APUAMA: PROCESSAMENTO DE CONSULTAS OLAP COM ATUALIZAÇÕES CONCORRENTES EM UM AGRUPAMENTO DE BANCO DE DADOS

Bernardo Faria de Miranda

DISSERTAÇÃO 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 MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

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

Prof. Alexandre de Assis Bento Lima, D.Sc.

Prof. Geraldo Zimbrão da Silva, D.Sc.

Prof. Sérgio Lifschitz, D.Sc.

RIO DE JANEIRO, RJ - BRASIL

MARÇO DE 2006

MIRANDA, BERNARDO

Apuama: Processamento de Consultas OLAP com Atualizações Concorrentes em um Agrupamento de Banco de Dados [Rio de Janeiro] 2006

VII, 100 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2006)

Dissertação - Universidade Federal do Rio de Janeiro, COPPE

1. Banco de Dados

2. Paralelismo

3. Consultas OLAP

I. COPPE/UFRJ II. Título (série)

AGRADECIMENTOS

Agradeço à minha orientadora, Profª. Marta Mattoso, por toda sua dedição durante esta longa jornada. Por mostrar os desafios da computação sob sua perspectiva, fazendo com que me motivasse profundamente para esta pesquisa. Por estar sempre pronta a receber meus questionamentos e dúvidas. Esta dissertação não seria possível sem sua participação. Sua visão empreendedora tornou o projeto ParGRES possível. Este projeto permitiu que durante um ano estivesse inteiramente envolvido na concepção, no desenvolvimento e na distribuição de um produto inovador da aérea de Banco de Dados. Não mediu esforços para viabilizar visitas a congressos e eventos.

Agradeço ao meu orientador, Prof. Alexandre Lima, por ter mostrado durante a sua tese de doutorado resultados que inspiraram o início desta pesquisa. Pela sua paciência em discutir o desenvolvimento do Apuama em todas suas etapas. Por enriquecer este trabalho com sua experiência e sabedoria.

Agradeço aos meus pais, Ronaldo e Regina, por estarem sempre ao meu lado e iluminarem meu caminho. São pessoas incríveis que não se cansam de mostrar que tudo é possível desde que tenhamos fé e amor em nossas vidas. Agradeço ao meu irmão, Guilherme, por todo seu companheirismo e carinho.

Agradeço à minha namorada, Carolina, por ser meu porto seguro. Em períodos de intenso trabalho, apenas conseguia recuperar minhas forças com sua presença e carinho. Acompanhava de perto cada detalhe de meu trabalho e me motivava a seguir com estusiasmo à próxima etapa.

Agradeço a toda minha família por compartilhar e brindar esta fase de minha vida. Agradeço à minha madrinha, Glorinha, por seus extensos conselhos ao telefone. Agradeço à minha tia Márcia por seu abraço emocionado após a apresentação da dissertação. Agradeço aos meus avós, Aïxa e Flávio, por mostrarem a importância de relembrar o passado e perceber que só chegamos até onde estamos porque somos cientes da responsabilidade que carregamos.

Agradeço aos meus amigos da COPPE e colegas de trabalho por compartilharem os desafios e as incertezas que passei durante o mestrado.

Agradeço à CAPES, ao Programa de Engenharia de Sistemas e Computação, ao Núcleo de Computação de Alto Desempenho de COPPE/UFRJ e ao grupo Paris de INRIA pelos recursos disponibilizados para que esta pesquisa fosse possível.

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

APUAMA: PROCESSAMENTO DE CONSULTAS OLAP COM ATUALIZAÇÕES CONCORRENTES EM UM AGRUPAMENTO DE BANCO DE DADOS

Bernardo Faria de Miranda

Março/2006

Orientadores: Marta Lima de Queirós Mattoso
Alexandre de Assis Bento Lima

Programa: Engenharia de Sistemas e Computação

O agrupamento de banco de dados (DBC) é constituído por diversos sistemas de gerenciamento de banco de dados (SGBD) seqüenciais instalados em nós de um agrupamento de PCs. Os SGBDs são controlados por uma camada de software responsável por prover o paralelismo de consultas. O DBC fornece uma solução de baixo custo e de alto desempenho para processamento de consultas. Usando tanto o paralelismo inter-consulta quanto o intra-consulta sobre bases de dados replicadas, o DBC pode acelerar individualmente o processamento de consultas e aumentar a vazão do sistema. No entanto, não existe DBC que combine o paralelismo inter e intra-consulta enquanto trata transações de atualização. O C-JDBC é um DBC de sucesso que oferece o paralelismo inter-consulta e controla consistência entre as réplicas da base de dados, mas não pode acelerar o tempo de execução de consultas pesadas, típicas de ambientes OLAP. Nesta dissertação, propomos o Apuama, que adiciona o paralelismo intra-consulta ao C-JDBC. O resultado é uma solução em software livre que comporta tanto aplicações OLAP quanto de OLTP. O Apuama foi avaliado em um agrupamento de 32 PCs executando consultas OLAP do *benchmark* TPC-H usando o PostgreSQL como SGBD. Nossos testes mostram que o Apuama proporciona aceleração de consultas e escalabilidade de vazão superlinear em ambientes apenas de leitura. Além disso, apresenta desempenho com ganhos mesmo sob operações de atualização de dados.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

APUAMA: OLAP QUERY PROCESSING WITH CONCURRENT UPDATES IN A DATABASE CLUSTER

Bernardo Faria de Miranda

March/2006

Advisors: Marta Lima de Queirós Mattoso
Alexandre de Assis Bento Lima

Department: Systems and Computing Engineering

Database clusters (DBCs) consist of many sequential database management systems (DBMSs) installed on a PC cluster. The DBMSs are controlled by a middleware responsible for providing query parallelism. DBCs provide a low cost solution for high performance query processing. By using either inter- or intra-query parallelism on replicated data, they can accelerate the query processing individually and increase system throughput on DBMS. However, there is no DBC that combines inter- and intra-query parallelism while supporting update transactions. C-JDBC is a successful DBC that offers inter-query parallelism and controls database replica consistency but cannot accelerate individual heavy-weight queries, typical of OLAP environments. In this dissertation, we propose the Apuama, which adds intra-query parallelism to C-JDBC. The result is an open-source solution that supports both OLAP and OLTP applications. We evaluate Apuama on a 32-node PC cluster running OLAP queries of the TPC-H benchmark using PostgreSQL as DBMS. Our tests show that the Apuama yields super-linear speedup and scale-up throughput with read-only environments. Furthermore, it yields good performance under data update operations.

Índice

1 Introdução

Um *data warehouse* (DW) é um repositório de dados projetado para apoiar o processo de tomada de decisão nos negócios de uma empresa (INMON, 2002). O repositório acumula o histórico de dados gerado ao longo do dia a dia operacional de uma empresa. Por meio de consultas feitas pelo analista de negócio ao DW, é possível mensurar indicadores de desempenho, identificar riscos e conhecer melhor o perfil dos seus clientes. A descoberta destas informações é importante para o bom posicionamento da empresa em um mercado competitivo (GORLA, 2003). A eficiência do processamento da análise de dados é crucial para que o DW forneça informações relevantes em tempo hábil. Quanto maior a competitividade, mais cresce a necessidade de que as informações fornecidas pelo sistema de suporte à decisão sejam atualizadas freqüentemente. A rapidez da geração de um novo relatório, que, por exemplo, relacione os produtos mais vendidos para uma determinada classe de clientes, pode ser decisiva para lançar uma nova promoção de vendas antes do concorrente. Portanto, as empresas dispostas a ter o auxílio do DW para extrair eficientemente informações de seus dados históricos devem investir no poder computacional da sua infra-estrutura.

A infra-estrutura de DW precisa ser robusta para acompanhar o ritmo de crescimento das bases de dados. Segundo relatório da *Winter Corporation* (AUERBACH *et al.*, 2005), o volume de dados em DW nas grandes corporações dobra em média a cada dois anos. A busca por informações no DW é baseada em consultas OLAP (*On-Line Analytical Processing*) processadas sobre sistemas de gerenciamento de banco de dados (SGBDs) responsáveis por acumular largo histórico de dados (CHAUDHURI, DAYAL, 1997). Como as consultas OLAP têm natureza *ad-hoc* (GORLA, 2003), não é simples fazer sintonia fina no banco de dados com base no histórico de consultas freqüentes. Por estas razões, estudos são feitos há muito tempo no sentido de reduzir o tempo de processamento dessas consultas. A exemplo de outras áreas de pesquisa, agrupamentos de PCs (*cluster of PCs*) têm sido usados com sucesso para atingir altos índices de desempenho para aplicações de banco de dados (PACITTI *et al.*). Agrupamentos de PCs podem escalar para grandes configurações e oferecem ótima relação custo/desempenho (GANÇARSKI *et al.*, 2002). Seguindo esta tendência, fornecedores tradicionais de SGBDs oferecem versões paralelas do seu software para processamento de consultas em agrupamento de PCs. O *Wal-Mart*, maior varejista

mundial, é cliente desta tecnologia e usa um SGBD paralelo sobre um enorme agrupamento de PCs para analisar mais de 500 *terabytes* de dados gerados por seus pontos de vendas (SCHUMAN, 2004) .

O uso de SGBDs paralelos, no entanto, implica em custos que podem inibir seu uso por muitas aplicações. Os custos associados à migração da base de dados do SGBD seqüencial para um novo SGBD paralelo são altos. Muitas vezes o uso desta versão paralela do SGBD implica em adicionar itens de *hardware* específicos, como por exemplo, novas placas de rede. Além disso, em geral o modelo de vendas de SGBDs paralelos obriga a aquisição de uma nova licença para cada nova cópia do software em todo nó do agrupamento. Todos estes fatores, se somados, resultam em custos que - muitas vezes - superam de longe o valor total dos PCs do agrupamento.

Recentemente, como alternativa aos SGBDs paralelos, tem sido proposto o uso do agrupamento de banco de dados (DBC - *Database Cluster*) (AKAL *et al.*, 2002; CECCHET, 2004a; COULON *et al.*, 2004; LIMA *et al.*, 2004a; MATTOSO *et al.*, 2005). O DBC apresenta uma solução de baixo custo para aumento de desempenho de consultas com base no paralelismo do agrupamento de PCs. Um DBC consiste em um conjunto de SGBDs seqüenciais independentes (não preparados para o ambiente paralelo) distribuídos ao longo de um conjunto de nós do agrupamento. Estes SGBDs são orquestrados por uma camada de software (*middleware*) responsável por oferecer uma visão externa única de todo o sistema, como um SGBD virtual. A aplicação cliente não precisa ser modificada quando o servidor de banco de dados é substituído por um DBC. Suas consultas são enviadas para a camada que provê transparência de distribuição de dados e gerencia o processamento paralelo de consultas. O DBC pode ser comparado ao conceito de RAID (*Redundant Array of Independent Disks*), em que dois ou mais discos são combinados por uma placa controladora para prover maior desempenho e disponibilidade (CECCHET, 2004b). Assim como os discos, cada SGBD funciona independentemente, não ciente da existência de outros SGBDs. O código-fonte do SGBD não precisa ser alterado para que funcione em um DBC. Desta forma, a grande vantagem do DBC está em proporcionar uma solução de baixo custo e não intrusiva para migração de uma aplicação baseada em um SGBD seqüencial para um ambiente de paralelismo.

Dois tipos de paralelismo podem ser explorados no processamento de consultas em um DBC: paralelismo inter-consulta e paralelismo intra-consulta. O paralelismo

inter-consulta consiste em executar várias consultas simultaneamente, distribuídas entre os nós do agrupamento de PCs. Em sua forma mais simples, o processamento de cada consulta é atribuído inteiramente a um único nó. O paralelismo inter-consulta é bem empregado em aplicações que visam aumentar a vazão de um sistema ao processar várias consultas simultaneamente. Essas aplicações, como, por exemplo, as aplicações OLTP (*On-Line Transactional Processing*), geralmente possuem consultas de baixo custo de execução, mas que ocorrem em número elevado. Por outro lado, as aplicações OLAP têm tipicamente consultas de alto custo de execução, ou seja, que acessam uma grande porção dos dados e desempenham operações complexas, como ordenação, agregação e junção. As consultas OLAP levam um longo tempo para serem processadas. Soluções como a do C-JDBC (CECCHET, 2004a), que provêem o uso isolado do paralelismo inter-consulta, não são apropriadas para o processamento de consultas pesadas porque não reduzem o tempo de processamento individual das consultas. Neste caso, o paralelismo intra-consulta é a solução mais adequada, como é mostrado em LIMA *et al.* (2004b).

O paralelismo intra-consulta consiste em envolver muitos nós no processamento de uma única consulta, reduzindo assim o tempo total de execução da consulta individual. Cada nó é responsável pelo processamento de apenas um subconjunto de dados. O objetivo principal é reduzir o tempo de execução individual da consulta pesada e, ao mesmo tempo, melhorar a vazão geral do sistema.

Os paralelismos inter e intra-consulta podem ser combinados em uma implementação de DBC. Além do mais, um DBC com os dois tipos de paralelismo e permitindo atualização concorrente dos dados pode ser usado tanto em aplicações OLAP quanto OLTP. Entretanto, as soluções existentes em DBC oferecem soluções de paralelismo voltadas exclusivamente a OLAP ou OLTP (AKAL *et al.*, 2002; CECCHET, 2004a; COULON *et al.*, 2004; LIMA *et al.*, 2004a; MATTOSO *et al.*, 2005). Em primeiro lugar, atualmente não existe este tipo solução porque os DBCs que atendem a aplicações OLAP não oferecem suporte a transações de atualização. Como tais soluções não permitem atualizações concorrentes, o processo de renovação dos dados das réplicas é manual e forçosamente ocorre apenas em horários pré-determinados de ociosidade do DW para não ocasionar resultados inconsistentes em consultas OLAP. A segunda razão é que a combinação paralelismo inter e intra-consulta pode ser conflitante. O paralelismo intra-consulta necessita que os dados estejam

fragmentados fisicamente e distribuídos entre os nós do agrupamento. Quando os dados são fisicamente fragmentados entre os nós do agrupamento, o processamento de paralelismo inter-consulta torna-se bastante limitado, pois a maioria das consultas precisa varrer todos os fragmentos em paralelo. Dependendo do projeto de fragmentação de dados, uma simples consulta OLTP pode ser processada apenas por paralelismo intra-consulta e tornar-se bastante ineficiente. Por outro lado, consultas OLAP sem fragmentação de dados podem não ser executadas eficientemente.

Nosso objetivo é, desta forma, prover uma solução de DBC de alto desempenho e de baixo custo para processamento concorrente de consultas OLAP e transações de atualização de dados. Para evitar problemas com a fragmentação física da base, adotamos a **fragmentação virtual** (*VP – virtual partitioning*) (AKAL *et al.*, 2002) para uma base de dados totalmente replicada. Nossa proposta tem como base o C-JDBC. O C-JDBC é um DBC em código livre de qualidade industrial que oferece suporte a paralelismo inter-consulta e a controle de consistência de réplicas. O C-JDBC provê excelente desempenho para aplicações OLTP (CECCHET, 2004a), mas não oferece paralelismo intra-consulta. Sendo assim, nós estendemos o C-JDBC de uma forma não intrusiva para oferecer uma solução com paralelismo intra-consulta.

