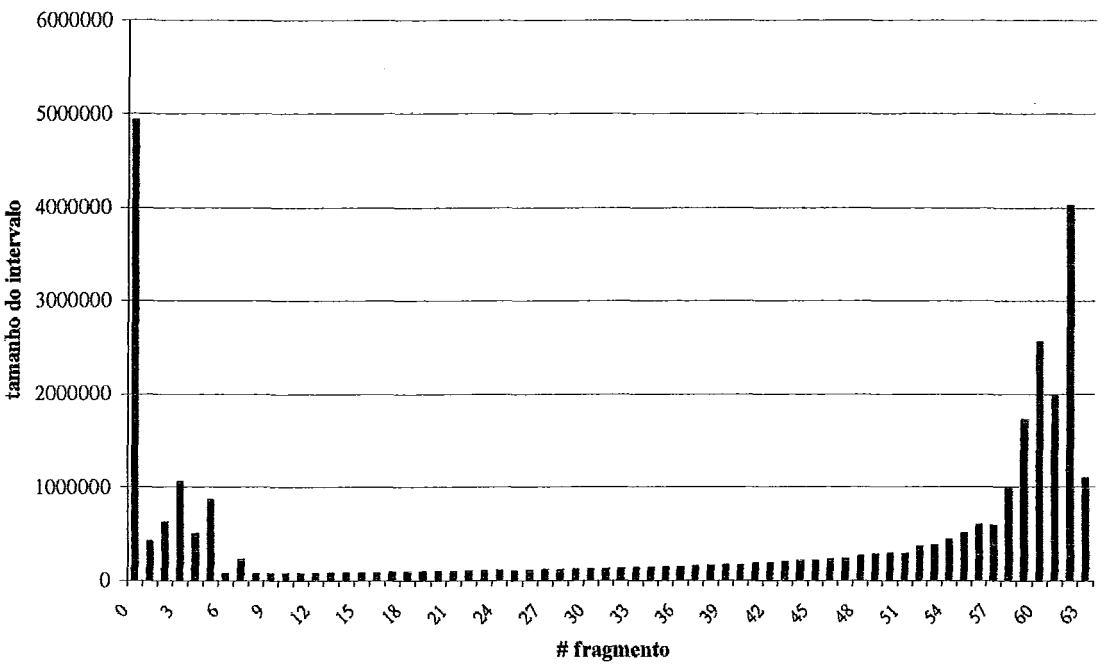

Figura 24 - Tamanhos dos fragmentos virtuais utilizados nos experimentos com distorção de dados

Com 2 e 4 nós, os tamanhos dos fragmentos virtuais são iguais a 19.088.003 e 16.377.919, respectivamente, que são maiores do que os tamanhos dos intervalos usados para 2 nós com distribuição uniforme, que possuem tamanho igual a 15.000.000. Seu desempenho com essas configurações é então inferior ao desempenho com 2 nós no caso uniforme. Com 8 nós, o maior intervalo possui tamanho 13.565.712, próximo ao caso de 2 nós com distribuição uniforme e muito superior ao tamanho usado para 4 nós (7.500.000). Com 16 nós, esse tamanho é igual a 9.660.189, superior ao usado por 4 nós, no caso uniforme. Com 32 e 64 nós, os intervalos possuem tamanhos iguais a 5.355.957 e 4.935.617, respectivamente, ou seja, maiores do que o tamanho utilizado

para 8 nós, no caso uniforme. Concluímos que o desempenho da SVP nesses experimentos não será melhor do que o desempenho obtido com 8 nós no caso uniforme. Por isso, dispensamos a análise consulta a consulta.

Para os experimentos realizados com apenas 1 nó, os desempenhos obtidos com AVP_WR e SVP são os mesmos do caso uniforme, uma vez que a distorção não faz diferença. Também como no caso uniforme, a SVP apresenta melhor desempenho para as consultas Q12 e Q14 com 2 nós, sendo 1,24 e 1,92 vezes melhor do que a AVP_WR, como esperado, dado o tamanho da maior partição. Para todas as demais consultas com 2 nós, a AVP_WR apresentou melhor desempenho. Em todas as outras configurações (4 a 64 nós), o desempenho obtido com AVP_WR foi superior ao obtido com SVP para todas as consultas. A distorção fez com que o desempenho da AVP_WR fosse menor do que no caso uniforme. Por isso, a taxa máxima de ganho com AVP_WR em relação à SVP também foi menor. Ainda assim, foi bastante satisfatória. Nos melhores casos, AVP_WR apresentou desempenhos 36,67, 22,27 (Q18, 64 e 32 nós), 21,81 (Q6, 64 nós) e 19,49 (Q14, 64 nós) vezes superiores aos apresentados pela SVP. Não consideramos o caso da consulta Q18 com 1 nó (36,65 vezes), pois não foi influenciado pela existência da distorção. No pior caso, o desempenho foi 1,14 vezes melhor (Q6, 2 nós).

V.3.2.3 Comparação de Desempenho entre AVP e AVP_WR

Nessa seção, comparamos os desempenhos obtidos com a utilização de AVP e AVP_WR em presença de distorção de dados. Os tempos de execução obtidos com a AVP se encontram na Tabela 11.

Tabela 11 - AVP - Tempos de execução de consultas com distorção de dados (em segundos)

Consulta	1 nó	2 nós	4 nós	8 nós	16 nós	32 nós	64 nós
Q1	770,447	521,315	441,144	359,616	245,165	133,153	125,995
Q4	361,44	233,93	200,66	198,47	110,95	59,26	56,39
Q5	366,40	238,63	197,34	199,09	118,33	85,00	84,49
Q6	203,34	130,26	111,18	114,67	67,71	36,37	35,41
Q12	368,44	239,46	198,36	188,40	114,97	57,31	52,83
Q14	386,12	249,00	215,16	206,17	132,34	82,57	95,21
Q18	275,57	181,09	155,40	152,15	91,67	51,56	49,65
Q21	542,64	336,19	284,93	257,20	177,82	103,34	87,57

A comparação é feita com o auxílio dos dados da Tabela 12 e do gráfico da Figura 25, que mostram o resultado da divisão do tempo de execução obtido com AVP pelo

obtido com AVP_WR para cada consulta em cada uma das configurações utilizadas, com distorção de dados.

Tabela 12- AVP / AVP_WR com distorção de dados

Consulta	1 nó	2 nós	4 nós	8 nós	16 nós	32 nós	64 nós
Q1	0,94	1,35	2,33	3,37	4,02	2,79	5,33
Q4	1,01	1,27	2,37	4,21	7,95	7,00	10,34
Q5	0,84	0,83	2,03	3,66	4,79	4,12	6,24
Q6	0,99	1,28	2,31	4,28	4,73	4,15	6,14
Q12	1,00	1,31	2,21	3,72	4,53	3,12	5,06
Q14	0,79	0,86	2,27	3,82	3,86	2,93	12,84
Q18	0,92	1,24	2,06	3,66	3,22	3,31	5,36
Q21	1,00	1,24	2,16	3,21	4,11	3,16	4,71

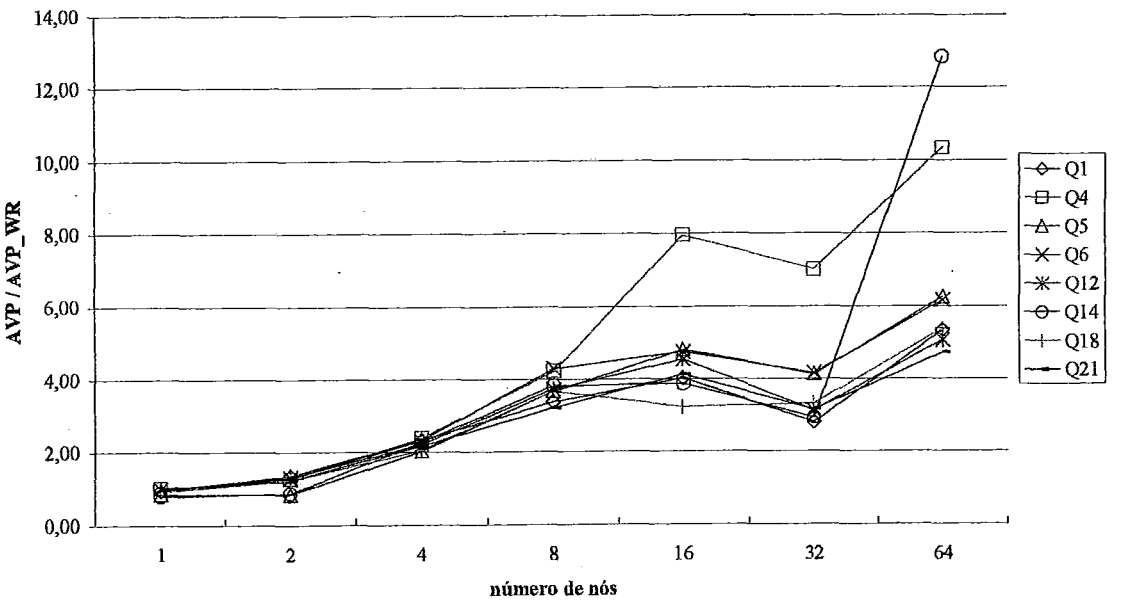

Figura 25 - Comparação dos tempos de execução obtidos com AVP e AVP_WR (distorção)

O objetivo desses experimentos é analisar o impacto da redistribuição de carga no desempenho da fragmentação virtual adaptativa em um ambiente em que ela se faz necessária, no caso, com a presença de distorção de dados. Não analisaremos os casos com um nó pois eles são próximos aos já analisados na seção sobre distribuição uniforme.