Nesta dissertação, apresentamos o Apuama[1] como uma extensão do C-JDBC. O objetivo principal é prover um ambiente de processamento de consultas OLAP usando o paralelismo intra-consulta e mantendo a eficiência de suporte do C-JDBC a transações OLTP. Nenhum código-fonte do C-JDBC foi modificado. O Apuama age como uma conexão representante (*proxy*) entre o C-JDBC e os SGBDs. O Apuama não interfere no processamento de consulta do C-JDBC, sendo usado apenas para processamento de consultas OLAP. Ao contrário de outras soluções que usam paralelismo intra-consulta para aplicações OLAP, o Apuama provê o controle de consistência entre réplicas durante o processamento de paralelismo intra-consulta. O desenvolvimento desta dissertação é um desdobramento dos resultados bem sucedidos no paralelismo intra-consulta proposto por LIMA (2004).

Para avaliar a implementação do Apuama, executamos experimentos baseados no *benchmark* TPC-H (TPC-H, 2005) (específico para aplicações OLAP) em um agrupamento de 32 nós usando PostgreSQL 8.0.1 (POSTGRESQL, 2005). A aceleração

---

[1] *Apuama* significa *veloz* em Tupi-Guarani.

do processamento da consulta e a escalabilidade de vazão foram mensuradas em experimentos com consultas apenas de leitura e uma combinação de consultas apenas de leitura com transações de atualização de dados concorrentes. Na maioria dos casos, Apuama atinge aceleração e escalabilidade de vazão superlineares. Como não houve nenhuma alteração no processamento paralelo inter-consulta do C-JDBC, consideramos que seus resultados anteriores para aplicações OLTP (CECCHET *et al.*, 2004) obtidos durante seu desenvolvimento foram mantidos.

Esta dissertação é organizada da seguinte forma. No capítulo II apresentamos os conceitos relacionados a DBC, suas alternativas para processamento de consulta e, por fim, os trabalhos relacionados são analisados. A nossa estratégia de processamento de consultas OLAP para DBC é mostrada no capítulo III. No capítulo IV explicamos como o C-JDBC foi estendido e apresentamos a arquitetura interna do Apuama. Os resultados experimentais que validam nossa proposta são apresentados no capítulo V. O capítulo VI conclui a dissertação apresentando as suas contribuições e as perspectivas de trabalhos futuros.

II Agrupamentos de Bancos de Dados

Apesar de ser um conceito recente, o DBC usa técnicas de paralelismo já consagradas por SGBDs paralelos. No entanto, diferentemente de um SGBD paralelo, o DBC é composto por SGBDs não-paralelos, sem qualquer modificação intrusiva no funcionamento destes. As técnicas de paralelismo são empregadas externamente ao SGBD não-paralelo por meio de uma camada de software. Na seção II.1 descrevemos os conceitos relacionados a DBC. Na seção II.2 fazemos análise das alternativas de projeto de distribuição de dados e suas aplicações para DBC. Na seção II.3 discutimos as técnicas de processamento paralelo de consultas clássicas e suas particularidades quando usadas em DBC. Na seção II.5 discutimos as alternativas de controle de replicação de dados. Os trabalhos relacionados a DBC são analisados na seção II.6. Na seção II.7 comentamos abordagens utilizadas por SGBDs paralelos. A seção II.8 apresenta as conclusões do capítulo.

II.1 Conceitos de Agrupamentos de Bancos de Dados

Há muito o modelo relacional é amplamente adotado no mercado de banco de dados. O SGBD relacional é o componente-chave de uma ampla gama de aplicações, inclusive de sistemas de suporte à decisão. Para estas e muitas outras aplicações, o desempenho da camada de banco de dados é fator crucial de sucesso.

O agrupamento de PCs (*Personal Computers* – computadores pessoais) tem sido largamente utilizado para melhorar o desempenho de aplicações de banco de dados por meio do uso do processamento paralelo. Isto ocorre principalmente porque oferece ótima relação entre custo e desempenho e por ser possível compor soluções com grande número de processadores. A arquitetura do agrupamento de PCs é conhecida como de memória distribuída. Os PCs em agrupamento compartilham apenas a rede que os interconecta. Uma instância do SGBD é instalada em cada nó do agrupamento. O principal benefício do uso de agrupamento de PCs é que, sem a necessidade de *hardware* específico e dispendioso, é possível aumentar linearmente o poder de E/S da solução. A vazão global de E/S é aumentada usando o paralelismo entre os dispositivos de armazenamento. Como não existe um dispositivo de armazenamento central, é necessário que os dados estejam distribuídos entre os PCs. O SGBD paralelo que atua

com este tipo de arquitetura tem um mecanismo de coordenação global para oferecer acesso transparente à base de dados distribuída. A adição de novos nós pode exigir a reorganização dos dados no agrupamento.

No entanto, a quantidade de aplicações que podem usufruir de SGBDs paralelos é limitada. O custo de implantação de um SGBD paralelo pode superar a soma do custo de *hardware* dos PCs do agrupamento. A licença de uso de uma versão paralela destes SGBDs freqüentemente é muito mais dispendiosa do que a sua versão seqüencial. Além disso, o valor da licença varia na mesma proporção que o número de processadores do agrupamento. Os custos envolvidos na migração da base de dados do ambiente seqüencial para o paralelo e a necessidade de *hardware* específico também tornam a solução custosa. Alguns SGBDs paralelos, por exemplo, exigem rede e dispositivos de armazenamento de alto desempenho.

Recentemente foi proposto o conceito de **agrupamento de banco de dados** (DBC - *Database Cluster*) por AKAL *et al.* (2002) como uma alternativa para explorar o paralelismo do agrupamento de PCs e melhorar o desempenho de processamento de consultas. O DBC é um conjunto de SGBDs não-paralelos organizados em um agrupamento de PCs e orquestrados por uma camada de software (*middleware*). Não há qualquer alteração do código-fonte do SGBD não-paralelo. Cada SGBD é usado como um componente "caixa-preta" que não está ciente da existência de outros SGBDs.

A proposta do DBC procura facilitar a migração da aplicação para o ambiente paralelo diminuindo a complexidade de migração e seus custos decorrentes. Ao invés de reimplementar rotinas do SGBD para ambientes paralelos, é mantido o uso de mecanismos já consolidados no próprio SGBD seqüencial, deixando a responsabilidade de visão global para a camada de software. Como pode ser visto na Figura 1, esta camada tem a responsabilidade de orquestrar o conjunto de SGBDs seqüenciais, escondendo toda a complexidade do agrupamento. A aplicação cliente não é capaz de perceber que existe um conjunto de SGBDs disponíveis, pois não se conecta diretamente a eles. A camada age como um representante (*proxy*) entre a aplicação cliente e o conjunto de SGBDs. A aplicação cliente envia todas as suas requisições à camada que, por sua vez, as redistribui entre os SGBDs do agrupamento. A camada implementa o paralelismo sem o uso de informações internas dos SGBDs locais ou até mesmo sem o plano de execução local de uma consulta. Como os SGBDs são usados como componentes "caixa-preta", o emprego da camada é independente do produto ou

versão do SGBD seqüencial utilizado. Basta que o SGBD siga os padrões de conectividade estabelecidos pela indústria, pois a camada apenas interage com sua a interface de comunicação externa.

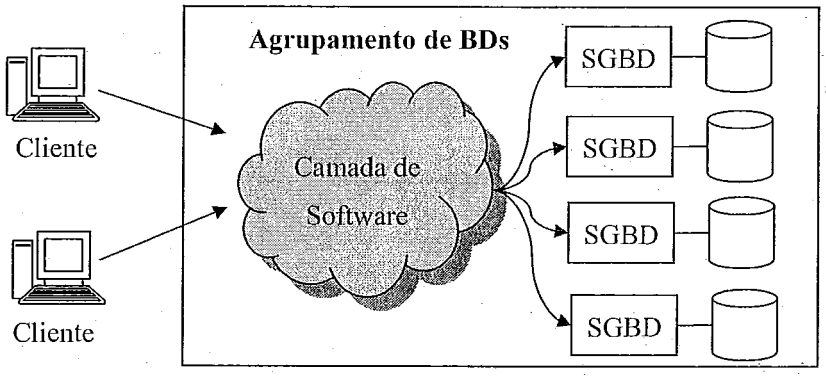

Figura 1 – Agrupamento de Banco de Dados (DBC – *Database Cluster*)

Por usar arquitetura de memória distribuída, a infra-estrutura de *hardware* do DBC pode ser estendida para grandes configurações sem o uso de *hardware* específico, dependendo apenas da adição de novos PCs à rede. Atualmente os pesquisadores tentam descobrir como melhorar o desempenho do DBC analisando alternativas de distribuição dos dados entre os nós do agrupamento e de processamento de consultas sem modificar o código-fonte do SGBD seqüencial usado na solução (AKAL *et al.*, 2002; CECCHET, 2004a; COULON *et al.*, 2004; LIMA *et al.*, 2004a; MATTOSO *et al.*, FURTADO *et al.*, 2005).

II.2 Projeto de Distribuição de Dados para Agrupamento de Banco de Dados

As decisões sobre a disposição de dados pelos nós do ambiente distribuído compõem um processo denominado de **projeto de distribuição de dados** (ÖZSU, VALDURIEZ, 1999). Como o DBC usa arquitetura de memória distribuída e, portanto, possui dispositivos de armazenamento independentes para cada nó, é necessário distribuir os dados entre os nós do agrupamento. O projeto de distribuição pode ser divido em duas etapas: **fragmentação** e **alocação**. As tabelas são decompostas em fragmentos que, por sua vez, são alocados em diferentes nós para atender a objetivos específicos da aplicação.

II.2.1 Projeto de Fragmentação

No modelo relacional, a etapa de fragmentação de um projeto de distirbuição tem por objetivo dividir as tabelas do esquema conceitual global de modo a diminuir o volume de dados irrelevantes acessados pelas consultas. O projeto de fragmentação busca encontrar a unidade apropriada de distribuição. A fragmentação de tabelas em um projeto de distribuição pode ser feita horizontalmente ou verticalmente.

A **fragmentação horizontal** decompõe uma tabela em subconjuntos de tuplas. Cada fragmento tem um subconjunto de tuplas com os mesmos atributos da tabela original. A fragmentação horizontal pode beneficiar operações de seleção e de junção. Para a junção é importante identificar tabelas relacionadas pela equi-junção, onde uma das tabelas é a dona do relacionamento (por exemplo, a que contém a chave primária) e a outra é membro do relacionamento (por exemplo, a que contém a chave estrangeira). Há dois tipos de fragmentação horizontal: fragmentação horizontal primária e fragmentação horizontal derivada. A **fragmentação horizontal primária** decompõe a tabela segundo uma função de fragmentação. Os tipos mais comuns dessa função de fragmentação horizontal são: circular, faixa de valores e dispersão (*hash*). A fragmentação por faixa de valores, por exemplo, pode ser especificada através de um predicado de seleção. O fragmento horizontal é o resultado da aplicação de uma operação de seleção, com o predicado contendo a faixa de valores, sobre a tabela a ser fragmentada. O predicado dessa seleção irá definir a faixa de valores dos atributos das tuplas do fragmento. Um predicado simples tem a forma:

`Atrib Φ valor`

"Atrib" especifica o atributo da tabela a ser usado como atributo de fragmentação, "Φ" especifica o operador booleano a ser aplicado ao atributo, podendo pertencer a $\{=, >, <, \neq, \geq, \leq\}$ e "valor" especifica um valor do domínio do atributo. A faixa de valores pode ser definida através da conjunção de dois predicados simples. Supondo uma tabela R, os fragmentos R_i são formalmente definidos da seguinte forma:

$$R_i = \sigma_{Atrib \geq valorInf_i \wedge Atrib < valorSup_i} (R), \quad 1 \leq i \leq n$$

O valor de "n" equivale ao número de fragmentos gerados, "valorInf$_i$" define o limite inferior do atributo de fragmentação que define o fragmento R_i e "valorSup$_i$"

define o valor superior. A união dos fragmentos horizontais corresponde exatamente à tabela original:

$$R = \bigcup R_i, \quad 1 \leq i \leq n$$

A **fragmentação horizontal derivada** visa otimizar a execução de operações de equi-junção. Assim, a fragmentação derivada é definida sobre uma tabela que é membro de um relacionamento equi-junção. A função de fragmentação decompõe a tabela membro de acordo com o atributo de relacionamento equi-junção com a tabela proprietária. Dadas duas tabelas relacionadas R e S, sendo S proprietária e R membro, os fragmentos R_i de R podem ser obtidos da seguinte forma:

$$R_i = R \ltimes S_i, \quad 1 \leq i \leq n$$

S_i corresponde a um fragmento de S e "n" informa o número de fragmentos da tabela S. A quantidade de fragmentos de S e R é a mesma.

A **fragmentação vertical** divide a tabela em fragmentos cujas tuplas possuem apenas um subconjunto dos atributos da tabela original. A cardinalidade da tabela fragmentada é a mesma da tabela original. Os fragmentos mantêm em comum os identificadores das tuplas para que seja possível reconstruir a tabela original usando uma operação de junção entre os fragmentos. O fragmento vertical R_i da tabela R pode ser definido formalmente como:

$$R_i = \Pi_{A_i}(R), \quad 1 \leq i \leq n$$

"A_i" corresponde a um subcojunto de atributos de R e "n" é o número de fragmentos verticais. A relação original R pode ser reconstruída através de seus fragmentos por meio da junção destes.

$$R = \bowtie_k R_i, \quad 1 \leq i \leq n$$

"k" corresponde à chave primaria de R.

As fragmentações horizontal e vertical podem ser ainda combinadas em uma fragmentação híbrida para geração de novos fragmentos.

II.2.1.1 Projeto de Fragmentação Virtual

O DBC usa os SGBDs como componentes caixa-preta e não pode interferir nos seus módulos internos para efetuar corretamente o processamento de consultas sobre uma base de dados fragmentada. A não ciência dos fragmentos pelo SGBD pode gerar complexidades para o DBC. Uma forma de utilizar fragmentação sem decompor fisicamente a tabela é por meio da **fragmentação virtual** (VP - *virtual partitioning*) (AKAL *et al.*, 2002). Ao contrário da fragmentação física, a VP define dinamicamente os fragmentos de uma tabela. O princípio básico da VP é definir os fragmentos virtuais com base na alteração de uma consulta pouco antes de ser executada. No caso de uma fragmentação horizontal, um fragmento horizontal virtual pode ser especificado através da definição dinâmica de um predicado de seleção envolvendo o atributo de fragmentação da tabela que se deseja fragmentar virtualmente.

Para simplificar o projeto de fragmentação virtual e prover máxima flexibilidade na alocação de nós durante a execução da consulta, as tabelas são totalmente replicadas em todos os nós do agrupamento. O objetivo da VP é fazer com que a consulta seja executada sobre um diferente subconjunto de tuplas. Vamos considerar o exemplo da consulta *C*:

```
C: select sum(l_extendedprice) from lineitem
   where l_quantity < 30
```

Para usar os fragmentos de acordo com VP, *C* seria reecrista da seguinte forma:

```
Ci: select  sum(l_extendedprice) from lineitem
    where   l_quantity < 30
    and     l_orderkey >= :v1 and l_orderkey < :v2
```

A diferença entre *C* e C_i é o predicado de seleção definido por faixa de valores sobre o atributo l_orderkey, conforme "l_orderkey >= *:v1* and l_orderkey < *:v2*". Chamamos de **atributo de fragmentação virtual** (*VPA - virtual partitioning atributte*) o atributo escolhido para fragmentar virtualmente a tabela.

Uma solução interessante para implementar o conceito de VP horizontal foi proposta por AKAL *et al.* (2002). Nesta dissertação, esta proposta é referenciada como **fragmentação virtual simples** (*SVP – Simple Virtual Partitioning*). Na SVP, o número de fragmentos virtuais é igual ao número de nós que irão executar a consulta. Seguindo o exemplo anterior, vamos considerar que a consulta *C* é executada em quatro nós de um DBC. A partir desta consulta *C* são criadas as subconsultas C_1, C_2, C_3, C_4. Os valores

usados para os parâmetros *v1* e *v2* variam de nó para nó e são computados de acordo com o intervalo total do domínio do VPA. Supondo que o intervalo de valores de l_orderkey é [1; 6,000,000], os intervalos cobertos por cada subconsulta seriam os seguintes: C_1: *v1*=1 e *v2*=1.500.001; C_2: *v1*=1.500,001 e *v2*=3.000,001; e assim por diante. Apesar de cada nó ter a mesma cópia de *lineitem*, a SVP força cada um a processar apenas um diferente subconjunto de tuplas de *lineitem*. Adicionalmente, em um cenário de replicação total da base de dados, é possível alocar o processamento de qualquer subconsulta em qualquer nó.

No entanto, esta abordagem não nos garante que a subconsulta submetida a um nó irá apenas varrer o subconjunto de dados de *lineitem* a ela correspondente. Para a SVP ser efetiva, as tuplas do fragmento virtual têm de ser agrupadas fisicamente em disco de acordo com o VPA e deve existir um índice associado a este atributo, ou seja, deve existir um índice agrupado em *lineitem* sobre o VPA. O **índice agrupado** é um índice que define a organização física da tabela de acordo com a ordem lógica do atributo a que se refere. Desta forma, as tuplas de um fragmemento virtual vão estar agrupadas no número menor possível de blocos. Com exceção do primeiro e do último bloco, todos os outros blocos deste fragmento virtual contêm extritamente as suas tuplas correspondentes. O primeiro e o último bloco são fronteira com outros fragmentos virtuais.Adicionalmente, o SGBD deve escolher o índice agrupado para ser usado no plano de execução.

Em recente publicação (FURTADO *et al.*, 2005), uma alternativa para projeto de distribuição foi proposta usando uma combinação de fragmentação física e virtual da base de dados. O objetivo é combinar a flexibilidade para definição de fragmentos da VP e o melhor uso do espaço em disco por meio da fragmentação física.

II.2.2 Projeto de Alocação

A etapa de alocação em um projeto de distribuição determina onde cada um dos fragmentos é alocado. Um mesmo fragmento pode ser alocado em um ou mais nós. O projeto de alocação pode ser dividido em três abordagens: replicação total, replicação parcial e sem replicação. Na abordagem de **replicação total**, cada fragmento é replicado em todos os nós. O DBC tem a disponibilidade maximizada. Isto significa que cada nó pode responder independentemente a qualquer requisição de leitura do banco de dados. A falha em um SGBD não compromete o funcionamento do DBC. A desvantagem desta

abordagem está no custo de manutenção da consistência dos fragmentos replicados. Adicionamente, o tamanho da base de dados fica limitado pelo nó com a menor capacidade de armazenamento.

No outro caso extremo, em que nenhum fragmento é replicado, existe a abordagem **sem replicação.** Há apenas uma cópia de cada fragmento no agrupamento de PCs. Se um SGBD do agrupamento sofrer uma falha, a base de dados fica incompleta. Nesta abordagem, como não há réplicas, não há custo adicional para atualizar um fragmento.

Na abordagem de **replicação parcial**, alguns fragmentos são alocados em uma quantidade parcial dos nós do agrupamento. Muitas vezes esta abordagem é adotada para diminuir as conseqüências negativas do uso da replicação total, limitando a replicação das tabelas mais frequentemente atualizadas a poucos nós.

A replicação dos fragmentos cria a necessidade de modificar mais de uma réplica para completar uma transação de atualização. Quanto mais réplicas do mesmo fragmento existirem, mais demorado pode ser o processo de atualização. Para minimizar o impacto da atualização de muitas réplicas, podem ser aplicadas técnicas alternativas de atualização. RÖHM *et al.* (2002), por exemplo, sugerem uma técnica de atualização que dedica algumas réplicas à atualização imediata. Isto permite que a atualização seja feita apenas em poucas réplicas, confirmando em seguida a atualização à aplicação cliente. Porteriormente, as outras réplicas recebem a atualização.

II.3 Processamento Paralelo de Consultas em DBC

O projeto de distribuição de dados influencia diretamente nas opções de processamento paralelo de uma consulta. Nesta seção analisaremos algumas alternativas de paralelismo em processamento de consultas em DBC.

II.3.1 Paralelismo Inter-Consulta

O paralelismo inter-consulta é a capacidade de processamento de mais de uma consulta simultaneamente. Usando este paralelismo, é possível iniciar o processamento de duas consultas, simultaneamente, em nós distintos. O objetivo é aumentar a vazão global do DBC fazendo com que a carga de um conjunto de consultas seja distribuída entre os seus nós. No entanto, usando apenas o paralelismo inter-consulta, o tempo de

execução de uma consulta não é reduzido, porque todo o trabalho de processamento de uma consulta é feito inteiramente no mesmo nó. Portanto, para o ganho de desempenho ser notado, o DBC que usa este paralelismo deve atender a aplicações que geram grande quantidade de consultas simultâneas.

O paralelismo inter-consulta é ideal para sistemas *online* de múltiplos-usuários com alto volume de transações, característico de ambientes OLTP. Ao receber a consulta, o DBC deve escolher qual SGBD deve processá-la. A escolha, por exemplo, pode ser baseada na carga de processamento atual de cada SGBD. O SGBD com o menor número de consultas em execução seria escolhido. Em seguida, o DBC envia a consulta ao SGBD escolhido e aguarda o resultado. Ao final do processamento, o DBC envia o resultado da consulta à aplicação cliente. O DBC apenas age como um mediador entre a aplicação cliente e os SGBDs.

II.3.2 Paralelismo Intra-Consulta

O **paralelismo intra-consulta** utiliza mais de um nó do agrupamento para processar uma mesma consulta. Os computadores executam os mesmos operadores relacionais em paralelo, mas cada um em uma parcela distinta dos dados. A carga de trabalho é distribuída entre os nós do DBC de acordo com a fragmentação dos dados a serem processados. O paralelismo intra-consulta é a técnica frequentemente empregada para reduzir o tempo de execução de uma consulta pesada. Idealmente, é esperado que o tempo de execução da consulta seja reduzido em n vezes caso o DBC use n nós para o processamento.

O paralelismo intra-consulta atende eficientemente sistemas de suporte à decisão, característicos por processarem consultas OLAP, que ocupam o SGBD por muito tempo e apresentam grande número de operações de E/S. As consultas OLAP percorrem grande parte da base de dados para sintetizar resultados.

Um SGBD paralelo geralmente oferece várias técnicas de fragmentação de dados que podem ser usadas durante o projeto de distribuição de dados. De acordo com a estratégia de fragmentação e alocação, o processamento de uma consulta pode ser tanto por paralelismo inter-consulta quanto por intra-consulta. O SGBD, para escolher o melhor plano de execução paralelo, pode consultar as estatísticas das bases de dados juntamente com seu projeto de distribuição. Este SGBD tem total controle sobre o plano de execução paralelo da consulta. Entretanto este não é o caso de um DBC. O DBC não

tem acesso a estatísticas do banco dados e também não tem controle total da maneira pela qual uma determinada tabela é lida, por exemplo. Sendo desacoplada do SGBD, a camada do DBC tem de implementar as técnicas de paralelismo externamente ao conjunto de SGBDs do agrupamento usando os mesmos recursos disponíveis a qualquer aplicação cliente.

Para realizar o paralelismo intra-consulta em um DBC, uma ou mais tabelas utilizadas na consulta precisam estar fragmentadas e distribuídas entre os nós do DBC. A fragmentação física das tabelas pode ser complexa, difícil de manter e pode causar distorção de dados. Além do mais, a geração de um plano de execução paralelo que atenda às particularidades do projeto de distribuição pode ser uma tarefa complexa para a camada do DBC.

Usando paralelismo intra-consulta sobre uma base de dados replicada e fragmentada virtualmente, o processamento de consultas pesadas pelo DBC é simplificado. Além disso, o DBC teria maior flexibilidade para definir os fragmentos e, conseqüentemente, para distribuir o processamento das consultas nos nós. O DBC deve ter a capacidade de analisar sintaticamente a consulta, identificar as tabelas candidatas a fragmentação virtual e gerar as subconsultas sobre cada fragmento.

II.4 Composição de Resultados

A **composição de resultados** é o processo pelo qual o DBC constrói o resultado final de uma consulta processada por paralelismo intra-consulta. O processo é necessário para processamento consultas por paralelismo intra-consulta tanto sobre bases de dados fragmentadas fisicamente quanto virtualmente. Antes de gerar uma resposta para a aplicação cliente, a camada de software do DBC deve aguardar a resposta de todas as requisições enviadas aos SGBDs e fazer uma agregação dos resultados. Nos casos mais simples, isto corresponde a apenas uma concatenação dos resultados parciais obtidos.

Vamos considerar o exemplo de uma consulta processada por paralelismo intra-consulta sobre uma tabela fragmenta horizontalmente. Os resultados parciais são armazenados até que todos sejam obtidos, para, então, efetuar a composição. Se a consulta contiver uma operação de ordenação, a composição consiste em concatenar

todos os resultados parciais e ordenar as tuplas segundo a operação de ordenação definida na consulta original.

II.5 Controle de Replicação

A manutenção da réplica dos fragmentos só é possível pelo funcionamento de dois controles de execução de consulta: controle de concorrência e controle de propagação de atualizações (KEMME, 2000). O primeiro controle define em que ordem as consultas recebidas simultaneamente são executadas contra as bases de dados. O último controla o momento em que as atualizações são enviadas para as réplicas e qual é a ordem em que isto ocorre.

II.5.1 Controle de Concorrência

O controle de concorrência é um problema inerente a qualquer SGBD – seqüencial ou paralelo – que atende simultaneamente a mais de uma transação. O mesmo também ocorre para DBC. O controle de concorrência pode ser definido como a atividade de gerência de operações de escrita simultâneas sobre um mesmo conjunto de dados compartilhados (BERNSTEIN *et al.*, 1986). Essas operações que modificam a base de dados e operações de leitura são agrupadas em unidades lógicas conhecidas como transações. As transações podem ser submetidas ao DBC simultaneamente, possivelmente gerando operações de escrita intercaladas. O DBC precisa resolver qual é a ordem possível para intercalar estas operações para que o resultado final das transações seja correto. O critério mais simples de verificação de correção para intercalar operações de escrita é a serializabilidade de cópia única (*1-copy-serializability).* Isto significa que o resultado final da execução simultânea de transações deve ser o mesmo que seria obtido se elas fossem executadas de forma serial, ou seja, uma após a outra.

O controle da ordem de execução de transações em um DBC fica sob responsabilidade de um componente denominado *escalonador* (BERNSTEIN *et al.*, 1986). O escalonador pode ter políticas de concorrência conservadoras ou liberais. Se usar políticas conservadoras, o escalonador pode efetivamente favorecer a execução de uma transação por vez enquanto posterga as restantes. Na direção de políticas mais liberais, a intercalação de operações de escrita de mais de uma transação é permitida.

Quanto mais liberal, melhor é o desempenho, mas também aumenta as chances de operações conflitantes. O mecanismo clássico para resolver o problema de acesso simultâneo aos dados é por meio de bloqueios (BERNSTEIN *et al.*, 1986). Cada objeto da base tem um bloqueio associado. Em DBCs, geralmente este objeto representa uma tabela. Antes de uma transação T_1 alterar o objeto, o escalonador verifica se existe algum bloqueio B_1 ativado para aquele objeto. Se o bloqueio B_1 não estiver ativado, ele é associado à transação T_1. Se outra transação T_2 já for dona do bloqueio B_1, a transação T_1 aguarda a liberação do bloqueio. Ou seja, o escalonador permite que apenas uma transação tenha o bloqueio B_1 por vez. Esta política é conhecida como *two phase locking* (2PL). No entanto, o uso de bloqueios sob longas transações pode levar o sistema a um travamento (*deadlock*). Se, por exemplo, a transação T_1 já tiver o bloqueio B_1, enquanto busca pelo bloqueio B_2, e a transação T_2 já tiver o bloqueio B_2, enquanto busca o bloqueio B_1, as duas transações vão esperar infinitamente pela liberação dos bloqueios. Para corrigir as situações de *deadlock* nos casos mais gerais, normalmente é empregado um contador de tempo de espera por liberação de recurso. Se o contador atingir um determinado valor, a transação é cancelada, liberando os bloqueios associados àquela transação.

Este algoritmo prevê que o escalonador seja único para as bases de dados. Isto se aplica para o caso de uma camada de DBC centralizada. Já no caso da *camada* do DBC com arquitetura autônoma e distribuída, em que cada banco de dados possui um escalonador distinto, novos problemas emergem. A ordem de execução das atualizações deve ser a mesma em todas as réplicas da base de dados, por exemplo. Os escalonadores devem colaborar para definir uma ordem única de execução das atualizações. COULON *et al.* (2004) analisam uma solução para este caso que garante preventivamente a consistência da bases de dados.

II.5.2 Controle de Propagação de Atualização

Quanto maior for o número de réplicas dos fragmentos, maior pode ser o atraso para processar uma transação de atualização no DBC. Dependendo da aplicação a que se atende, diferentes políticas podem ser utilizadas para propagação de atualizações entre as réplicas.

A política de replicação, segundo GRAY *et al.* (1996), pode ser classificada de acordo com dois parâmetros: *onde* atualização é executada primeiramente e *quando* a

atualização é executada. Entendemos que "onde" significa se a propagação da atualização privilegia algumas réplicas ou é feita simultaneamente em todas as réplicas. A primeira forma é denominada **replicação de cópia primária**. A propagação da atualização é feita para réplicas primárias e, em seguida, assincronamente, para as cópias secundárias. Cada objeto da base de dados tem sua cópia primária em um ou mais nós. A outra forma, **replicação de cópia de múltiplos donos**, proporciona maior autonomia para os SGBDs do agrupamento porque não há dependência de nenhuma cópia primária. A propagação da requisição de atualização enviada pelo cliente não privilegia nenhum SGBD. Todos os SGBDs do DBC são atualizados simultaneamente.