Com 2 nós, AVP apresenta melhor desempenho que AVP_WR para as consultas Q5 e Q14. Os tempos de execução obtidos com AVP são 1,20 e 1,16 vezes menores, respectivamente. Analisando o número de operações de entrada e saída de cada nó,

verificamos que, com AVP_WR, foram executadas menos operações no pior caso do que com AVP. Analisando o tempo despendido por cada nó sem executar operações, constatamos que os piores casos foram de apenas 120 mili-segundos para a consulta Q5 (nó 0) e 61 mili-segundos para a consulta Q14 (nó 1). Logo, não podemos atribuir essa diferença nos desempenhos ao algoritmo de redistribuição. Concluímos que ela foi devida a flutuações momentâneas de carga nos nós durante a execução dos experimentos. As demais consultas com 2 nós obtiveram melhor desempenho com AVP_WR do que com AVP.

Em todas as outras configurações (4 a 64 nós), o desempenho obtido com AVP_WR foi superior ao obtido com AVP para todas as consultas. Os melhores casos foram os das consultas Q14 e Q4, para as quais a AVP_WR apresentou desempenhos 12,84 (64 nós) e 10,84 (64 nós) vezes melhores do que os da AVP, respectivamente. Nessas configurações, o pior caso foi o desempenho obtido para a consulta Q5 com 4 nós, apenas 2,03 vezes superior ao da AVP.

Como conclusão geral dessa série de experimentos com execução de consultas isoladas, ressaltamos o melhor desempenho e a maior robustez da AVP_WR em relação à SVP e à AVP. Em ambos os cenários avaliados, a AVP_WR apresentou aceleração superlinear em vários casos e próxima à linear em outros.

Com distribuição uniforme de dados, a AVP_WR mostrou-se muito menos suscetível às idiossincrasias do SGBD do que a SVP, obtendo ganhos de aceleração consistentes de acordo com o aumento do número de nós utilizados. Além disso, o algoritmo de redistribuição de carga mostrou pouco ou nenhum impacto negativo mesmo quando sua presença não se faz tão necessária.

Nos cenários com distorção de dados, a AVP_WR apresentou melhor desempenho do que SVP e AVP para todas as consultas em todas as configurações com mais de 2 nós. Com 2 nós, obteve pior desempenho em raros casos.

V.4. Experimentos com Lotes de Consultas

Nessa seção, analisamos o comportamento do processador de consultas para clusters em cenários com execução simultânea de lotes de consultas. O objetivo é analisar a vazão e aceleração num cenário representando a utilização do cluster de

bancos de dados por usuários submetendo simultaneamente consultas analíticas (OLAP). Em geral, análises de dados são iterativas e não realizadas através de uma só consulta. O usuário submete uma consulta, analisa os resultados, refina a consulta, submete a nova consulta ao banco de dados e assim sucessivamente. Outra característica de OLAP é que o número de usuários acessando concorrentemente é baixo, ao contrário de OLTP.

Para simularmos esse comportamento, utilizamos processos concorrentes em uma máquina cliente. Cada processo submete seqüencialmente um lote de consultas ao cluster de bancos de dados. Consultas de processos diferentes são submetidas paralelamente. Como sugere o TPC-H, utilizamos lotes com as mesmas consultas ordenadas de maneiras diversas. Como trabalhamos com oito consultas, produzimos oito tipos de lotes, correspondentes a diferentes permutações, como mostra a Tabela 13.

Tabela 13 - Lotes de consultas

Lote	Consultas
L0	{Q21, Q18, Q5, Q6, Q12, Q4, Q14, Q1}
L1	{Q6, Q14, Q12, Q4, Q5, Q1, Q18, Q21}
L2	{Q5, Q4, Q6, Q1, Q18, Q14, Q21, Q12}
L3	{Q12, Q21, Q4, Q18, Q1, Q5, Q6, Q14}
L4	{Q18, Q1, Q14, Q5, Q21, Q12, Q4, Q6}
L5	{Q1, Q5, Q21, Q12, Q14, Q6, Q18, Q4}
L6	{Q4, Q12, Q18, Q14, Q6, Q21, Q5, Q1}
L7	{Q14, Q6, Q1, Q21, Q4, Q18, Q12, Q5}

Se, em um experimento, necessitamos de n lotes ($n \leq 8$), tomamos os n primeiros da lista mostrada na Tabela 13. Para casos com $n > 8$, tomamos os oito lotes (n div 8) vezes, onde div representa a divisão inteira, mais os (n mod 8) primeiros, onde mod representa o resto da divisão inteira. Como exemplo, se necessitamos simular dez usuários diferentes, tomamos os lotes na seguinte ordem: L0, L1, L2, L3, L4, L5, L6, L7, L0 e L1.

Os resultados dos experimentos descritos na seção V.3 indicaram a AVP_WR como melhor estratégia para processamento de consultas isoladas sobre o cluster tanto em relação ao desempenho quanto à robustez, tanto na distribuição uniforme quanto na distorcida. Por isso, nos experimentos dessa seção, utilizamos apenas a AVP_WR para fragmentação virtual.

O banco de dados TPC-H gerado nos nossos testes possui fator de escala 5. Para testes com lotes de consultas (análise de vazão), o TPC-H (TPC, 2003a) especifica que com bancos gerados com fator de escala 1, deve-se utilizar, no mínimo, dois lotes, e com bancos de escala 10, três lotes. Baseados nessa recomendação e no fator de escala que utilizamos, consideramos como cenário típico para aplicações OLAP a utilização de três lotes, simulando três usuários simultâneos. Os resultados de utilização do *benchmark* disponíveis no sítio do TPC (TPC, 2004) utilizam sempre o número mínimo recomendado para o seus fatores de escala.

Nas seções a seguir, descrevemos os experimentos realizados. Primeiramente, analisamos, na seção V.4.1, a aceleração obtida no processamento de três lotes individuais, sem considerar execuções concorrentes. Na seção V.4.2, analisamos a vazão obtida com a utilização de paralelismo intra-consulta no processamento de lotes concorrentes. Finalmente, na seção V.4.3, analisamos vários aspectos de comportamento do processador de consultas para clusters que propusemos no cenário típico para consultas OLAP (com três lotes de consultas concorrentes).

V.4.1. Aceleração no Processamento de Lotes Individuais

Analisaremos primeiramente a aceleração obtida com a AVP_WR durante o processamento de três lotes individuais de consultas com distribuição de dados uniforme e distorção. Os lotes foram avaliados individualmente, sem concorrência. O gráfico da Figura 26 mostra os tempos de execução normalizados obtidos para os três lotes com distribuição uniforme de dados. O experimento mostra que foram obtidas acelerações similares para os três lotes. Em vários casos, aceleração superlinear foi conseguida: L0, com 16 e 32 nós, L1, com 16, 32 e 64 nós e L2, com 32 e 64 nós. No pior caso, L2 com 4 nós, a aceleração foi apenas 4,92% inferior à linear. A aceleração do lote L0 com 64 nós ficou apenas 0,012% abaixo da linear.