As políticas de replicação também são caracterizadas segundo o momento em que a atualização é propagada. Podem ser de duas outras formas: as que propagam a atualização dentro da mesma transação – **replicação imediata** *(eager)* - e as que utilizam transações diferentes para a propagação – **replicação deferida** (*lazy*). Isto significa que, ao adotar a replicação imediata, é garantida a consistência de todas as réplicas da base ao final da transação. Porém, isto leva a um atraso por conta da necessidade de comunicação da atualização para todos os nós do agrupamento. Desta forma, a replicação deferida foi explorada em SGBDs paralelos e DBCs para proporcionar maior vazão de transações mesmo quando ocorrem atualizações freqüentes. Nesta política de replicação, a operação de atualização é efetuada no nó que recebeu a solicitação de atualização. Depois, então, assincronamente, a atualização é comunicada a outros nós. Atualmente ainda são feitas pesquisas que tentam garantir que não haja discordância entre as bases utilizando a replicação deferida. A Tabela 1 (GRAY *et al.*, 1996) resume a classificação proposta.

	Quando	
Onde	**i.** Imediata, cópia primária	**ii.** Deferida, cópia de múltiplos donos
	iii. Imediata, cópia de múltiplos donos	**iv.** Deferida, cópia de múltiplos donos

Tabela 1 – Classificação das políticas de replicação.

II.6 Soluções para DBC

Os principais projetos de DBCs que podem ser encontrados na literatura são PowerDB (AKAL *et al.*, 2002; RÖHM *et al.*, 2002), C-JDBC (CECCHET, 2004a), RepDB* (PACITTI *et al.*, 2005), SmaQ (LIMA, 2004) e o ParGRES (PARGRES, 2005). Apenas o PowerDB, o SmaQ e o ParGRES implementam o paralelismo intra-consulta e inter-consulta. Os outros somente implementam o paralelismo inter-consulta.

II.6.1 PowerDB

O projeto PowerDB pesquisa como distribuir a carga de trabalho de um banco de dados em um agrupamento de PCs e como processar as consultas de leitura juntamente com atualizações, dando todas as garantias transacionais. A pesquisa destas questões é realizada com o uso de uma camada que coordena o funcionamento de um agrupamento de SGBDs não-paralelos. Em AKAL *et al.* (2002), é avaliado o uso da técnica de paralelismo intra-consulta por meio da SVP. Neste trabalho não foram feitos experimentos com transações de atualizações concorrentes. Referenciamos este trabalho como **PowerDB/SVP**. Apesar de prover paralelismo intra-consulta, não garante aceleração de consulta porque depende da otimização específica do plano de consulta do SGBD. Seus resultados, portanto, são instáveis. Além do mais, em contraste com nossa motivação de oferecer uma solução baseada em software livre, o software do PowerDB não está livremente disponível.

Em outra publicação (RÖHM *et al.*, 2002), aqui denominada de **PowerDB-FAS**, é avaliado o desempenho de requisições de atualização em um DBC. Neste caso, ao invés de processar as consultas pelo paralelismo intra-consulta, é usado o paralelismo inter-consulta. O diferencial desta proposta é o envio de consultas juntamente com um parâmetro determina o grau de atualização de dados necessário à consulta (*freshness*). Usando um mecanismo de replicação deferida, as atualizações são enviadas para um nó do agrupamento dedicado a isto, e propagada aos poucos para os outros nós. Assim as atualizações podem ser postergadas para minimizar o impacto negativo no desempenho das consultas apenas de leitura. Desta forma, o grau de atualização indica tolerância para iniciar o processamento de consulta sobre dados que ainda não foram atualizados.

II.6.2 SmaQ

O SmaQ é um protótipo que resultou de recente tese de doutorado de Alexandre A.B Lima (LIMA, 2004) que, assim como o PowerDB, é focado em sistemas OLAP. O SmaQ emprega uma técnica chamada de fragmentação virtual adaptativa (*AVP – Adaptive Virtual Partitioning*) (LIMA *et al.*, 2004a) que reduz o tempo de execução da consulta e permite o balanceamento dinâmico de carga entre os nós durante sua execução. O SmaQ suporta paralelismo inter e intra-consulta, mas não está preparado para resolver transações de atualização. A proposta da AVP é melhorar a SVP, pois é estável, ou seja, independe de otimizações do plano de consulta do SGBD e seu balanceador de carga dinâmico compensa as distorções na distribuição dos dados.

A camada do SmaQ possui um componente global que é posicionado em apenas um nó e componentes locais que são colocados junto a cada SGBD do agrupamento. Quando a consulta é enviada ao componente global, este envia subconsultas sobre fragmentos virtuais distintos para cada componente local da camada. Entretanto, diferentemente do PowerDB, não executa a subconsulta sobre o fragmento virtual de imediato. O componente local, ao invés de enviar uma única subconsulta sobre o fragmento virtual, faz várias subconsultas sobre fragmentos deste fragmento virtual. O processamento e o cálculo do tamanho destes fragmentos são incrementais. O processamento de pequenos fragmentos pretende levar o otimizador de consultas do SGBD a utilizar o índice agrupado. A probabilidade do otimizador escolher varrer toda a tabela é mínima. Como o fragmento virtual é processado incrementalmente, os componentes locais podem redistribuir a responsabilidade de processamento destes fragmentos em tempo de execução.

No entanto, experimentos mostraram que a execução de consulta OLAP em ambientes de alto nível de concorrência pode levar a baixo desempenho (LIMA, 2004). Para ilustrar esta situação, vamos, por exemplo, analisar o caso em que componente local está processando três subconsultas diferentes C_1, C_2, C_3 sobre o mesmo fragmento virtual. Se iniciadas em tempos distintos, em cada instante suas subconsultas estarão lendo porções diferentes do fragmento virtual. Uma C_1 poderia estar consultando o início do fragmento, enquanto a C_2 estaria processando o meio e a C_3 o final. Se o fragmento virtual couber em memória, não teremos degradação do desempenho porque o tempo de acesso aos dados seria constante. No entanto, no caso contrário, estas subconsultas estariam forçando o movimento não organizado do disco de

armazenamento para buscar os blocos de dados que não estão em memória. O movimento do disco aumenta bastante o tempo de acesso ao dado. Além disso, como o acesso ao disco não é organizado, as subconsultas estariam utilizando mal o *cache* de disco do sistema operacional. Já na SVP, o controle da concorrência de acesso ao disco fica sob responsabilidade do SGBD, que pode escalonar as leituras de forma mais inteligente. O disco se movimentaria menos e o *cache* seria melhor utilizado.

II.6.3 ParGRES

O ParGRES (PARGRES, 2005), é um software livre desenvolvido na COPPE/UFRJ e publicado no portal de software livre ObjectWeb, através da licença LGPL. O ParGRES (MATTOSO *et al.*, 2005a) é uma evolução do protótipo SmaQ, transformado em um DBC que possa ser utilizado em ambientes de produção. O ParGRES mantém as vantagens do SmaQ, oferece um analisador sintático de consultas que segue o padrão SQL 99 e um compositor de resultados robusto. Além disso, ele prevê que operações de atualização sejam recebidas. Os mecanismos de concorrência do ParGRES são concervadores. Mesmo que receba solicitações de atualizações simultâneas, o ParGRES efetua apenas uma transação de atualização por vez. A próxima atualização só é executada após a confirmação de término da atualização corrente. Adicionalmente, pode-se dizer que o ParGRES possui a mesma limitação do SmaQ quanto à execução de consultas em situações de alta concorrência. Como utiliza o AVP para paralelismo intra-consulta, o processamento de consultas OLAP é eficiente apenas em ambientes de baixa concorrência.

II.6.4 RepDB*

O RepDB* (PACITTI *et al.*, 2005) é um projeto acadêmico de software livre que estuda a gerência e processamento de consultas em base de dados autônomas. Os pesquisadores deste projeto buscam desenvolver técnicas que encontrem uma boa relação entre alta vazão de processamento de consultas e consistência de dados. A tese é a de que um pequeno relaxamento da consistência entre as bases já oferece aumento significativo de vazão.

Experimentos em PACITTI *et al.* (2003) analisam o comportamento de consultas processadas por paralelismo inter-consulta em base de dados de consistência relaxada. Sua arquitetura é totalmente descentralizada – se assemelhando a uma abordagem

ponto-a-ponto (*peer-to-peer*) –, pois sua camada de software é igualmente distribuída entre os nós do agrupamento. Cada nó do agrupamento é autônomo, ou seja, é capaz de receber requisições da aplicação cliente sem depender de outro nó do agrupamento. A sua replicação é deferida, uma vez que cada nó processa as atualizações independentemente, sem a necessidade de aguardar a confirmação de propagação da atualização para os outros nós. Para minimizar os conflitos em atualizações recebidas por nós distintos, o RepDB* assegura a mesma ordem de execução das atualizações em todas as bases de dados atrasando a submissão da atualização em um tempo t. O tempo t é igual à soma da diferença máxima entre os relógios de máquina do agrupamento e do tempo máximo de propagação de uma mensagem no agrupamento. Como cada atualização recebe um timbre de hora h, no momento $t+h$ estima-se que todas as atualizações mais recentes já tenham sido processadas e, portanto, a atualização possa ser processada. Experimentos comprovam que este modelo é escalável, ou seja, o desempenho não degrada com o uso de mais nós no agrupamento. Durante os testes foi verificado que o efeito negativo do relaxamento de consistência foi minimizado pelo uso do protocolo proposto.

II.6.5 C-JDBC

O C-JDBC (CECCHET, 2004a) é um projeto em código livre para DBC de qualidade industrial que objetiva o aumento de desempenho e disponibilidade por meio do uso de um conjunto redundante de SGBDs. O C-JDBC permite o uso de SGBDs heterogêneos. Para o SGBD ser usado pelo C-JDBC basta apenas implementar a interface JDBC. Em sua configuração mais simples, o C-JDBC tem um controlador centralizado responsável por escalonar requisições simultâneas e balancear a carga de requisições entre os nós do agrupamento. No caso de configuração para alta disponibilidade, os nós do agrupamento podem ser separados entre dois ou mais controladores que notificam o recebimento de requisição de atualização por meio de um *software* de comunicação de grupo.

O C-JDBC, por ter as aplicações OLTP como foco, está melhor preparado para o processamento de alto volume de consultas e atualizações. Apenas a técnica de paralelismo inter-consulta é empregada. O C-JDBC permite que sejam usados diferentes esquemas físicos de distribuição de dados: totalmente replicado e parcialmente replicado. Como C-JDBC não usa fragmentação das tabelas, a granularidade da

replicação é a tabela inteira. No caso de sistemas de intensa atualização, o modelo de replicação parcial reduz o tempo de execução de uma requisição de atualização, posicionando as tabelas de atualização freqüente em menos nós do agrupamento. Não é usado nenhum esquema de replicação deferida.

Outro parâmetro importante utilizado no protocolo de propagação de atualizações utilizado pelo C-JDBC é: quantos SGBDs precisam responder à notificação de atualização para que seja enviada uma mensagem de retorno à aplicação cliente? Pode ser definido que o C-JDBC deva esperar que todos os SGBDs respondam, a maioria dos SGBDs respondam ou apenas o primeiro responda. Note que nas duas últimas opções, o cliente recebe a mensagem de confirmação de atualização, mas a transação ainda está em execução para alguns SGBDs. No caso do parâmetro estar definido para esperar todos os SGBDs responderem, o tempo de execução da atualização é igual ao tempo de execução do SGBD que demorou mais a responder. No caso de aguardar apenas a resposta do primeiro SGBD, o tempo de execução da atualização do DBC é igual ao melhor tempo de atualização em um SGBD. Esta é a configuração que oferece melhor desempenho.

II.7 Banco de Dados para Agrupamento de PCs

As soluções apresentadas nesta seção são extensões de tradicionais SGBD seqüenciais para processamento paralelo. Ao contrário do DBC, essa extensão é intrusiva e está intrinsecamente ligado ao SGBD a que estende.

II.7.1 MySQL

O MySQL (MYSQL, 2005) é um famoso SGBD em código livre, reconhecido pelo alto desempenho para requisições de leitura. Oferece duas extensões para ganho de desempenho com o uso do paralelismo do agrupamento de PCs. A primeira, batizada de *MySQL Replication*, oferece um modelo de replicação deferida em que um SGBD age como principal (*master*), enquanto outros agem como secundários (*slaves*). Todas as requisições de atualizações são enviadas apenas ao SGBD principal e as notificações de atualização são propagadas assincronamente ao outros nós. Os SGBDs secundários são conectados ao SGBD principal e aguardam no estilo do produtor-consumidor por notificações de atualização. Se atualizações forem enviadas diretamente a SGBD

secundários, não serão propagadas para outros SGBDs. Quando o SGBD principal considerar que está sob alta carga de trabalho, algumas consultas podem ser encaminhadas para os SGBDs secundários.

A segunda extensão, conhecida como *MySQL Cluster*, tem um mecanismo de replicação imediata, isto é, síncrona. São definidos três tipos de nós: nó de gerência, nó de SQL e nó de dados. Apesar destes componentes do *MySQL Cluster* serem recomendados para instalação em nós diferentes, mais de um componente pode ser agrupado em um nó. O nó de gerência provê serviços de controle para o agrupamento como um todo, assim como a configuração de distribuição de dados no agrupamento. O nó de SQL representa uma instância comum do MySQL que recebe as requisições da aplicação cliente. Finalmente, o nó de dados armazena as tabelas replicadas e fragmentadas da base de dados. O nó de dados é baseado no NDB (*network database*), que é um sistema de armazenamento de dados em memória. Por ser em memória, toda a base de dados deve ser pelo menos do mesmo tamanho do somatório de memória RAM disponível no agrupamento. Para grandes bases de dados, isto é um importante limitador.

O *MySQL Cluster* pode diminuir o tempo de execução de atualizações porque a base de dados pode ser fragmentadas entre os nós de dados. Por ter a base fragmentada e replicada, as consultas podem ser processadas tanto por paralelismo intra-consulta quanto por inter-consulta. O *MySQL Cluster*, desta forma, pode atender tanto a aplicações OLAP quanto OLTP, desde que a base de dados não ultrapasse os limites de memória RAM do agrupamento de PCs.

II.7.2 Oracle Real Application Cluster 10g

O Oracle 10g, tradicional SGBD comercial, possui uma extensão para execução em agrupamento de PCs. Este SGBD paralelo propõe alto desempenho tanto a aplicações OLAP quanto OLTP (CRUANES *et al.*, 2004). O Oracle RAC (*Real Application Cluster*), desta forma, processa requisições por paralelismo inter-consulta ou por intra-consulta. Sua arquitetura é de disco compartilhado e, sendo assim, todos os PCs do agrupamento compartilham um mesmo dispositivo de armazenamento. Para que o disco não se torne o gargalo do desempenho, geralmente é utilizada uma SAN (*Storage Area Network*) de alto desempenho. Os dados estão centralizados no disco

compartilhado e são acessados simultaneamente por instâncias do Oracle RAC em cada nó do agrupamento.