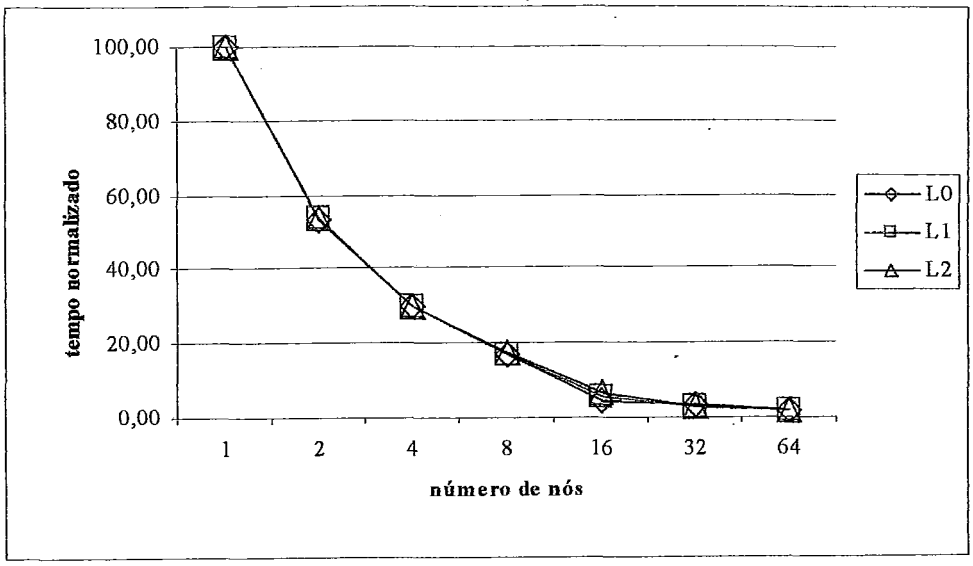

Figura 26 - Tempo normalizado de processamento de três lotes sem concorrência com distribuição uniforme de dados

Avaliamos também a aceleração obtida com distorção de dados. Os resultados estão no gráfico da Figura 27. Mais uma vez, foi obtida aceleração em todas as configurações. Obtivemos acelerações próximas à linear. No pior caso, Lote L0 com 4 nós, a aceleração apresentada é apenas 7,88% menor do que a linear.

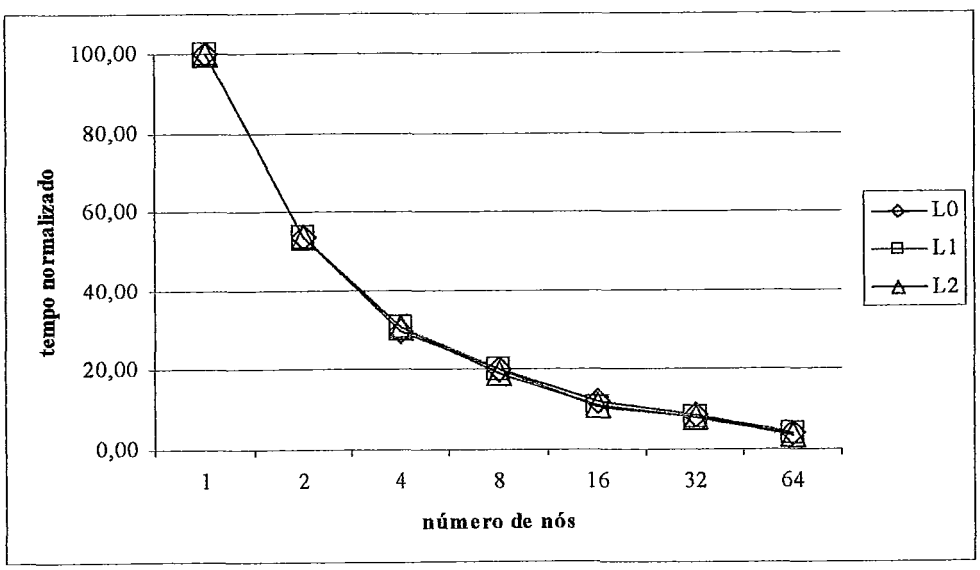

Figura 27 - Tempo normalizado de processamento de três lotes sem concorrência com distorção de dados

V.4.2. Paralelismo Intra-Consulta no Processamento de Lotes Concorrentes

No processamento de consultas concorrentes em um cluster, duas opções foram avaliadas. A primeira, que denominamos "inter-c", consiste na utilização exclusiva de paralelismo inter-consultas. Ou seja, cada nova consulta a ser processada é direcionada para um dos nós do cluster e executada de modo seqüencial. Consultas diferentes, são alocadas em nós diferentes, se houver disponibilidade. A segunda opção, que denominamos "intra-c", consiste na utilização de paralelismo inter- e intra-consulta para o processamento de cada consulta. Nessa estratégia, todas as consultas são processadas paralela e simultaneamente por todos os nós do cluster utilizando a AVP_WR.

Nessa seção, avaliamos a vazão obtida com a utilização de ambas as estratégias. O número de lotes utilizados nesses experimentos é igual ao número de nós. Como mostram os gráficos da Figura 28 e da Figura 29, foram utilizadas configurações com 4, 8 e 16 nós e, conseqüentemente, 4, 8 e 16 lotes, respectivamente. Configurações com 1 e 2 nós ficariam aquém do cenário típico escolhido (3 lotes). Configurações com 32 e 64 nós seriam irrealistas devido ao grande número de lotes a serem utilizados.

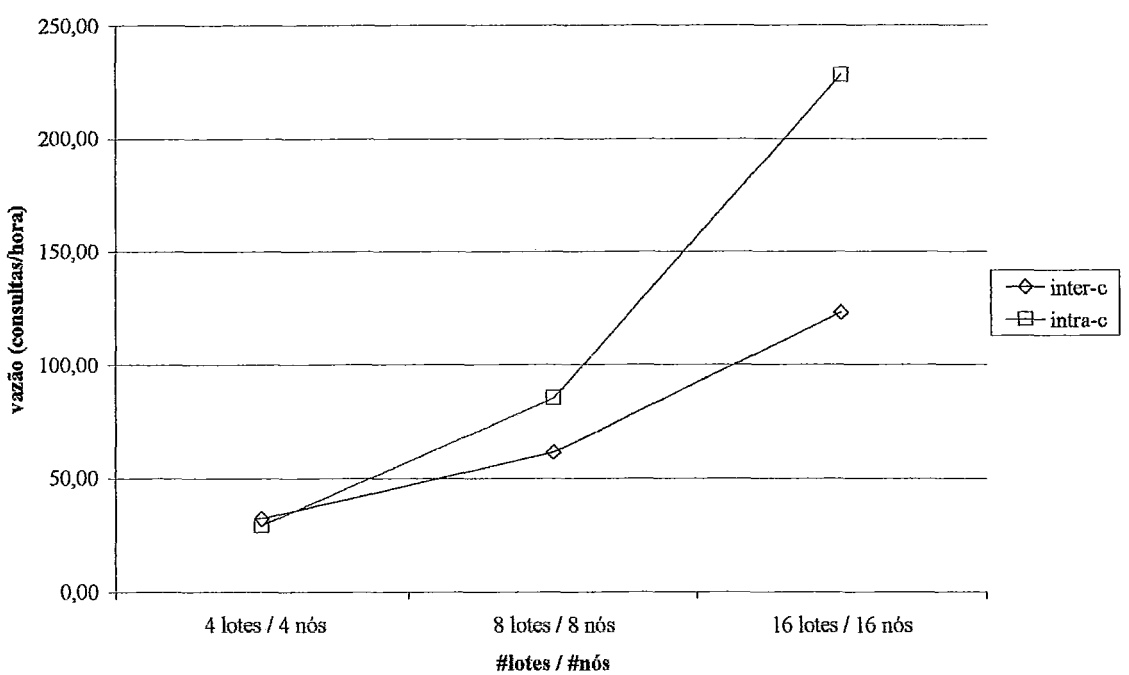

Figura 28 - Vazão obtida com o uso das estratégias inter-c e intra-c (distribuição uniforme)

O gráfico da Figura 28 mostra a vazão (em consultas por hora) obtida com cada uma das estratégias nas configurações utilizadas, com distribuição uniforme de dados. Com 4 nós/lotes, as vazões são próximas, sendo a obtida com inter-c apenas 9,53% acima da obtida com intra-c. Com 8 e 16 nós/lotes, a estratégia intra-c obtém vazões 39,04% e 85,59% melhores do que as obtidas com inter-c, respectivamente. Em relação ao aumento da vazão obtida com 4 nós, a estratégia inter-c apresenta ganho sublinear com 8 nós, obtendo vazão 1,91 vezes maior, e superlinear com 16 nós, obtendo vazão 3,84 vezes superior. A estratégia intra-c, no entanto, apresenta ganho superlinear em ambos os casos: com 8 nós, a vazão é 2,28 vezes superior à obtida com 4 nós, e com 16 nós, ela é 7,87 vezes maior. Os resultados indicam uma melhoria mais acentuada de desempenho com o aumento de recursos para a estratégia intra-c.

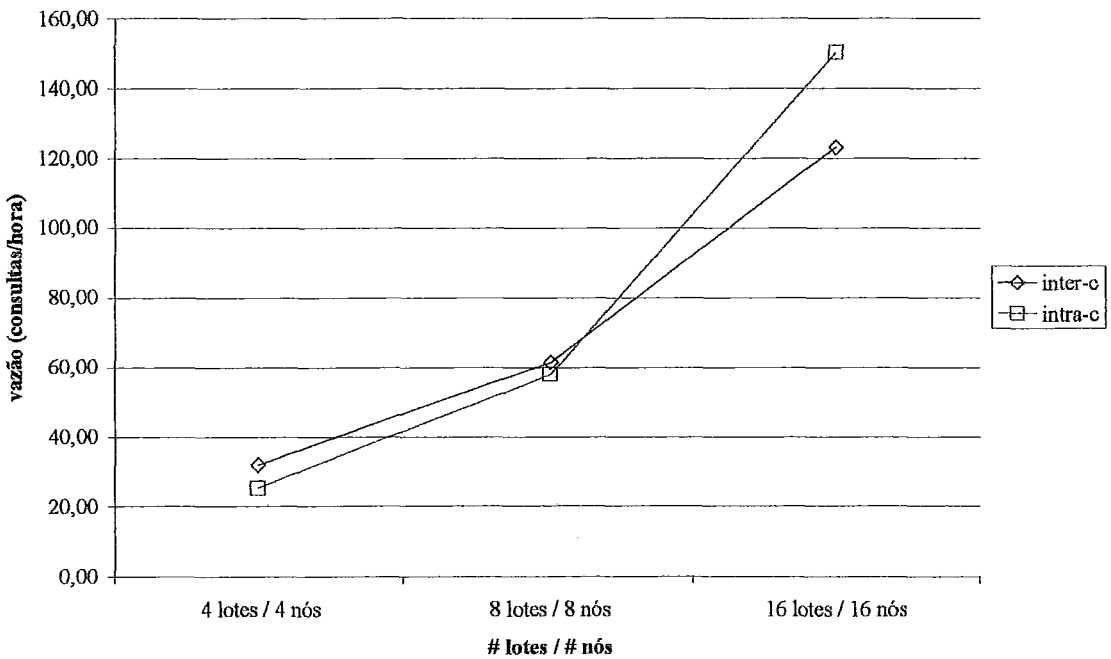

Figura 29 – Vazão obtida com o uso das estratégias inter-c e intra-c (distorção)

O gráfico da Figura 29 mostra a vazão obtida com o uso das estratégias, com distorção de dados. As vazões obtidas com a estratégia inter-c são idênticas ao caso uniforme, uma vez que não há paralelismo intra-consulta. Ela mostrou melhor desempenho do que a intra-c com 4 nós/lotes, apresentando vazão 20,09% superior. Para 8 nós/lotes, a diferença decresce para 5,66% a favor da inter-c. Com 16 nós/lotes, porém, a vazão obtida com intra-c é 22,07% superior. A melhoria de desempenho da estratégia intra-c foi mais uma vez superior. Com 8 nós o aumento da vazão foi mais

uma vez superlinear, sendo 2,28 vezes maior do que a obtida com 4 nós. Com 16 nós, a vazão foi 5,93 vezes maior do que a obtida com 4 nós, também superlinear. Mais uma vez, o ganho de desempenho é mais acentuado para a estratégia intra-c com o aumento de recursos utilizados.

Concluímos, pelos resultados apresentados, que a estratégia intra-c, que utiliza paralelismo inter- e intra-consulta, é mais eficiente do que a inter-c, apresentando maior ganho de desempenho com o aumento dos recursos disponíveis.

V.4.3. Análise de Cenário Típico para Aplicações OLAP

Nessa seção, analisamos o desempenho do processador de consultas no cenário OLAP típico para o tamanho do banco de dados utilizados, ou seja, com lotes de consultas concorrentes, simulando a utilização simultânea do sistema por três usuários diferentes.

Iniciemos pela análise do aumento da vazão de acordo com o aumento de recursos utilizados na base de dados com distribuição uniforme. Os resultados desses experimentos são mostrados pelo gráfico da Figura 30. Podemos notar aumento acentuado da vazão de acordo com o aumento do número de nós utilizados. Esse aumento é sublinear até 8 nós. A partir de 16 nós, no entanto se torna superlinear. Com 64 nós, vazão atinge 1072,73 consultas por hora, ficando 135,34% acima do ganho linear. Atribuímos o aumento acentuado nas configurações acima de 16 nós à queda abrupta no número de operações de entrada e saída em disco, que é de 75% de 8 para 16 nós e continua decaindo para 32 e 64 nós. Isso indica bom aproveitamento do *cache* por parte do SGBD, característico da AVP_WR.

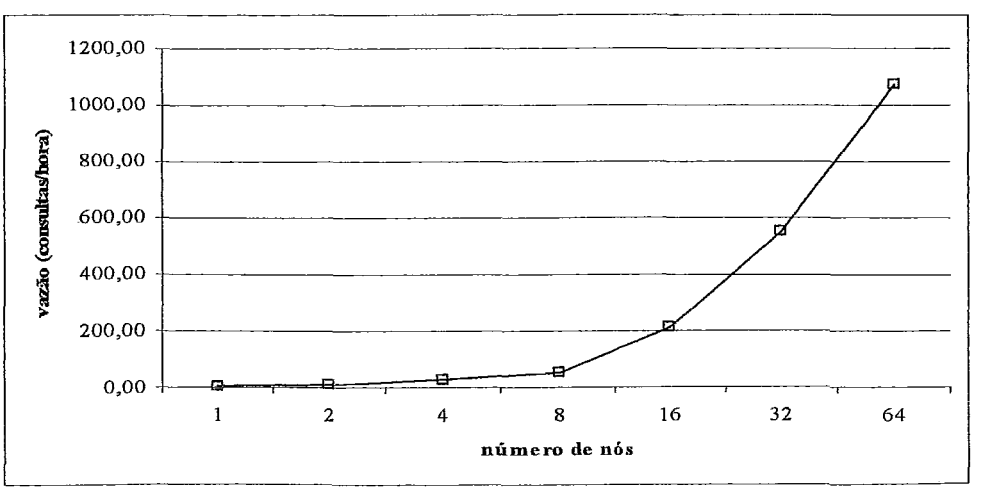

Figura 30 - Vazão obtida com 3 lotes de consultas simultâneos (distribuição uniforme)

O gráfico da Figura 31 mostra o mesmo tipo de experimento realizado com distorção de dados. Como podemos notar, há também aumento de vazão. Porém, ele é menos acentuado, como se poderia esperar. Em todos os casos, o aumento é sublinear. No melhor caso, é apenas 7,53% menor do que o linear (4 nós). No pior caso, fica 43,82% abaixo do ganho linear (32 nós). A vazão máxima obtida (64 nós) é de 313,38 consultas/hora. Apesar do ganho de desempenho, notamos que a maior vazão é, em termos absolutos, 3,42 vezes menor do que a obtida no caso uniforme. Atribuímos esse fato à distorção simulada provocada pela distribuição *zipf* que utilizamos. Outros trabalhos, como o de WOLF *et al.* (1993), usam a distribuição *zipf* dessa maneira para simular casos extremos de distorção. Variações podem ser aplicadas à fórmula utilizada a fim de se obter distorções mais moderadas. Devido à intenção de avaliarmos o desempenho da AVP_WR mesmo em casos extremos, não fizemos experimentos com as variações da *zipf*. Isso justifica a baixa vazão sem contanto desfavorecer a AVP_WR, mostrando sua escalabilidade mesmo em condições muito desfavoráveis.