O Oracle RAC usa tecnologia de redes de alto desempenho para evitar acesso a disco e controlar a coerência entre os *caches* de disco de cada instância. Tradicionalmente, os SGBD paralelos que utilizam esta arquitetura usam o próprio disco compartilhado para notificar as alterações de dados entre os nós do agrupamento. Por meio da tecnologia de fusão de *caches* (*Cache Fusion*) (LAHIRI *et al.*, 2001), o Oracle emula um espaço de memória cache único e compartilhado para todos os SGBDs. Usando este mecanismo, é possível até transferir pela rede blocos de discos entre duas instâncias do Oracle RAC.

O alto custo de disco de compartilhamento, a necessidade de *hardware* de rede especializado e o custo da licença do Oracle RAC 10g tornam a solução dispendiosa. Sendo assim, são restritas as aplicações que podem usar este tipo de solução.

II.7.3 IBM DB2 Integrated Cluster Environment

O *DB2 Integrated Cluster Environment* (ICE) (DB2, 2005) é uma extensão do SGBD IBM DB2 também fornecido pela IBM. O DB2 é proposto como uma solução de baixo custo que usa agrupamento de PCs *commodities*, baseado no sistemas operacional Linux. Este SGBD paralelo usa arquitetura de memória distribuída e não requer nenhum hardware específico. O pacote é distribuído em conjunto com uma ferramenta para auxiliar fragmentação física da base. Como o DB2 ICE usa arquitetura de memória distribuída, os dados são fisicamente fragmentados e alocados entre os nós do agrupamento. O DB2 ICE, baseado no esquema físico dos dados, pode usar o processamento intra-consulta ou inter-consulta.

II.8 Conclusão

Nesta seção, o projeto PowerDB é distinguido entre PowerDB/SVP e PowerDB/FAS. Apesar de ser um mesmo projeto, foram implementados *softwares* distintos, cada um focado em um grupo de técnicas diferente. O PowerDB/SVP porque aplica o algoritmo SVP e o PowerDB/FAS porque estuda o aumento da vazão postergando as transações de atualizações. O MySQL, pelo mesmo motivo, foi subdividido entre *MySQL Replication* e *MySQL Cluster*.

Analisando as soluções existentes de banco de dados no ambiente de agrupamento de PCs fica claro que cada técnica de paralelismo de consulta pode melhor atender um determinado perfil de aplicação. Aplicações OLTP são reconhecidas por gerarem alto volume de requisições simples tanto de leitura quanto de escrita. Para este tipo aplicação, é esperado que o banco de dados seja escalável, ou seja, consiga manter a média de tempo de execução das consultas independentemente da quantidade de clientes participantes. Manter a vazão do sistema é mais importante do que diminuir o tempo individual da consulta. Portanto, o paralelismo inter-consulta é a técnica indicada para OLTP.

Aplicações OLAP são dominadas pela execução de consultas complexas que varrem grande parte da base de dados e, conseqüentemente, são demoradas. Nesta situação, o banco de dados da aplicação deve responder às requisições no menor tempo possível. A quantidade de usuários simultâneos dessa aplicação geralmente é pequena. O tempo de execução individual da consulta é determinante para o sucesso da infra-estrutura de DW. Em virtude da necessidade de garantir o menor tempo, as soluções de banco de dados específicas para este tipo de aplicação priorizam a execução das consultas em detrimento das transações de atualização. As propriedades ACID do banco de dados são relaxadas ou, simplesmente, o ambiente de DW é controlado para que as atualizações aconteçam apenas fora do horário de uso intenso do agrupamento (como por exemplo: de madrugada, final de semana). Desta forma, verificamos que a recomendação de postergar o processo de atualização nas bases de DW faz com que os dados usados pelas aplicações OLAP estejam defasados em relação aos que são utilizados pelas aplicações OLTP.

Este distanciamento entre OLAP e OLTP pode ser verificado entre soluções de DBCs existentes. São apenas três DBCs que dispõe de paralelismo intra-consulta: PowerDB/SVP, SmaQ e ParGRES. O foco em alto desempenho em consultas complexas é tão forte que apenas uma dentre estas soluções dispõe de suporte a transações de atualização: o ParGRES. Entretanto, o ParGRES não foi preparado para alta concorrência de consultas OLAP ou transações de atualização.

Entre os SGBDs para agrupamento de PCs, identificamos a combinação entre paralelismo intra-consulta e atualizações na mesma solução. Entretanto, o *MySQL Cluster*, apesar de empregar o paralelismo intra-consulta, não pode ser listado como solução para DW porque a soma do tamanho dos fragmentos da base instalada em cada

nó tem de ser necessariamente menor do que a memória RAM disponível. A solução é baseada em armazenamento de dados em memória. O Oracle RAC 10g e o DB2 ICE, segundo as funcionalidades oferecidas, podem comportar aplicações OLAP com as características transacionais do OLTP. Porém, ambos têm um universo de aplicações restrito porque a aquisição de qualquer um dos dois é dispendiosa. O pagamento de suas licenças requer uma soma recursos que geralmente ultrapassam o próprio valor do agrupamento de PCs. Além disso, o Oracle RAC 10g ainda requer o uso de SAN.

A Tabela 2, uma das contribuições de nossa pesquisa, mostra uma análise comparativa dos trabalhos expostos neste capítulo relacionando suas características de paralelismo em processamento de consultas e replicação de dados. Os trabalhos são agrupados entre "DBC" e "Banco de Dados para Agrupamento".

		Paralelismo		Replicação			
				Onde		Quando	
		Inter-consulta	Intra-consulta	Cópia primária	Cópia de múltiplos donos	Imediata	Deferida
Agrupamento de BDs	PowerDB/SVP	x	x	*não controla atuallização*			
	PowerDB/FAS	x		x			x
	RepDB*	x			x		x
	C-JDBC	x			x	x	
	SmaQ	x	x	*não controla atualização*			
	ParGRES	x	x		*x*	*x*	
BD para Agrupam. de PCs	MySQL Rep.	x		x			x
	MySQL Cluster	x	x		x	x	
	Oracle RAC	x	x	?	x	?	?
	DB2 ICE	x	x	?	x	?	?

Tabela 2 – Tabela comparativa dos trabalhos relacionados; campos marcados como *x* indicam a existências da característica e *?* indica que a característica não foi identificada.

Podemos observar na Tabela 2 que apenas o ParGRES oferece uma solução de banco de dados para ambiente de agrupamento de PCs que seja de baixo-custo (composto apenas por *hardware commoditie* e software livre) e que oferece alto desempenho para processamento de consultas OLAP, também prevendo transações de atualização. No entanto, o ParGRES não está preparada para alto desempenho em execução de consultas OLAP concorrentes ou executar transações de atualização simultaneamente. Não existe atualmente um DBC que execute muitas consultas OLAP concorrentes, mesmo que durante o processamento de atualizações.

No próximo capítulo apresentamos a abordagem de DBC adotada em nosso trabalho para aumentar o desempenho de consultas OLAP processadas concorrentemente a atualizações.

III Apuama e o Paralelismo Intra-Consulta

O Apuama compõe uma infra-estrutura de DBC para processamento de consultas OLAP sobre base de dados freqüentemente atualizada. Ele usa o paralelismo de processamento de um agrupamento de PCs para diminuir o tempo de execução das consultas e aumentar a vazão do sistema. Como vimos no capítulo anterior, a técnica de paralelismo de consultas mais indicada para consultas OLAP é o paralelismo intra-consulta. A estratégia de paralelismo intra-consulta adotado pelo Apuama é o mesmo usado pelo PowerDB: a SVP. O Apuama estende o C-JDBC adicionando a capacidade de processamento por paralelismo intra-consulta. As consultas processadas por paralelismo inter-consulta não sofrem interferência do Apuama e continuam a ser controladas exclusivamente pelo C-JDBC. O controle de concorrência e a propagação de atualizações são de responsabilidade do C-JDBC. O Apuama faz apenas uma interferência no funcionamento da propagação de atualizações. Como veremos a seguir, isto é necessário para garantir a consistência dos resultados do processamento por paralelismo intra-consulta.

Neste capítulo explicaremos como o Apuama usa o paralelismo intra-consulta combinado com transações de atualização. Na seção III.1 analisamos como a fragmentação virtual é usada pelo Apuama. Na seção III.2 apresentamos a nossa abordagem para processamento de consultas. Na seção III.3, apresentamos a composição de resultados para consultas processadas por paralelismo intra-consulta pelo Apuama. Na seção III.4 mostramos como o Apuama controla a consistência entre nós para que cada base de dados seja idêntica entre si antes de disparar as subconsultas da SVP.

III.1 Fragmentação Virtual no Apuama

A VP oferece flexibilidade porque, ao contrário da fragmentação física, os fragmentos podem ser definidos em tempo de execução. Na SVP, a fragmentação virtual de uma tabela R é feita de acordo com um atributo a da tabela R. A aplicação de uma função de fragmentação por faixa de valores sobre a, em um agrupamento de n nós, gera os fragmentos virtuais R_{A1}, R_{A2}, ..., R_{An}. Se o Apuama recebe uma consulta C,

são disparadas n subconsultas (C_1, C_2, ..., C_n) contra os n fragmentos virtuais correspondentes.

A simulação da fragmentação física pela VP apenas é eficiente caso as tuplas da tabela R estejam agrupadas fisicamente. Se uma tabela corresponde fisicamente a x blocos em disco, cada fragmento deveria ter o tamanho aproximado de x/n blocos de leitura. Se as tuplas de um determinado fragmento estiverem fisicamente espalhadas pelos blocos em disco da tabela, no pior caso a leitura de x blocos é necessária para recuperar todas as tuplas desse fragmento. Neste caso, o uso da VP para o paralelismo intra-consulta pode ser pior que o processamento seqüencial. Para obter o agrupamento físico das tuplas dos fragmentos basta que seja criado um índice agrupado sobre o VPA.

A maioria dos SGBDs permitem que as tuplas sejam fisicamente organizadas e agrupadas segundo um determinado atributo. Os SGBDs podem ser divididos entre aqueles que são responsáveis por automaticamente manter a organização física das tuplas e aqueles que precisam que a reorganização física seja expressamente solicitada. Os que pertencem à segunda categoria precisam ter suas tuplas reorganizadas periodicamente pelo administrador do banco de dados.

Entretanto, é necessário garantir que, por exemplo, o processamento da subconsulta C_3 seja feito pelo SGBD, acessando apenas os blocos de disco que contêm as tuplas de R_{A3}. Se, para buscar as tuplas de R_{A3} em disco, for necessário trazer para memória as tuplas de R_{A1} e R_{A2}, por exemplo, o tempo de execução da subconsulta não será satisfatório. Por esta razão, um índice agrupado I_A deve ser criado sobre o atributo a no SGBD. O SGBD, então, pode saber exatamente em que blocos do disco o fragmento virtual R_{A3} se inicia e termina. O SGBD, no pior dos casos, varre registros que não pertencem ao fragmento virtual apenas no primeiro e no último bloco lido. São os blocos que contém a fronteira deste fragmento virtual com outro fragmento virtual.

Mesmo que a base de dados possua o índice agrupado I_A sobre a tabela virtualmente fragmentada, o acesso dedicado às tuplas do fragmento virtual depende de que o otimizador do SGBD local escolha o uso do índice agrupado I_A ao gerar o plano de execução. Sem o uso do índice I_A, a SVP é ineficiente. O desafio de garantir o uso da VP é justamente forçar o uso do índice agrupado I_A, sem nenhuma alteração no código-fonte do otimizador de consultas do SGBD. Como o DBC é externo ao SGBD, não participa da construção ou seleção de nenhum plano de execução da consulta. Se o

otimizador de consultas do SGBD não reconhecer a necessidade do índice I_A, o desempenho do DBC baseado na SVP será considerado instável.

Portanto, o Apuama, para garantir o acesso eficiente ao fragmento virtual, interfere diretamente no otimizador do SGBD para forçar o uso do índice agrupado. Isto é feito enviando uma solicitação ao SGBD que desabilita a varredura seqüencial total da tabela, durante o processamento de consultas pesadas escolhidas pelo Apuama. Isto pode ser feito em muitos SGBDs populares, como é o caso do MySQL e do PostgreSQL[2]. Em alguns SGBDs, ao invés de desabilitar a leitura seqüencial, também é possível indicar explicitamente qual o índice a ser utilizado para leitura da tabela virtualmente fragmentada. Para o Apuama, é suficiente desabilitar a varredura seqüencial total de tabelas, antes de iniciar o processamento da consulta, usando o paralelismo intra-consulta. Este comando não interfere nas outras seções abertas do SGBD. Quando o processamento da consulta termina, a configuração original do SGBD é re-estabelecida. Essa interferência garante o que estamos chamando de estabilidade de acesso aos fragmentos virtuais definidos pela SVP. Na seção V.4.1 do capítulo de experimentos, mostramos o quanto é importante esta garantia.

Esta estratégia não é independente de SGBD porque os comandos necessários não são padronizados. Por esta razão, o Apuama precisa detectar qual *driver* de SGBD está sendo usado para fazer mudanças específicas no plano de execução da consulta. A informação do comando específico, que deve ser usado para cada SGBD, é registrada previamente junto ao catálogo do Apuama.

Um problema que surge com a utilização de VP em um DBC, que prevê transações de atualização, é a determinação dos limites de cada fragmento virtual. Como o controle sobre o armazenamento das tuplas de uma tabela é feito pelo SGBD, o DBC não possui a informação de quais são os limites da VPA de uma tabela virtualmente fragmentada. Caso o DBC fosse apenas utilizado em um ambiente apenas de leitura, o intervalo de valores do VPA, que define cada fragmento virtual de uma tabela, poderia ser determinado estaticamente pelo administrador do banco de dados. Entretanto, como o Apuama prevê atualizações das réplicas da base de dados, os limites precisam ser determinados dinamicamente. Caso contrário, o resultado pode estar incorreto.

---

[2] O comando específico do PostgreSQL é "`set enable_seqscan to off`"

Para que o resultado de uma consulta processada usando VP seja correto, a fragmentação utilizada deve ser completa e disjunta (ÖZSU, VALDURIEZ, 1999). Isto significa que no caso de VP sobre uma tabela R, que tem a como VPA, as tuplas de R devem pertencer a pelo menos um fragmento virtual R_{A1}, R_{A2}, ..., R_{An} (completo) e nenhuma tupla de R pode ser comum a mais de um fragmento virtual (disjunto). Portanto, para fragmentação ser completa, o menor valor de a em R deve ser o limite inferior da faixa de valores que define R_{A1} e o maior valor de a em R deve ser o limite superior da faixa de valores que define R_{An}. Além disso, para a fragmentação ser disjunta, o limite superior da faixa que define R_{AX-1} deve ser o limite inferior da faixa que define R_{AX}, para qualquer valor de X maior que 1 e menor ou igual a n.

A garantia de que a VP utilizada pelo Apuama é disjunta é resolvida de maneira simples. Definimos intervalos de VPA de mesmo tamanho para cada fragmento virtual. O problema está em descobrir os limites da VPA da tabela fragmentada virtualmente e, conseqüentemente, garantir que a fragmentação seja completa. Ao usar intervalos de mesmo tamanho, assumimos que o valor do VPA é homogeneamente distribuído. A distorção na distribuição destes valores pode gerar desbalanceamento de carga e, conseqüentemente, gerar má utilização dos recursos do agrupamento de PCs. Em nosso trabalho, está fora do escopo o tratamento de distorções na distribuição dos valores do VPA. Entretanto, a distorção na distribuição de valores do VPA não costuma ser um problema porque a VPA geralmente corresponde à chave primária e esta é gerada incrementalmente.