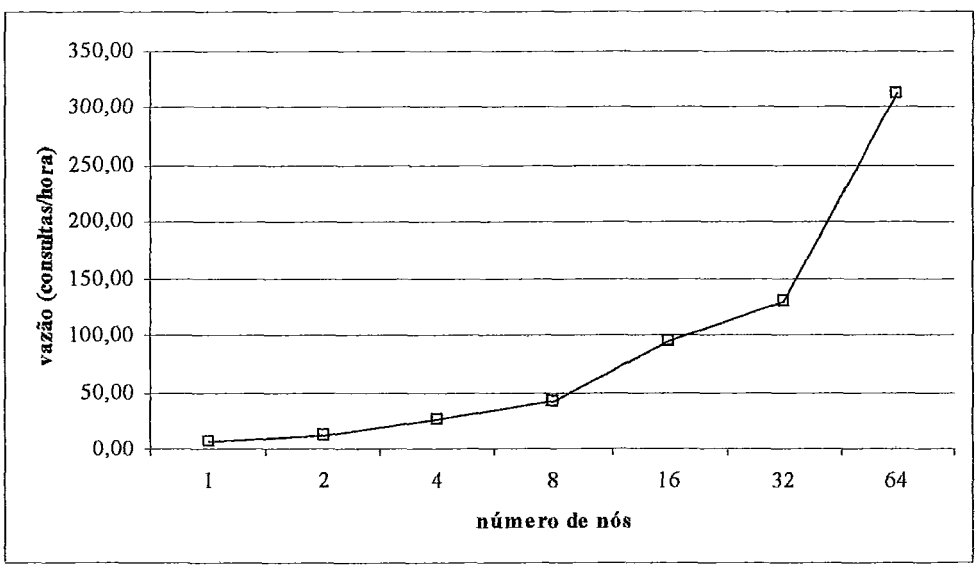

Figura 31 - Vazão obtida com 3 lotes de consultas simultâneos (distorção)

Avaliemos agora o uso das estratégias inter-c e intra-c (descritas na seção V.4.2) no cenário OLAP típico. Os gráficos da Figura 32 e da Figura 33 mostram essa comparação em experimentos com distribuição de dados uniforme e distorção de dados, respectivamente. A estratégia inter-c apresenta melhoria na vazão apenas até a configuração com 4 nós. A partir daí, a vazão apresentada mantém-se a mesma. Esse

comportamento é explicado pelo fato de que ela utiliza no máximo três nós do cluster simultaneamente, pois as consultas de cada lote são submetidas seqüencialmente e cada uma é executada em um nó diferente. Ao contrário, a estratégia intra-c utiliza sempre todos os nós disponíveis. Concluímos com isso que a utilização de inter-c provoca desperdício de recursos em cenários OLAP típicos.

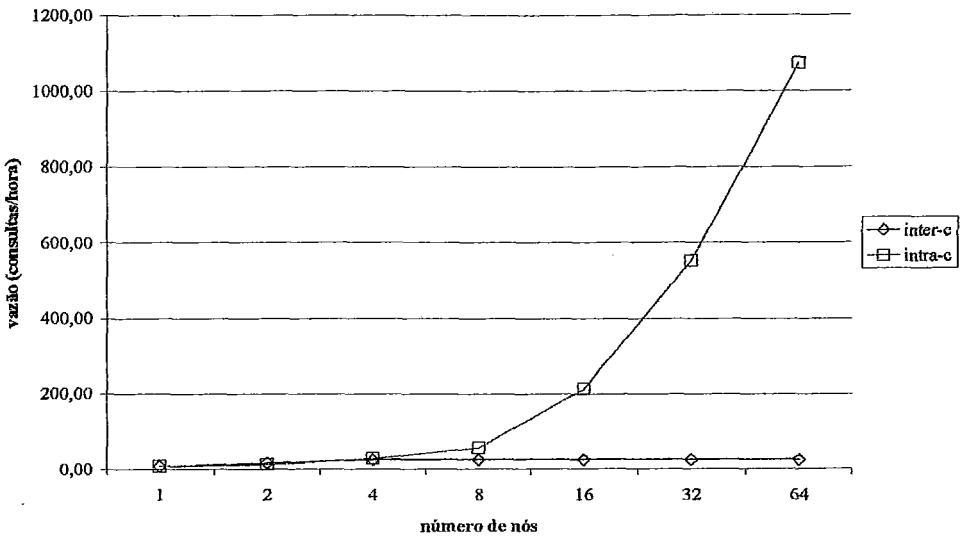

Figura 32 - Vazão obtida com inter-c e intra-c para 3 lotes de consultas (distribuição uniforme)

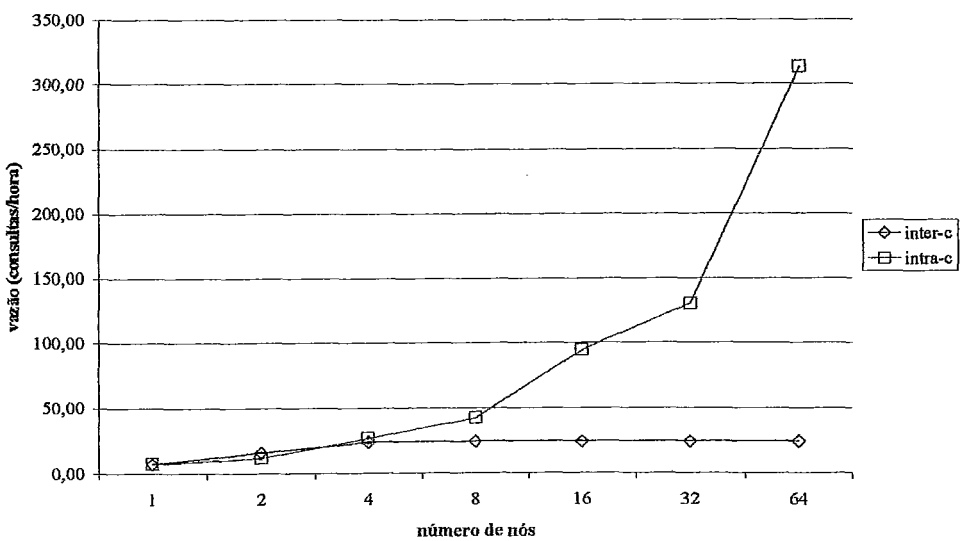

Figura 33 - Vazão obtida com inter-c e intra-c para 3 lotes de consultas (distorção)

Por fim, analisemos o tempo de espera do usuário pelo resultado de uma única consulta com a utilização das estratégias inter-c e intra-c. Para prosseguir com suas

análises, os usuários devem aguardar o término de suas consultas individuais. As consultas com maior tempo médio de execução nos nossos experimentos com 3 lotes concorrentes foram Q5 e Q14, executadas com 1 nó em torno de 1900 e 4000 segundos, respectivamente. A Figura 34 mostra a aceleração obtidas por essas consultas com a utilização das estratégias inter-c e intra-c e distribuição uniforme de dados. Essas consultas são aquelas para as quais obtivemos pior desempenho com AVP_WR nos experimentos com execuções isoladas. Com 2 nós, as execuções com intra-c obtêm desempenho inferior. Com 4 nós, Q14 apresenta melhor desempenho com intra-c. A partir de 8 nós, a aceleração é maior com intra-c, enquanto não há ganho com inter-c. Em termos absolutos, Q5 é executada em 10,25 segundos com 64 nós utilizando intra-c, enquanto a utilização de inter-c acarreta tempo de execução de 436,8 segundos com a mesma configuração. Q14, com 64 nós, é executada em 9,67 segundos com intra-c e em 520,0 segundos com inter-c. A utilização de intra-c agiliza o processo de análise do usuário.

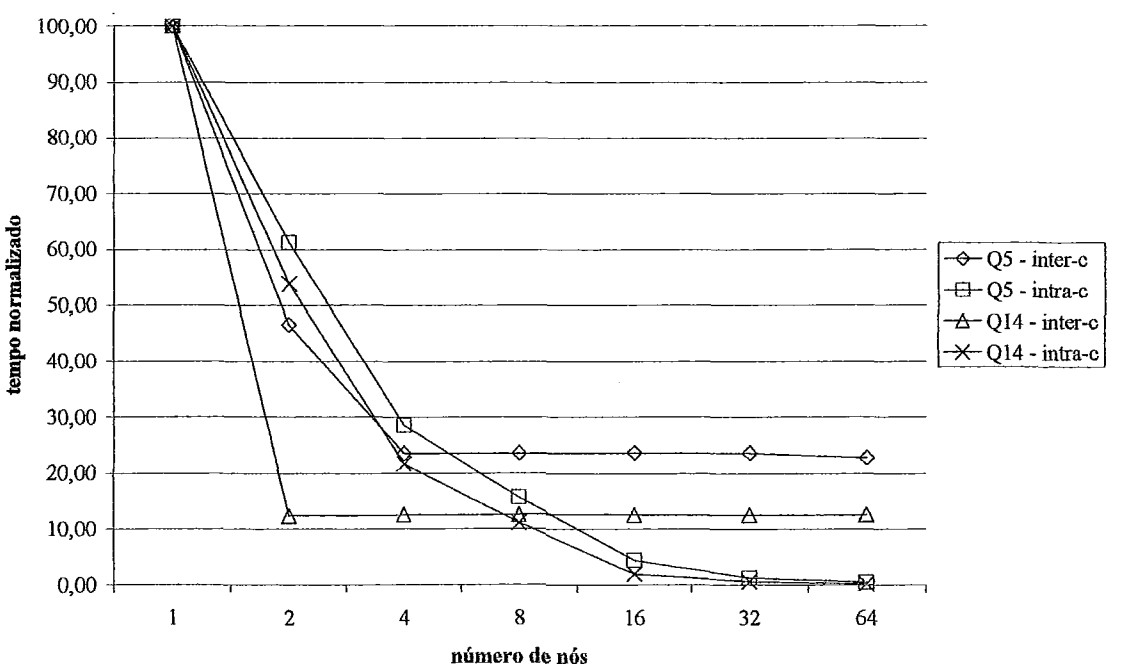

Figura 34 - Consultas Q5 e Q14 - tempo de execução normalizado com 3 lotes (uniforme)

Os mesmos experimentos foram realizados em caso de distorção de dados e os seus resultados são mostrados no gráfico da Figura 35. Notamos que Q5 começa a ter melhor desempenho com intra-c a partir da utilização de 8 nós e Q14, a partir de 16 nós. Com 32 e 64 nós, ambas apresentam melhor desempenho com intra-c. Comparando os

tempos de execução com 64 nós, Q5 é executada em 59,07 segundos com intra-c e em 436,8 segundos com inter-c, enquanto Q14 é executada em 82,22 segundos com intra-c e em 520,0 segundos com inter-c. Mesmo na presença de distorção, o processo de análise será mais ágil com o uso da estratégia intra-c.

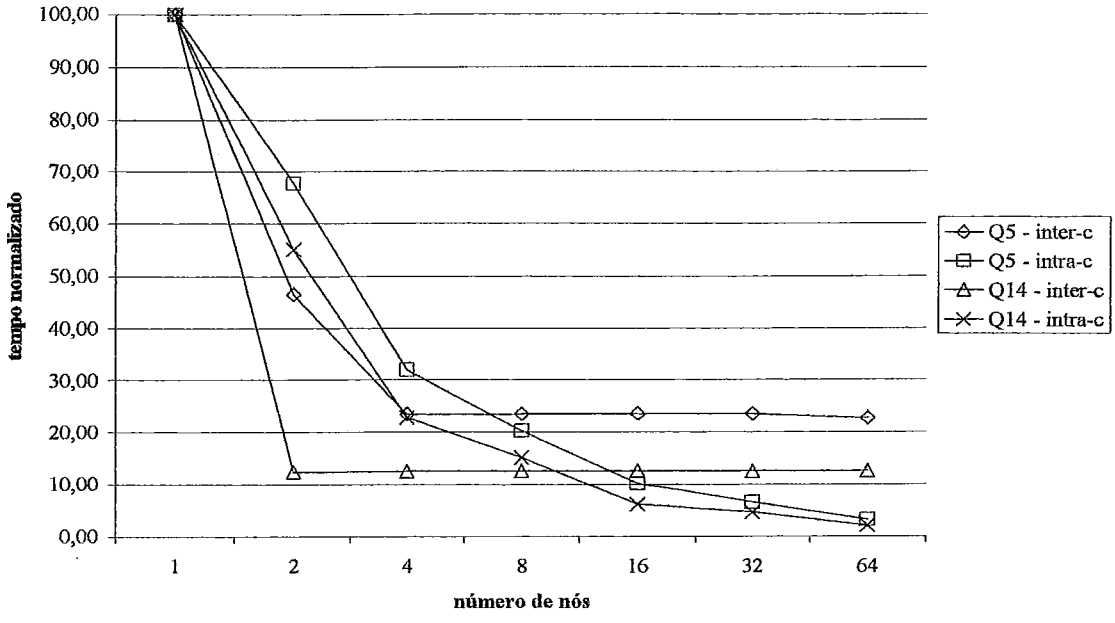

Figura 35 - Consultas Q5 e Q14 - tempo de execução normalizado com 3 lotes (distorção)

Como conclusão dessa série de experimentos, ressaltamos o bom desempenho da AVP_WR no processamento de lotes de consultas concorrentes, tanto em cenários OLAP típicos quanto em cenários com grande número de lotes e distorção de dados acentuada. No caso de distribuição de dados uniforme, aumentos super-lineares da vazão foram obtidos em várias situações, atingindo 1072,73 consultas/hora. Na presença de distorção de dados extrema, como a utilizada em nossos experimentos, o aumento da vazão também foi significativo.

Comparando a utilização exclusiva de paralelismo inter-consultas com a utilização da combinação inter-/intra-consulta, no cenário OLAP, concluímos que a estratégia que melhor utiliza os recursos de processamento do cluster é a segunda. Além disso, é a que proporciona maior agilidade nos processos de análise de dados.

VI. Conclusão

A utilização de paralelismo intra-consulta em sistemas de bancos de dados é uma das estratégias mais empregadas para o processamento de consultas de alto custo. Tradicionalmente, tem-se investigado e proposto técnicas para a obtenção desse tipo de paralelismo em sistemas com hardware especializado e versões específicas de SGBDs para o processamento paralelo. Tais soluções tendem a ser dispendiosas e normalmente requerem grandes investimentos por parte das empresas e instituições que as adotam. Computadores e SGBDs paralelos costumam ser produtos de alto custo, o que encarece essa alternativa. Além disso, ela costuma exigir grande esforço quando se deseja migrar de um ambiente seqüencial tradicional para uma plataforma desse tipo. Esse esforço envolve, na maioria das vezes, a re-elaboração do projeto do banco de dados para sua utilização eficiente no novo ambiente.

O declínio nos preços dos processadores, discos magnéticos e dispositivos de interconexão, aliado ao aumento da sua capacidade e desempenho, fez surgir uma alternativa aos computadores de arquitetura paralela para a obtenção de alto desempenho computacional. Tal alternativa são os clusters de PCs, que figuram, hoje, entre as plataformas com mais alto poder de processamento disponíveis (TOP500, 2004). Eles são equivalentes a computadores paralelos com arquitetura de memória distribuída. Naturalmente, surgiram iniciativas no sentido de se utilizar tal plataforma para suporte a sistemas de bancos de dados, o que resultou na definição de uma nova arquitetura, denominada "cluster de bancos de dados".

O cluster de bancos de dados foi definido por AKAL *et al.* (2002) como sendo um cluster de PCs, executando, cada um deles, um SGBD padrão, ou seja, seqüencial, não estendido para utilização em clusters. Em um cluster de bancos de dados, os SGBDs são utilizados como componentes do tipo "caixa-preta", sendo coordenados por uma camada de software intermediária (*middleware*), responsável pela orquestração desse componentes para a obtenção de paralelismo.

Há vários trabalhos que investigam a utilização de clusters de bancos de dados. Muitos deles utilizam replicação total da base de dados, ou seja, cada nó possui uma cópia da base. Tal alternativa é a que exige menor esforço de migração para aplicações provenientes de um ambiente seqüencial. Os trabalhos investigados propõem soluções

para a utilização de clusters em aplicações OLAP e OLTP. No entanto, em apenas um deles há uma proposta para a obtenção de paralelismo intra-consulta.

AKAL *et al.* (2002) propõem a fragmentação virtual como estratégia para o processamento de consultas OLAP. Trata-se de uma técnica simples, que implementa paralelismo intra-consulta fazendo com que cada nó execute uma mesma consulta sobre diferentes sub-conjuntos de dados, que são determinados por predicados acrescentados à consulta original. A fragmentação virtual assim proposta, a que nos referimos como SVP, é bastante atraente por sua simplicidade e obtém bom desempenho em várias situações. No entanto, sob certas condições não raras, pode apresentar desempenho muito aquém do esperado, principalmente em situações onde os tamanhos dos fragmentos virtuais sejam muito grandes e em casos de distorção de dados. Isso a torna uma técnica não muito robusta pois sua eficiência depende em grande parte das características da base de dados e do SGBD utilizado no cluster.