Os valores máximos e mínimos de a para toda tabela R são armazenados pelo Apuama, devendo ser calculados durante a inicialização do DBC e recalculados após cada atualização na base de dados. Após uma atualização em uma tabela de fato R precisamos recalcular os limites, porque não podemos ter certeza se os valores de a máximo e mínimo em R foram mantidos após esta operação. A consulta dos limites de a na tabela R pode ser feita pelo seguinte comando SQL:

```
select
       max(a) as limite_superior,
       min(a) as limite_inferior
from R
```

Contudo, para que esse recálculo não seja repetido depois de cada atualização da tabela R, o recálculo é postergado até antes da próxima consulta OLAP sobre R. Isto

evita recálculos desnecessários dos limites de *a*, como efetuar o recálculo dos limites entre a execução de duas operações de atualização.

No entanto, esta consulta pode ser processada ineficientemente, transformando o processo de recálculo dos limites dos fragmentos virtuais demorado. Verificamos que o PostgreSQL percorre todas as tuplas para descobrir os valores máximo e mínimo, quando na verdade era apenas necessário usar as informações de máximo e mínimo armazenadas no próprio índice da tabela. Para o PostgreSQL, forçar o uso do índice desabilitando a leitura seqüencial não é suficiente. O PostgreSQL continua consultando todas as tuplas. Para contornar este problema, seguimos sugestão oferecida por desenvolvedores do PostgreSQL. A consulta deve ser reescrita para forçar o uso do índice. A consulta dos valores máximo e mínimo deve ser feita em dois passos. Primeiro, buscamos o valor máximo do atributo *a*:

```
select a from R order by a desc limit 1
```

Em seguida, verificamos o valor mínimo:

```
select a from R order by a asc limit 1
```

Os experimentos com o Apuama mostram que a diferença do tempo de execução nesta segunda abordagem (com uso do índice) pode chegar a um ganho de desempenho de $4x10^3$ vezes em relação ao tempo que levaria usando a primeira abordagem (sem uso do índice).

III.2 Processamento Paralelo de Consultas no Apuama

Nossa abordagem prevê VP sobre as tabelas de fato de uma base de dados totalmente replicada. Isto permite que o Apuama processe consultas tanto por paralelismo inter-consulta quanto por intra-consulta. Na visão do C-JDBC, todas as consultas são processadas por inter-consulta. Mas, o Apuama, que media todas as interações entre o C-JDBC e o SGBD, tem a responsabilidade de decidir se a consulta é processada por paralelismo inter-consulta ou se deve ser também processada por paralelismo intra-consulta. Apesar do paralelismo intra-consulta visar a redução do tempo de processamento da consulta, nem todas as consultas OLAP recebidas pelo Apuama devem ser processadas por intra-consulta. A identificação da consulta

paralelizável (processada por paralelismo intra-consulta) é o que determina a interrupção do paralelismo inter-consulta para então iniciar o processamento do paralelismo intra-consulta. De uma forma geral, podemos dizer que a identificação de uma consulta paralelizável se baseia em verificar na sintaxe da consulta a existência de referência para uma tabela de fato. Entretanto, a atividade de identificação pode ser complexa. É necessário estimar se a consulta iria se beneficiar do paralelismo intra-consulta ou até mesmo se é possível reescrevê-la corretamente na forma de subconsultas SVP.

Em AKAL *et al.* (2002) é feita uma análise de um conjunto de razões para que o processamento por SVP de algumas consultas seja evitado. De acordo com esta análise, dividimos as consultas entre 4 categorias:

Tipo 1. consultas que não contêm tabelas de fato;

Tipo 2. consultas que contêm uma ou mais tabelas de fato, podem ser paralelizadas e têm potencial de ganho;

Tipo 3. consultas que contêm uma ou mais tabelas de fato, podem ser paralelizadas, mas não podemos prever se farão bom proveito do paralelismo intra-consultas.

Tipo 4. consultas que contêm uma ou mais tabelas de fato, mas não podem ser paralelizadas;

As consultas classificadas como tipo 1 são aquelas que simplesmente referenciam apenas as tabelas de dimensão. São consultas que geralmente não precisam ser paralelizadas porque a cardinalidade das tabelas referenciadas é baixa e o tempo de execução é curto. Caso uma tabela de dimensão tenha alta cardinalidade, podemos também aplicar VP sobre esta tabela. Esta tabela teria o mesmo tratamento que é dado a tabelas de fato para classificação das consultas. O tipo 2 indica a categoria das consultas que devem ser identificadas pelo Apuama como paralelizáveis. Estas consultas fazem referências a uma ou mais tabelas de fato. Veremos a seguir que nos casos de mais de uma tabela de fato, elas precisam estar relacionadas pelo seu atributo de fragmentação. A consulta pode também incluir referências a tabelas de dimensão. Os exemplos desta seção referenciam consultas especificadas pelo *benchmark* TPC-H, que é detalhado no Apêndice I. Em um primeiro exemplo, vamos observar a consulta Q14 do TPC-H:

```
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'
```

A consulta Q14 é paralelizável e considerada como de tipo 2. Ela contém apenas uma referência para a tabela de fato *lineitem* e outra referência para tabela de dimensão *part*. A única tabela virtualmente fragmentada é a *lineitem*. A aceleração esperada da consulta Q14 é próxima do linear porque a cardinalidade da tabela *lineitem* é muito superior a da tabela *part*.

As consultas que fazem apenas uma referência à tabela de fato são consideradas como de tipo 2. Entretanto, se a tabela de fato estiver presente em uma subconsulta, devemos observar algumas características. Se a consulta não tiver nenhuma referência a uma tabela de dimensão, mas apenas a tabela de fato referenciada pela subconsulta, a consulta pode ser paralelizada. Caso a consulta tenha uma ou mais referências a uma tabela de dimensão externa a subconsulta, é necessário que a tabela virtualmente fragmentada esteja relacionada a uma tabela de dimensão por meio do VPA para que a consulta seja considerada como paralelizável. Esta tabela de dimensão não pode fazer parte de uma subconsulta.

Para exemplificar esta situação, vamos considerar uma consulta com referência a duas tabelas: F e D. A tabela F é uma tabela de fato virtualmente fragmentada e D é uma tabela de dimensão. A consulta possui uma subconsulta que contém a referência à tabela de fato F. Durante o processamento da consulta, a subconsulta é executada sobre a tabela F para cada registro da tabela D. Isto significa que temos um resultado da subconsulta para cada registro da tabela D. Se a tabela F for virtualmente fragmentada, o resultado da subconsulta pode ser alterado. A subconsulta não irá percorrer os mesmos registros da tabela F. Para garantir que os resultados da subconsulta não sejam alterados, as tabelas F e D devem estar relacionadas na consulta através da VPA. Desta

forma, a adição dos predicados da VPA não alteraria o padrão de leitura dos registros da tabela F.

No caso da consulta ter mais de uma referência a tabelas de fato, alguns detalhes devem ser analisados para saber se consideramos a consulta como paralelizável ou não. Vamos tomar como exemplo a consulta Q4 do *benchmark* TPC-H:

```
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
```

A consulta possui referências às tabelas de fato *lineitem* e *orders*. A tabela *lineitem* é referenciada por uma subconsulta. A VP sobre as tabelas *lineitem* e *orders* é possível porque as tabelas são relacionadas pelos seus atributos de VP no predicado "`l_orderkey = o_orderkey`". O resultado final é correto porque é feita uma VP similar à fragmentação horizontal derivada. Os limites dos fragmentos virtuais de *lineitem* são os mesmos limites colocados para os fragmentos virtuais de *orders*. Esta consulta OLAP é paralelizável e categorizada também como tipo 2.

A análise de consultas que possuem mais de uma tabela virtualmente fragmentada deve verificar se a VP não altera a combinação entre as tuplas. Suponha uma consulta com duas tabelas virtualmente fragmentadas que sofrem uma operação de junção. Considere dois registros, um de cada tabela, que normalmente seriam combinados durante o processamento seqüencial da consulta. Se, usando o processamento por paralelismo intra-consulta, os registros estiverem em fragmentos virtuais que serão processadas por SGBDs distintos, o resultado do processamento desta consulta será

alterado. A única garantia de que o resultado não será alterado é por meio da verificação do relacionamento por meio da VPA entre as tabelas virtualmente fragmentadas.

Por esta razão, existem consultas que contêm mais de uma referência a tabelas de fato que não podem ser paralelizadas. Vamos considerar a consulta Q17 do *benchmark* TPC-H como exemplo:

```
Q17: select
        sum(l_extendedprice) / 7.0 as avg_yearly
from
        lineitem, part
where
        p_partkey = l_partkey and p_brand = 'Brand#13'
        and p_container = 'LG CASE' and l_quantity <
             (select
                0.2 * avg(l_quantity)
             from
                lineitem
             where
                l_partkey = p_partkey);
```

A consulta possui duas referências à tabela de fato *lineitem* e uma à tabela de dimensão *part*. Uma das referências à tabela de fato está contida em uma subconsulta. Se a VP fosse utilizada considerando todas as referências a tabelas de fato, a reescrita da Q14 iria inserir predicados sobre as duas referências de *lineitem*, considerando *l_orderkey* como atributo de fragmentação. O problema para paralelizar a consulta desta forma é que não seria possível construir um resultado final correto. O predicado "`l_quantity < (subconsulta)`" sobre a referência mais externa de *lineitem* exige que o resultado da subconsulta envolva todos os fragmentos virtuais da tabela *lineitem*. O uso de VP em duas ou mais tabelas de uma consulta só é possível quando estas estão relacionadas por seus respectivos VPAs. Ao contrário do que ocorre na consulta Q4, na consulta Q17 não há nenhum predicado na subconsulta que relacione as tabelas virtualmente fragmentadas. Fazendo a VP da tabela *lineitem* contida na subconsulta, o valor de retorno da subconsulta não envolveria todos os fragmentos virtuais necessários e, conseqüentemente, o resultado da consulta pode ficar incorreto.

Para ser paralelizada, apenas a referência mais externa de *lineitem* deve ser fragmentada virtualmente. Entretanto, como as duas referências a *lineitem* não foram fragmentadas, a quantidade de tuplas analisadas pela consulta não reduziu quase linearmente. A tabela *lineitem* seria analisada totalmente porque estaria sendo

referenciada sem VP pela subconsulta. Sendo assim, não sabemos se a consulta iria aproveitar eficientemente o paralelismo do agrupamento de PCs. Consideramos esta consulta como do tipo 3.

Modificando um pouco o exemplo da consulta Q17, vamos supor que a tabela mais externa não fosse uma tabela de fato. Como não podemos fragmentar virtualmente a tabela da subconsulta, a consulta não poderia ser paralelizada. Esta Q17 modificada seria considerada como do tipo 4.

O Apuama considera como consultas OLAP paralelizáveis apenas aquelas que pertencem ao tipo 2. São consultas que possuem referências a pelo menos uma tabela de fato e possivelmente a algumas tabelas de dimensão. Caso possuam referências a mais de uma tabela de fato e essas tabelas sofram junção ou estejam em diferentes níveis de profundidade de subconsulta, as tabelas precisam estar relacionadas por predicados que testam a equivalência dos seus VPA. Todas as consultas que não se encaixam neste perfil são diretamente repassadas pelo Apuama para o SGBD.

O processamento por paralelismo intra-consulta do Apuama segue o especificado pelo SVP. Cada SGBD tem a responsabilidade de processar um fragmento virtual. Uma subconsulta SVP é enviada para cada SGBD,

III.3 Composição de Resultados

As subconsultas produzidas pela técnica da SVP são processadas independentemente por cada nó e seus resultados parciais precisam ser combinados para formarem um resultado final da consulta. O Apuama usa o HSQLDB (HSQL, 2005), um rápido SGBD em memória para desempenhar a composição de resultado. Funcionar em memória permite que o SGBD seja acoplado ao processo do próprio Apuama e minimiza o tempo de processamento porque não exige operação de E/S. Para explicar a solução, vamos considerar, por exemplo, o caso da subconsulta *C'* abaixo:

```
C': select
        o_orderpriority, count(*) as order_count
from
        orders
where
        o_orderdate >= date '1993-07-01'
        and o_orderkey >= ? and o_orderkey < ?
group
        by o_orderpriority
```

A tabela *orders* é fragmentada virtualmente e o trecho em negrito indica o predicado adicionado à consulta *C* original. A subconsulta *C'* enviada para cada fragmento virtual produz um resultado parcial que deve ser combinado para construir um resultado final. Isto significa que os valores de *count(*)* agrupados por o_*orderpriority* obtidos em cada resultado parcial devem ser somados no cálculo do resultado final. Logo após o recebimento de um primeiro resultado parcial, o Apuama cria uma tabela temporária (*temp_result*) na base de dados em memória do HSQLDB. Para não ter conflito com o nome de tabelas temporárias de duas consultas diferentes e concorrentes, o nome da tabela pode ser acompanhado de um identificador gerado seqüencialmente. O esquema do resultado parcial equivale exatamente ao esquema da tabela temporária. Neste exemplo, como a subconsulta *C'* gera um resultado com apenas dois atributos, a tabela *temp_result* tem os mesmo atributos *o_orderpririty* e *order_count*, seguindo o mesmo tipo (data, inteiro, decimal...). Para todo novo resultado parcial recebido, o Apuama insere suas tuplas na tabela temporária. Após receber todos os resultados parciais, Apuama produz o resultado final da consulta, executando o seguinte comando SQL no HSQLDB:

```
select
        o_orderpriority, sum(order_count) as order_count
from
        temp_result
group by
        o_orderpriority
```

O resultado é enviado de volta à aplicação cliente e a tabela temporária é descartada. Esse método provou ser eficiente durante nossos experimentos. Em todos os experimentos, agregações desempenhadas pelo HSQLDB não duraram mais do que um segundo para serem processadas, mesmo que trabalhando com grandes resultados parciais.

III.4 Controle de Consistência entre Réplicas

O processamento de consultas por paralelismo intra-consulta pode ocorrer concorrentemente a atualizações da base de dados. Desta concorrência emerge o problema da consistência entre cada réplica da base de dados no momento do início da

execução das subconsultas SVP. A partir de uma consulta paralelizável *C* são disparadas *n* subconsultas (C_1, C_2, ..., C_n) contra cada uma das bases de dados replicadas. Se uma subconsulta C_i ($1 \leq i \leq n$) for processada em uma base de dados de diferente estado das bases de dados usadas nas outras subconsultas, o resultado final da consulta *C* será incorreto.

Nossa solução de DBC deve controlar a consistência entre as réplicas para garantir a correção do resultado do processamento de uma consulta por paralelismo intra-consulta. Em uma abordagem conservadora, o DBC permitiria apenas a execução de um operação de atualização por vez. Desta forma, todas as bases de dados sempre são idênticas ao fim da execução da operação de atualização. Como não teríamos execução concorrente entre operações de atualização e consultas por paralelismo intra-consulta, a consistência entre réplicas é garantida. Esta abordagem, utilizada pelo ParGRES, é eficiente para aplicações que não tenha carga de atualizações intensa.

Para prover melhor desempenho no cenário de operações de atualizações concorrentes, o DBC deve permitir que a execução de consultas por paralelismo intra-consulta ocorra simultaneamente a transações de atualização. A elaboração do controle de consistência entre réplicas do Apuama deve considerar as premissas de propagação de atualização específicas do C-JDBC.

III.4.1 Premissas de Funcionamento do C-JDBC

O C-JDBC controla a execução concorrente das operações de transações distintas, ordenando as requisições de acordo com o nível de isolamento desejado pelo administrador da solução de DBC. Na perpectiva do C-JDBC, por possuir apenas paralelismo inter-consulta, cada operação de leitura é enviada a apenas um SGBD. Em contrapartida, uma operação de atualização é enviada a todos os SGBDs que possuem a tabela que está sendo atualizada. Para garantir a consistência das bases de dados, o C-JDBC utiliza a premissa de **ordem total**. Se todas as operações de atualização forem executadas na mesma ordem serial em cada base de dados, a consistência da base de dados é garantida. Isto significa que o histórico de execução de atualização em cada base de dados é idêntico, dinstinguindo apenas o tempo de início e fim de execução da operação.

Adicionalmente, é necessário que a próxima operação de atualização seja executada apenas ao final da execução da operação de atualização corrente. Isto é

necessário para garantir que as operações sejam executadas pelo SGBD estritamente na ordem estabelecida pelo C-JDBC. A única forma de garantir que uma operação U_2 será processada pelo SGBD depois da operação U_1 é aguardando a resposta de U_1 para então submeter U_2. Em qualquer instante de tempo, o C-JDBC garante que apenas uma operação de atualização está em progresso em um SGBD específico. As operações de atualizações subsequentes a serem executadas em um SGBD específico são bloqueadas pelo C-JDBC em uma fila de espera. Esta fila segue o padrão de serviço FIFO. A fila é organizada para cada SGBD. Após a execução de uma operação de atualização, o SGBD está disponível para executar a próxima operação de atualização.

O Apuama recebe estas requisições do C-JDBC que seriam enviadas diretamente para cada SGBD. Cada operação recebida pelo Apuama está especificamente endereçada para um SGBD. Apenas uma operação de atualização pode estar endereçada a um SGBD em um dado instante. Em um C-JDBC com *n* nós, a mesma operação de atualização é recebida *n* repetidas vezes pelo Apuama, distinguindo apenas o SGBD a que está endereçado. O tempo de recebimento de cada uma destas operações geralmente é próximo, podendo aumentar caso o C-JDBC esteja sob alta carga de trabalho.

Para ilustrar o funcionamento da propagação de atualizações do C-JDBC, vamos observar a . O C-JDBC está conectado a três SGBDs. Existem quatro operações de atualização ainda em execução: U_1, U_2, U_3, U_4. Estas requisições foram enviadas por uma mesma aplicação cliente. O C-JDBC foi configurado para responder a requisição de atualização da aplicação cliente após o primeiro recebimento de resposta de um SGBD. Cada SGBD recebe um mesmo conjunto de operações de atualizações da mesma ordem definida pelo C-JDBC.

Como o terceiro SGBD já processou as operações de atualização U_1 e U_2, consideramos que aplicação cliente já recebeu as confirmações de execução de U_1 e U_2. No instante ilustrado na figura, o terceiro SGBD está recebendo a operação de atualização U_3. O primeiro SGBD é o mais lento, sendo o terceiro SGBD pouco mais rápido para o processamento de transações. Isto pode ser identificado por meio da fila de operações de atualizações que cada SGBD ainda resta processar. O primeiro SGBD está recebendo a operação U_1 e resta receber as operações U_2 e U_3. O segundo SGBD já processou U_1, está recebendo U_2 e ainda está para receber U_3. Note que a ordem total está garantida porque todas as operações são executadas na mesma ordem. Entretanto, a figura mostra que as operações de atualizações não estão ocorrendo no mesmo instante

para todos os SGBDs. Caso seja inicializado o processamento de consultas por paralelismo intra-consulta no exato instante ilustrado na figura, o resultado obtido deste processamento seria incorrento. As bases de dados não possuem o mesmo estado.

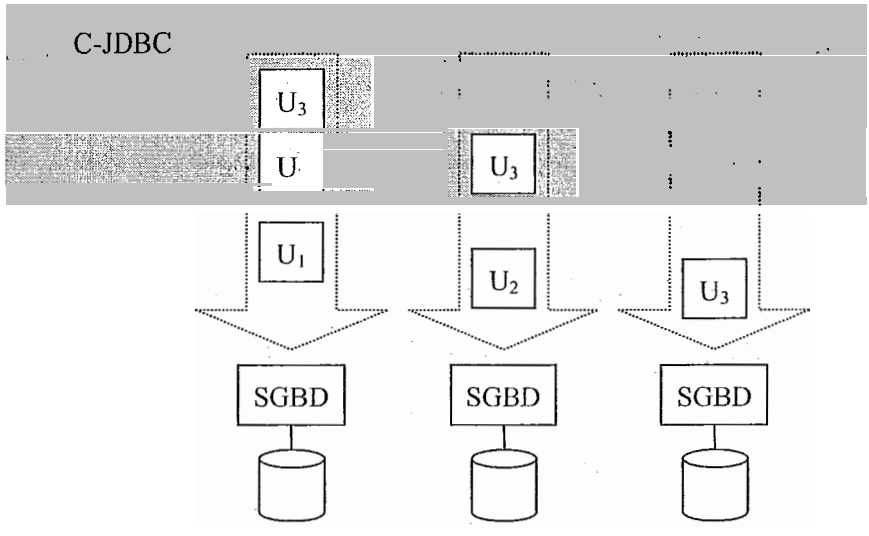

Figura 2 – Ilustração do funcionamento da propagação de atualização do C-JDBC

III.4.2 Solução

Para produzir resultados consistentes para consultas OLAP paralelizáveis, o Apuama deve garantir que, antes de iniciar o processamento usando paralelismo intra-consulta, as réplicas de todos os nós estejam consistentes entre si. O C-JDBC não está ciente da existência de subconsultas geradas pela SVP. O Apuama, por sua vez, não tem controle sobre a propagação das atualizações, mas apenas repassa as solicitações ao SGBD. Para o Apuama, a única forma de verificar o estado de cada SGBD é monitorar a execução das transações enviadas para cada um.

O Apuama provê um mecanismo de monitoração e bloqueio de atualizações para sincronizar a execução concorrente de atualizações e subconsultas SVP. O Apuama possui um contador de operações de atualização para cada réplica da base de dados. Como as operações são efetuadas estritamente na mesma ordem em cada SGBD, o uso de contador de operações de atualização é suficiente para indicar o estado de uma réplica. Quanto maior o contador, mais recente é a última atualização processada sobre aquela réplica.

Antes de submeter uma subconsulta SVP, o Apuama aguarda até que um estado consistente seja atingido por aquela réplica. O estado consistente corresponde a um valor de contador definido pelo Apuama. A definição deste estado consistente ocorre no momento do recebimento pelo Apuama da consulta a ser processada por paralelismo intra-consulta. O Apuama escolhe o maior valor comparando todos os contadores dos SGBDs. A subconsulta SVP apenas é enviada ao SGBD correspodente quando seu contador atinge o estado de consistência estipulado. O Apuama deve monitorar a execução das operações de atualização em cada SGBD para identificar o momento de envio da subconsulta SVP. Desta forma, todas as subconsultas SVP são processadas sobre réplicas idênticas da base de dados.

No caso em que uma nova operação de atualização é submetida enquanto o SGBD processa uma subconsulta SVP, o Apuama poderia bloquear a submissão desta operação até o término de execução da subconsulta SVP. Submeter uma nova operação de atualização antes do fim do processamento da subconsulta SVP poderia levar a resultados incorretos. Existe o risco de que a nova operação de atualização seja processada antes do ínico subconsulta SVP. No entanto, garantir a ordenação da execução destas requisições isolando o momento de execução de cada uma pode diminuir significativamente a vazão do sistema. Como a subconsulta SVP geralmente tem tempo de processamento muito superior a uma operação de atualização, o tempo de bloqueio da operação de atualização pode ser significativo.

No intuito de minimizar o efeito do uso de bloqueios, o Apuama permite a execução simultânea de operação de atualização e subconsultas SVP em um mesmo SGBD. A abordagem utilizada pelo Apuama permite que a submissão da operação de atualização não precise aguardar o fim do processamento da subconsulta SVP. Para garantir que a nova operação de atualização não seja processada antes do início da execução da subconsulta SVP, definimos uma transação de leitura antes do início da subconsulta SVP. Uma transação de leitura garante que, enquanto a transação está aberta, as consultas de leitura desta transação não são afetadas por modificação de dados gerados por diferentes transações. Sendo assim, o Apuama, ao identificar que um SGBD atingiu um estado consistente, abre uma transação de leitura para aquele SGBD. Isto garante que a subconsulta SVP será processada exatamente sobre o estado de consistência estipulado. Após a confirmação de que a transação de leitura foi iniciada, o Apuama pode liberar a submissão de novas operações de atualização sem ter o risco de

que a base de dados seja atualizada antes do início do processamento da subconsulta SVP. Sendo assim, o Apuama bloqueia as novas operações de atualização apenas enquanto aguarda pelo início da definição de transação de leitura. Ao receber a resposta de início da operação de leitura, o Apuama libera as operações de atualização bloqueadas na mesma ordem que foram recebidas.

III.5 Conclusão

Este capítulo apresentou a abordagem utilizada pelo Apuama para adicionar ao C-JDBC a capacidade de processamento de consultas por paralelismo intra-consulta. Como em outras soluções de DBC apresentadas, o Apuama usa a VP para processamento por paralelismo intra-consulta. Entretanto, diferentemente delas, o Apuama garante a utilização correta da VP interferindo no plano de consulta do SGBD. Esta interferência é feita utilizando um comando específico do SGBD que assegura o uso do índice agrupado, necessário para o funcionamento da VP. Apesar de não ser um comando padronizado entre os diferentes fornecedores de SGBDs, é uma funcionalidade comum a maioria dos SGBDs disponíveis no mercado.

Neste capítulo mostramos como o Apuama escolhe entre o processamento de consultas por paralelismo inter-consulta ou por paralelismo intra-consulta. É analisado o perfil das consultas que podem ser beneficiadas pelo processamento por paralelismo intra-consulta. Algumas consultas não são processadas por paralelismo intra-consulta porque não tem alto custo de processamento e, conseqüentemente, a redução absoluta do tempo de processamento seria mínimo. Outras consultas não são processadas por paralelismo intra-consulta porque não podem ser utilizadas com a técnica de VP.

O mecanismo composição de resultados do Apuama foi apresentado neste capítulo. O uso de um SGBD em memória permite flexibilidade para manipulação dos resultados parciais.

A única solução de DBC apresentada neste trabalho que atende a aplicações OLAP é o ParGRES. No entanto, o seu controle de concorrência segue uma política conservadora, não permitindo execução concorrente entre consultas processadas por paralelismo intra-consulta e operações de atualização. Neste capítulo apresentamos o controle de consistência entre réplicas do Apuama necessário para executar concorrentemente consultas processadas por paralelismo intra-consulta e operações de

atualização. O Apuama garante resultado consistente de consultas processadas por paralelismo intra-consulta. As operações de atualização bloqueadas são atrasadas no tempo mínimo necessário para iniciar uma transação de leitura e iniciar o processamento de uma subconsulta SVP.

IV Arquitetura do Apuama

Neste capítulo é descrita a arquitetura do Apuama. Na seção IV.1 explicamos como a arquitetura do C-JDBC foi estendida para integrar a execução de consultas OLAP. A seção IV.2 apresenta os aspectos internos da arquitetura do Apuama.

IV.1 Apuama como Extensão do C-JDBC

A Figura 3 exibe os componentes da arquitetura do C-JDBC relevantes para nossa explicação. O propósito principal do C-JDBC é oferecer acesso transparente aos SGBDs do agrupamento sem modificação da aplicação cliente. O único requisito para migração de um ambiente de SGBD seqüencial para um agrupamento é que a comunicação entre a aplicação e o SGBD precisa ser feita por meio de um *driver* JDBC (JDBC, 2005). A aplicação, ao invés de estar conectada diretamente ao SGBD, está conectada ao controlador C-JDBC usando um *driver* JDBC do C-JDBC.

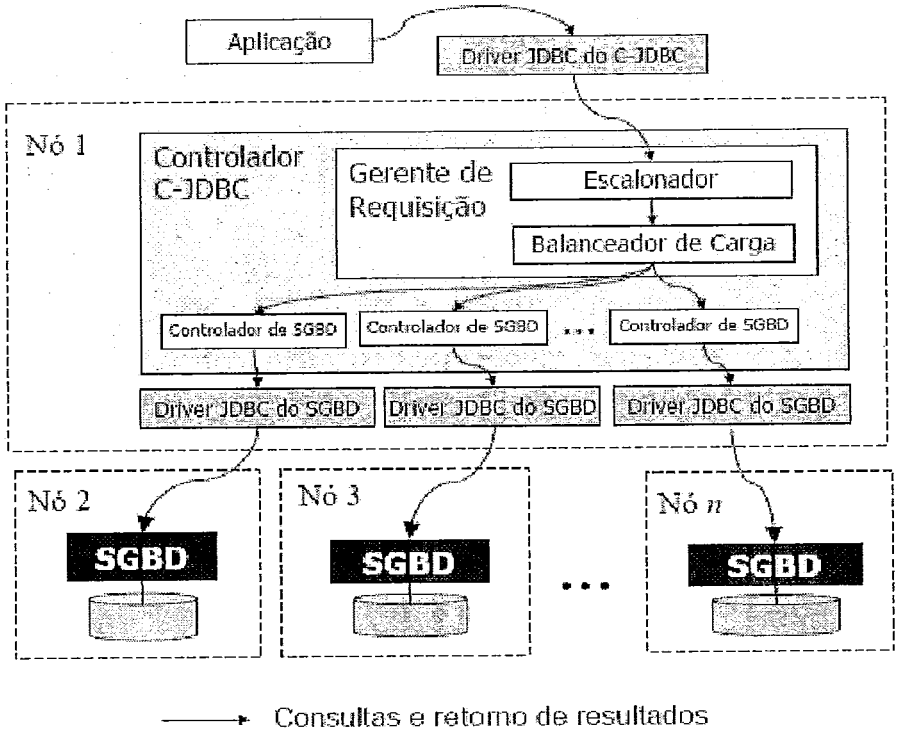

Figura 3 – Arquitetura do C-JDBC

A camada de software do C-JDBC pode ser configurada para ser distribuída entre os nós do agrupamento de PCs, aumentando a disponibilidade do sistema. Em nosso trabalho, utilizamos a configuração centralizada do C-JDBC. Como pode ser visto na figura, o C-JDBC é instalado em apenas um nó do agrupamento de PC, enquanto os SGBDs estão distribuídos entre os outros PCs do agrupamento.