A obtenção de bom desempenho no processamento de consultas de alto custo em clusters de bancos de dados é um problema complexo. A utilização exclusiva de paralelismo inter-consultas, como no trabalho de RÖHM *et al.* (2002), não reduz o tempo de execução de consultas individuais. Sua única vantagem é aumentar a vazão do sistema quando há um número elevado de usuários concorrentes, o que não é típico em aplicações OLAP, por exemplo. Para se obter a redução do tempo de processamento de consultas individuais, a solução mais indicada é a combinação de paralelismo inter-consultas com paralelismo intra-consulta. Esse fato fica evidente ao analisarmos o bom desempenho de soluções proprietárias que implementam essa estratégia, como o Oracle 10g (ORACLE, 2004) e o IBM DB2 ICE (DB2, 2004). No entanto, essas soluções costumam ser de elevado custo, além de não serem consoantes com a opção dos clusters de bancos de dados de utilizar SGBDs como componentes do tipo "caixa-preta".

A análise dos problemas da proposta de AKAL *et al.* (2002) mostra que a implementação de paralelismo intra-consulta em clusters de bancos de dados com SGBDs "caixas-pretas" não é uma tarefa trivial. Idiossincrasias do SGBD utilizado no cluster podem reduzir muito o desempenho dessa solução. Essa característica advém do fato de que não se pode ter acesso a informações privilegiadas a respeito dos bancos de dados como, por exemplo, a organização física dos dados e das suas estruturas de acesso, bem como dos algoritmos utilizados pelo SGBD para o processamento de consultas. A utilização dessas informações exigiria o desenvolvimento de soluções de

pouca portabilidade, que deveriam ser adaptadas para cada novo tipo de SGBD utilizado. Uma solução alternativa seria coletar periodicamente estatísticas a respeito dos dados e utilizá-las na determinação de fragmentos virtuais. Porém, os tipos de estatísticas fornecidas por SGBDs de diferentes fabricantes costumam ser também diferentes, bem como o modo pelo qual elas podem ser obtidas. Além disso, tal abordagem provavelmente aumentaria a complexidade do processo de determinação dos fragmentos, o que poderia afetar negativamente o desempenho do processamento de consultas, principalmente na sua fase inicial. Por esse motivo, optamos por uma abordagem adaptativa, independente de tais informações.

Nesta tese, é proposta uma estratégia eficiente para obtenção de paralelismo intra-consulta. Essa estratégia tem como componentes principais a Fragmentação Virtual Adaptativa (AVP) e um algoritmo para redistribuição dinâmica de carga. Como a SVP, nossa solução pode ser utilizada em arquiteturas de disco compartilhado e em arquiteturas de memória distribuída com replicação total de dados. Sua utilização em arquiteturas de memória distribuída com replicação parcial de dados exige pequenas adaptações no algoritmo de redistribuição dinâmica de carga.

As principais contribuições desta tese são:

1. A especificação de uma arquitetura para processadores de consultas em clusters de bancos de dados.
2. A proposta da AVP, uma técnica adaptativa para fragmentação virtual eficiente que supera os problemas da SVP mantendo sua portabilidade.
3. A proposta de um algoritmo descentralizado para redistribuição dinâmica de carga entre nós do cluster durante o processamento de consultas de alto custo.
4. O desenvolvimento de um protótipo de código aberto que implementa a arquitetura e as técnicas propostas utilizando apenas software livre e clusters de PCs.
5. A obtenção de resultados com alto desempenho a partir de experimentos realizados com o *benchmark* TPC-H que mostram a superioridade da utilização combinada de paralelismo inter-consultas e intra-consulta sobre a utilização exclusiva de paralelismo inter-consultas no processamento de consultas de alto custo.

A AVP trabalha com pequenos fragmentos virtuais, cujos tamanhos são ajustados dinamicamente durante o processamento de uma consulta. Com isso, ela soluciona os

problemas da SVP relacionados a grandes fragmentos. Na SVP, esses grandes fragmentos podem levar o SGBD a executar buscas lineares na tabela virtualmente fragmentada, o que reduz severamente o seu desempenho, chegando a provocar a obtenção de tempos de execução superiores ao obtido através da estratégia seqüencial. Ao utilizar pequenos fragmentos, a AVP evita as buscas lineares.

Obviamente, surge uma questão: quando um fragmento pode ser considerado "pequeno"? Ou ainda: como calcular o tamanho ideal dos fragmentos, caso ele exista? Uma alternativa para esse problema seria tentar calcular os tamanhos dos fragmentos através de estatísticas presentes nos catálogos dos SGBDs. No entanto, isso tornaria a AVP muito complexa e reduziria a portabilidade do processador de consultas que a utilizasse, pois não há padronização para a implementação de catálogos em SGBDs. Ou, se há, a grande maioria dos SGBDs não a utiliza, uma vez que seus catálogos são bastante diferentes, tanto em relação à organização quanto aos tipos de informações que armazenam.

Por essas razões, optamos pela adoção de uma abordagem totalmente independente de estatísticas para a AVP, baseada em monitoração dos tempos de execução das consultas. A AVP utiliza esses tempos para aumentar ou diminuir dinamicamente os tamanhos dos fragmentos virtuais. Com isso, conseguimos uma solução "leve" e totalmente independente de estatísticas, o que torna nosso processador de consultas portável, podendo ser utilizado com qualquer SGBD sem necessidade de adaptações.

Além de solucionar os problemas relacionados a grandes fragmentos virtuais, a AVP possibilita a implementação de técnicas para balanceamento dinâmico de carga, o que não é possível na SVP, dada a sua abordagem de "uma sub-consulta por nó". Esse balanceamento é importante para a resolução de problemas relacionados à distorção de dados. Nesse sentido, propusemos também nesta tese um algoritmo totalmente descentralizado para redistribuição dinâmica de carga entre os nós do cluster durante o processamento de uma consulta. Ele é baseado em trocas de ofertas de ajuda entre os nós. Nós livres propagam mensagens ofertando seus serviços. Nós ocupados podem então dividir sua carga de trabalho. O algoritmo é independente de um controle central, além de ser independente do SGBD, contribuindo para a portabilidade do processador de consultas.

Para validar nossa solução, implementamos as técnicas propostas em um protótipo desenvolvido em linguagem Java e realizamos experimentos em um cluster com 64 nós, com arquitetura de memória distribuída e replicação total do banco de dados. O SGBD utilizado foi o PostgreSQL. Utilizamos consultas do *benchmark* TPC-H e a base original do *benchmark*, que não apresenta distorção de dados. Para avaliar o algoritmo de redistribuição dinâmica de carga, simulamos distorção de dados nessa base utilizando distribuição *zipf*.

Os resultados dos experimentos mostraram que nossas técnicas obtêm aceleração linear e, muitas vezes, superlinear em várias das situações avaliadas. Com distorção simulada, obtivemos melhor desempenho do que a SVP em todos os casos, exceto em duas consultas utilizando uma configuração do cluster com apenas dois nós. Os resultados mostraram ainda que a sobrecarga causada pelo algoritmo de redistribuição é desprezível.

Analisando os planos de execução gerados pelo PostgreSQL, constatamos que as sub-consultas geradas pela AVP não são influenciadas pelo número de nós utilizados durante o processamento de uma consulta. Concluímos então que ela pode prover ganhos de desempenho tanto em pequenos clusters (com poucos nós) como naqueles com configurações maiores. Por outro lado, os planos de execução gerados para as sub-consultas da SVP variam bastante de acordo com o número de nós utilizados, o que a torna menos robusta.

Realizamos ainda experimentos no sentido de comparar a utilização exclusiva de paralelismo inter-consultas com a opção de combiná-lo com paralelismo intra-consulta no processamento concorrente de consultas de alto custo. Utilizamos cenários típicos para aplicações OLAP e cenários com um número maior de consultas concorrentes em bases de dados com distribuição uniforme e com distorção de dados. No caso de distribuição de dados uniforme, aumentos super-lineares de vazão foram obtidos em várias situações pela estratégia que combina os dois tipos de paralelismo. Na presença de distorção de dados extrema, o aumento da vazão também foi significativo. Concluímos que a estratégia que melhor utiliza os recursos de processamento do cluster é a que combina paralelismo inter-/intra-consulta, sendo também a que proporciona maior agilidade nos processos de análise de dados.

Como desdobramentos desta tese, destacamos três dissertações de mestrado, duas em andamento e uma com início próximo. A primeira, investiga a incorporação da AVP

ao C-JDBC para que este também disponha de paralelismo intra-consulta, ao mesmo tempo em que oferece configurações de arquitetura voltada para OLTP. A segunda, dentro do projeto ClusterMiner (CLUSTERMINER, 2004), investiga a utilização da AVP na implementação de algoritmos de mineração de dados, especialmente com regras de associação. A terceira tese investigará a utilização de AVP com replicação parcial de dados. Destacamos ainda as contribuições em relatórios técnicos ao projeto ClusterMiner (CT-Info, CNPq) e ao projeto de colaboração internacional Brasil-França, DAAD (Capes/Cofecub), dois trabalhos publicados em conferências científicas (LIMA *et al.* (2004a) e LIMA *et al.* (2004b)) e a elaboração de um novo trabalho a ser submetido para publicação em periódico internacional, atualmente em fase de conclusão.

As principais limitações de nossa solução são a replicação total, o não tratamento de atualizações de dados e o grande número de mensagens trocadas pelos nós para implementação da redistribuição dinâmica de carga. A utilização de replicação parcial ou mesmo de fragmentação completa da base de dados exige adaptações no algoritmo de redistribuição dinâmica de carga, uma vez que os nós não terão cópias locais de todos os dados. A redistribuição pode incluir trocas de dados entre os nós. Em relação ao tratamento de atualizações, alternativas para sincronização de réplicas e seu impacto junto à AVP devem ser investigadas. Em relação ao elevado número de mensagens para redistribuição de carga, estamos estudando variações do algoritmo que evitem a repetição da difusão da oferta de ajuda pelos nós livres.

Como trabalhos futuros, pretendemos investigar a utilização da AVP em arquiteturas de disco compartilhado. Esperamos obter resultados semelhantes aos já conseguidos. Pretendemos também colaborar com as teses de mestrado supracitadas, especialmente com a que investigará a utilização de replicação parcial do banco de dados. Por fim, faz-se também necessária uma solução mais robusta para a composição de resultados, problema propositalmente não tratado aqui dada a sua complexidade.

Referências Bibliográficas

AKAL, F., BÖHM, K., AND SCHEK, H.-J., 2002, "OLAP Query Evaluation in a Database Cluster: a Performance Study on Intra-Query Parallelism", In: *Proceedings of the East European Conf. on Advances in Databases and Information Systems (ADBIS), 6th European East Conference*, pp. 218-231, Bratislava, Slovakia, Sep.

BAR-NOY, A., PELEG, D., 1991, "Square Meshes are not always Optimal", *IEEE Transactions on Computers*, v. 40, n. 2, pp. 196-204.

CECCHET, E., MARGUERITE, J., AND ZWAENEPOEL, W., 2004, "C-JDBC: Flexible Database Clustering Middleware", *USENIX Annual Technical Conference 2004, Freenix track*, June.

CLUSGRES, 2004, url: http://www.linuxlabs.com/clusgres.html, consultada em novembro de 2004.

CLUSTERMINER, 2004, url: http://clusterminer.nacad.ufrj.br, consultada em dezembro de 2004.

DB2, 2004, url: http://ibm.com/db2/linux/ice/, consultada em dezembro de 2004.

GANÇARSKI, S. *et al.*, 2002a, "Parallel Processing with Autonomous Databases in a Cluster System", In: *International Conference on Cooperative Information Systems*, pp. 410-428, Irvine, USA, Oct.

GANÇARSKI, S., NAACKE, H., VALDURIEZ, P., 2002b, "Load Balancing of Autonomous Applications and Databases in a Cluster System." In: *4th Workshop on Distributed Data and Structure (WDAS)*, Paris.

GORLA, N., 2003, "Features to Consider in a Data Warehousing System", *Communications of the ACM*, v. 46, n. 11 (Nov.), pp. 111-115.

GRAEFE, G., 1993, "Query Evaluation Techniques for Large Databases", In: *ACM Computing Surveys*, v. 25, n. 2 (June), pp. 73 - 170.

KNUTH, D., 1973, *The Art of Computer Programming – Vol. 3 – Searching and Sorting*, Addison-Wesley.

LEGNET, 2001, url: http://www.telecom.gouv.fr/rntl/AAP2001/Fiches_Resume/LEG@NET.htm, consultada em novembro de 2004.

LIMA, A. A. B., MATTOSO, M., VALDURIEZ, P., 2004a, "OLAP Query Processing in a Database Cluster", In: *Proceedings of the 10th Euro-Par Conference*, Pisa, Italy, Aug.

LIMA, A. A. B., MATTOSO, M., VALDURIEZ, P., 2004b, "Adaptive Virtual Partitioning for OLAP Query Processing in a Database Cluster", In: *Proceedings of the 19th Brazilian Symposium on Databases*, Brasília, Brazil, Oct.

MÄRTENS, H., 1999, "A Classification of Skew Effects in Parallel Database Systems" In: *Proceedings of the 5th Euro-Par Conference*, Toulouse, France, Aug.

MYSQL, 2004a, "A Guide to High Availability Clustering – How MySQL Supports 99.999% Availability", *MySQL Business White Paper*, url: http://www.mysql.com/products/cluster (consultada em novembro de 2004), June.

MYSQL, 2004b, MySQL DBMS, url: http://www.mysql.com, consultada em novembro de 2004.

ORACLE, 2003, "Oracle RAC 10g Overview", *Oracle White Paper*, November.

ORACLE, 2004, Oracle DBMS, url: http://www.oracle.com, consultada em novembro de 2004.

ÖSZU, T., AND VALDURIEZ, P., 1999, *Principles of Distributed Database Systems*, 2 ed., New Jersey, Prentice-Hall.

PGCLUSTER, 2004, "PG-Cluster: the multi-master synchronous replication system for PostgresSQL", url: http:// http://www.csra.co.jp/~mitani/jpug/pgcluster/en/index.html, consultada em novembro de 2004.

POSTGRESQL, 2004, PostgreSQL DBMS, url: http://www.postgresql.org, consultada em novembro de 2004.

POWERDB, 2004, "The Project PowerDB", url: http://www.dbs.ethz.ch/~powerdb, acessada em novembro de 2004.

RÖHM, U., BÖHM, K., SCHEK, H., 2000, "OLAP Query Routing and Physical Design in a Database Cluster", In: *Proceedings of the 7th Int. Conf. On Extending Database Technology*, March.

RÖHM, U., BÖHM, K., SCHEK, H.-J., 2001, "Cache-Aware Query Routing in a Cluster of Databases", *In: Proceedings of the 17th International Conference on Data Engineering (ICDE)*, pp. 641-650, Heidelberg, Germany, April 2-6.

RÖHM, U., BÖHM, K., SCHEK, H.-J., *et al.*, 2002, "FAS - A Freshness-Sensitive Coordination Middleware for a Cluster of OLAP Components", In: *Proceedings*

of the 28th International Conference on Very Large Databases (VLDB), pp. 754-765, Hong Kong, China.

STERLING, T., 2001, "An Introduction to PC Clusters for High Performance Computing", *The International Journal of High Performance Computing Applications*, v. 15, n. 2, pp. 92-101.

STÖHR, T., MÄRTENS, H, RAHM, E., 2000, "Multi-Dimensional Database Allocation for Parallel Data Warehouses", In: *Proceedings of the 26th International Conference on Very Large Databases (VLDB)*, pp. 273-284, Cairo, Egypt.

TOP500, 2004, *TOP500 Supercomputer Sites*, url: http://www.top500.org/, acessada em novembro de 2004.

TPC, 2003a, "TPC Benchmark™ H – Revision 2.1.0", url: http://www.tpc.org, acessada em novembro de 2004.

TPC, 2003b, "TPC Benchmark™ R – Revision 2.1.0", url: http://www.tpc.org, acessada em novembro de 2004.

TPC, 2004, *Transaction Processing Performance Council*, url: http://www.tpc.org, acessada em novembro de 2004.

VALDURIEZ, P., 1993, "Parallel Database Systems: Open Problems and New Issues", *International Journal on Distributed and Parallel Databases*, v. 1, n. 2, pp. 137–165.

WALTON, C. B., DALE, A. G., JENEVEIN, R. M., 1991, "A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins", In: *Proceedings of the 17th International Conference on Very Large Databases (VLDB)*, pp. 537–548, Barcelona, Spain.

WHITE, S., FISHER, M., CATTEL, R., *et al.*, 2001, *JDBC API Tutorial and Reference*, 2 ed., Addison-Wesley.

WOJCIECHOWSKA, I., 1999, *Broadcasting in Grid Graphs*, Ph.D. Dissertation, West Virginia University - College of Engineering and Mineral Resources, West Virginia, USA.

WOLF, J. L., YU, P. S., TUREK, J., DIAS, D. M., 1993, "A Parallel Hash Join Algorithm for Managing Data Skew", *IEEE Transactions on Parallel and Distributed Systems*, v. 4, n. 12, pp. 1355-1371.