A camada de software do C-JDBC é representada por seu controlador. O **controlador do C-JDBC** é um processo que gerencia todos os recursos de banco de dados. O controlador é composto por um gerente de requisição e um conjunto de controladores de SGBD. Para cada SGBD conectado, existe um componente **controlador de SGBD** que gerencia um repositório (*pool*) de conexões. O repositório de conexões gerencia o uso das conexões que são atribuídas a cada requisição enviada àquele SGBD.

O gerente de requisição é formado pelos componentes escalonador e balanceador de carga. Cada requisição recebida pelo C-JDBC é enviada ao componente **escalonador,** que controla a execução concorrente de requisições e se certifica de que as atualizações são executadas na mesma ordem por todos SGBDs. O escalonador pode ser configurado para permitir diferentes níveis paralelos de concorrência. O C-JDBC oferece três níveis de concorrência. O primeiro deles bloqueia a execução de qualquer transação de atualização concorrente. Uma transação é executada por vez. No segundo nível, são permitidas transações de atualização concorrentes, mas é necessário um mecanismo de detecção de travamento (*deadlock*). Este nível corresponde ao algoritmo exemplificado na seção II.4.1. O terceiro nível de concorrência ignora as transações e passa a responsabilidade de bloqueios das requisições ao SGBD. Em nossos experimentos, o escalonador foi configurado para o segundo nível. É o nível de concorrência recomendado pela equipe de desenvolvimento do C-JDBC à maioria das aplicações. Não é tão bloqueante quanto o primeiro nível e também não assume que o SGBD é capaz de resolver a maioria dos conflitos entre transações concorrentes. Como em nossos experimentos não usamos mais de uma transação de atualização, a decisão de usar este ou aquele nível não tem impacto em nossos resultados.

Depois do agendamento da execução da requisição, o componente balanceador de carga do C-JDBC escolhe qual controlador de SGBD irá executá-lo. Se for uma requisição de escrita, ela é executada em todos controladores de SGBD com objetivo de manter a consistência entre as bases de dados. Entretanto, se é uma requisição de leitura,

o balanceador de carga deve escolher o nó que irá executá-la. Podem ser configuradas três diferentes políticas para escolha do nó. A primeira delas é política circular (round robin). A segunda política também é circular, mas pode repetir a execução para determinados nós (weighted round robin). É atribuído um peso a cada nó. A escolha do nó é repetida quantas vezes for o peso do nó. A última política escolhe o nó que possui a menor quantidade de requisições pendentes. No nosso trabalho, usamos esta última política para o balanceador de carga. Consideramos que esta é a política de balaceamento mais adequada entre as apresentadas porque é única que usa uma medida de carga de trabalho em execução no nó para fazer o roteamento da requisição de leitura. Não é interessante usar uma política circular porque as consultas OLAP possuem diferentes complexidades e, conseqüentemente, não geram a mesma carga de processamento sobre o nó.

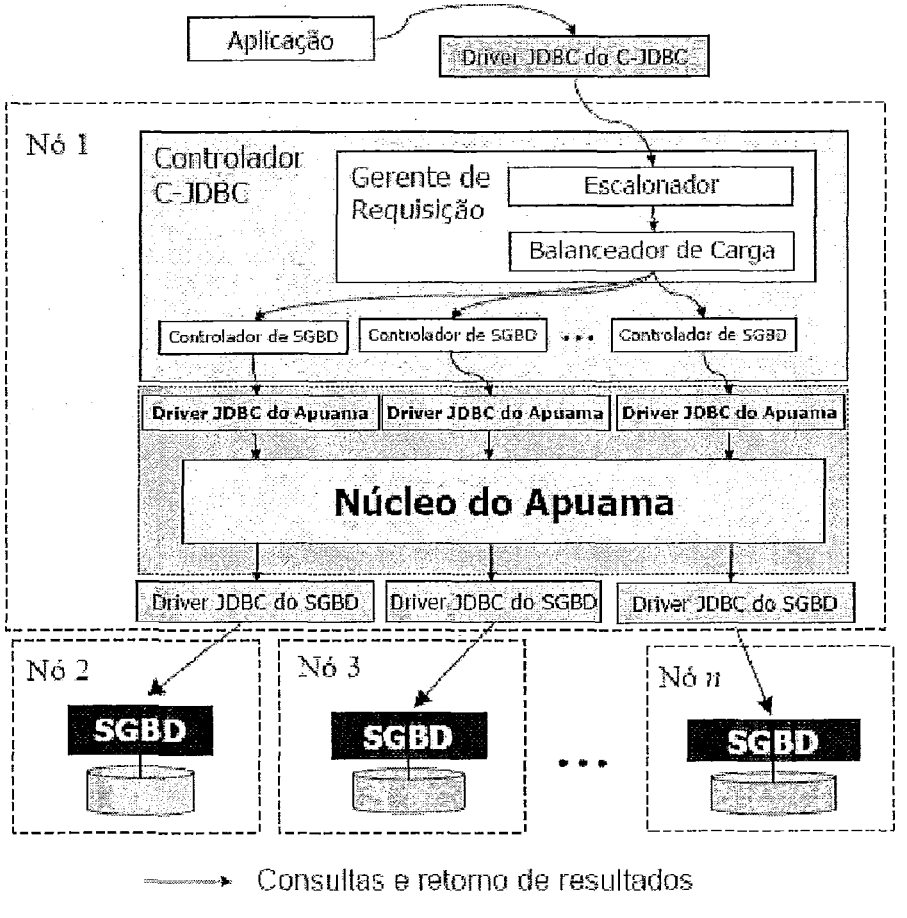

Figura 4 – Apuama como extensão do C-JDBC

Na Figura 4, podemos observar como o Apuama estende o C-JDBC. O Apuama não precisa que qualquer mudança seja feita no código-fonte do C-JDBC. A decisão de não alterar o código-fonte do C-JDBC visa não criar dependência entre o Apuama e uma versão específica do C-JDBC. Fazendo esta extensão de forma não intrusiva, permitimos que o Apuama seja utilizado com qualquer versão futura do C-JDBC.

O **núcleo do Apuama** corresponde a uma camada de software intermediária entre o C-JDBC e os SGBDs. É por meio do núcleo do Apuama que são implementadas as técnicas de paralelismo intra-consulta adicionadas ao C-JDBC. A requisições entre o C-JDBC e os SGBDs são monitoradas e repassadas pelo Apuama. Quando decide por processar uma consulta por paralelismo intra-consulta, o Apuama intervem no fluxo de execução da consulta. Após ser estendido pelo Apuama, o C-JDBC não mais faz conexões diretas aos SGBDs. Cada controlador do SGBD conecta-se ao Apuama usando um **driver JDBC do próprio Apuama**. É o Apuama que conecta o C-JDBC aos SGBDs.

IV.2 Arquitetura Interna do Apuama

A Figura 5 mostra a arquitetura detalhada do Apuama. O *driver* JDBC do Apuama é o ponto de entrada das requisições enviadas pelo C-JDBC. Quando o processo do controlador do C-JDBC é iniciado, o controlador cria um reservatório de conexões para cada SGBD do agrupamento de PCs. Como o Apuama é o mediador entre o C-JDBC e os SGBDs, o *driver* JDBC do Apuama cria um objeto de conexão falsa para o C-JDBC. Este objeto nada mais é do que um representante da conexão com o SGBD. O **processador de nó** é o componente que media o objeto de conexão falsa do *driver* JDBC do Apuama e a conexão real do SGBD. A conexão real do SGBD será criada e manipulada por um componente chamado de **executor de consulta**. Para estar habilitado a processar múltiplas requisições, o executor de consulta possui um repositório de conexões. Se mais de uma conexão daquele SGBD for solicitada pelo C-JDBC, um novo objeto de conexão falsa será ligado ao processador de nó já criado. O processador de nó apenas analisa e trata as requisições SQL recebidas, todas as outras requisições (como por exemplo: consultas ao catálogo do SGBD, solicitação de informação de versão do SGBD) enviadas à conexão falsa são simplesmente redirecionadas à conexão real.

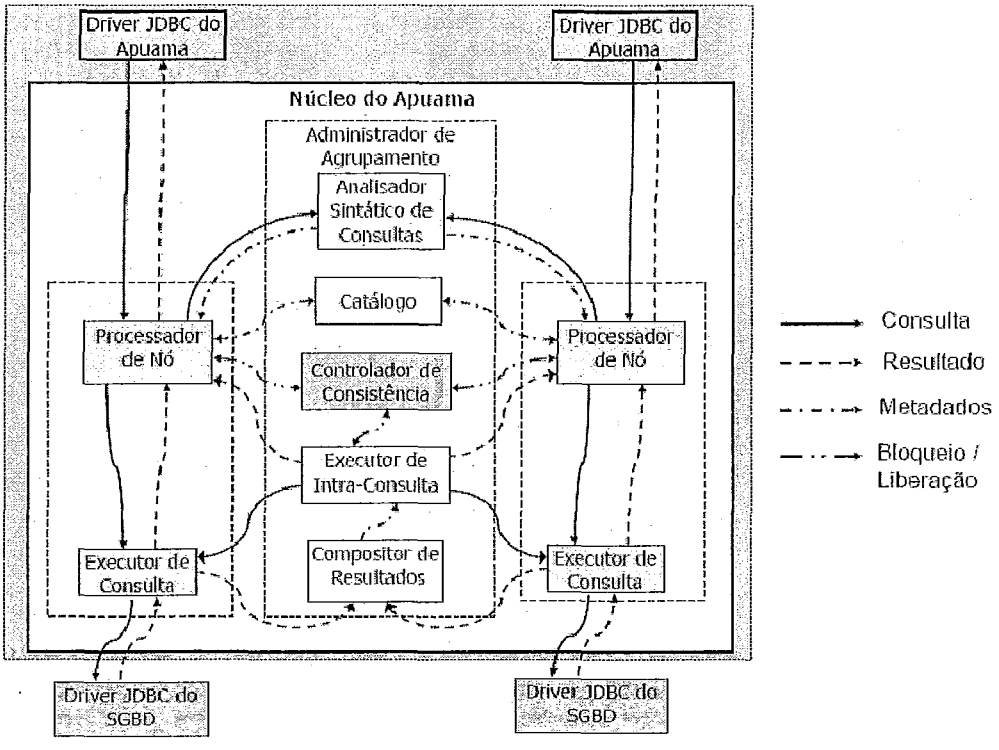

Figura 5 – Visão da arquitetura intera do Apuama

Para analisar as requisições recebidas, o processador de nó tem o auxílio de um componente que tem a visão global de todos os processadores de nó. Este componente é chamado de **administrador do agrupamento**. O administrador de agrupamento é criado pelo primeiro processador de nó instanciado e uma referência a ele é passada a cada processador criado em seguida. Ele tem a função de coordenação dos nós. Para analisar as consultas, o administrador do agrupamento possui dois componentes: o analisador sintático de consultas e o executor de intra-consulta. Com base na sintaxe da consulta recebida, o componente **analisador sintático de consultas** é capaz de informar quais tabelas são referenciadas e como são relacionadas pela consulta. Este componente também é utilizado para fazer a tradução da consulta em subconsulta SVP. O componente **catálogo** contém a lista de quais tabelas são virtualmente fragmentadas. As tabelas virtualmente fragmentadas são aquelas que possuem a definição de um índice agrupado sobre seu VPA. Adicionamente, o catálogo informa qual é o VPA da tabela virtualmente fragmetada e armazena os valores máximo e mínimo para aquele VPA.

A execução do processamento por paralelismo intra-consulta fica a cargo dos componentes executor de intra-consulta e compositor de resultados. O **executor de intra-consulta** é o componente responsável por coordenar globalmente a execução de cada subconsulta SVP. O **compositor de resultados** é usado no final do processamento de cada consulta SVP para armazenar os resultados parciais e, em seguida, para processar o resultado final.

A consistência entre réplicas prevista pelo Apuama é assegurada pelo componente **controlador de consistência**. Este componente auxilia a execução das subconsultas SVP pelo processador de nó em um estado consistente. Para ter controle sobre o estado de cada réplica da base de dados, o controlador de consistência recebe uma notificação para cada requisição de atualização recebida por um processador de nó.

Para cada consulta de leitura recebida, o Apuama deve decidir se será processada por paralelismo intra-consulta ou não. Esta decisão é baseada na classificação de tipos de consultas definida na seção III.2. Apenas são processadas as consultas do tipo 2. As outras consultas são redirecionadas ao SGBD que o C-JDBC selecionou para processá-la. Para esclarecer a interação entre os componentes internos do Apuama, a seguir ilustramos seu fucionamento por meio de um exemplo.

IV.2.1 Exemplo de Processamento de Consulta no Apuama

O C-JDBC recebe uma consulta de leitura, seleciona o SGBD candidato a processá-la e envia esta consulta ao driver *JDBC* do Apuama correspodente. Neste exemplo, vamos considerar que a consulta enviada é a consulta Q6 do TPC-H listada abaixo:

```
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 0.06 - 0.01
   and 0.06 + 0.01 and l_quantity < 24
```

O processador de nó, ao receber a consulta, questiona o analisador sintático a fim de determinar se a consulta é de leitura ou escrita. Caso seja de leitura, também devem

ser informadas quais são as tabelas envolvidas e qual é o relacionamento entre elas. Esta consulta referencia apenas a tabela *lineitem* e, de acordo com a classificação de consultas paralelizáveis, sabemos que podemos processá-la por paralelismo intra-consulta. O catálogo do administrador de agrupamento tem a função de informar se a consulta *lineitem* é fragmentada virtualmente. Dada estas informações, o processador de nó passa o controle do processamento da consulta ao administrador de agrupamento. O administrador de agrupamento delega a responsabilidade de coordenação do processamento por paralelismo intra-consulta ao executor intra-consulta. Este usa o analisador sintático de consultas para adicionar os predicados à consulta original com a finalidade de gerar as subconsultas da SVP. A resposta do analisador sintático para a consulta Q6 seria a seguinte:

```
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 0.06 - 0.01 and 0.06 + 0.01
        and l_quantity < 24
        and l_orderkey >= ? and l_orderkey < ?
```

A notação *?* representa os parâmetros de VPA que ainda devem ser adicionados à subconsulta SVP. O catálogo provê a informação de quais são os limites máximo e mínimo do VPA de cada tabela. Dado o número de processadores de nó existentes, são atribuídos intervalos de VPA de igual tamanho a cada processador de nó. O executor de intra-consulta faz a substituição dos parâmetros das subconsultas SVP a serem enviadas aos nós.

Quando a subconsulta SVP está pronta para ser processada, é enviada em ao executor de consulta de cada processador de nó. O executor de consultas é responsável por enviar a subconsulta SVP ao SGBD correspondente e aguardar a resposta. Para cada nova subconsulta SVP enviada, o executor de consulta solicita ao repositório de conexões uma conexão do SGBD. Esta conexão é devolvida ao repositório ao final do processamento. Antes de enviar a subconsulta SVP, o executor de consulta precisa desabilitar a varredura seqüencial da tabela, necessário para garantir o uso eficiente da

SVP. O catálogo do Apuama armazena e provê os comandos específicos de cada SGBD para essa finalidade.

Enviada a subconsulta SVP, o executor de consultas aguarda o envio do resultado pelo SGBD. Quando o resultado é recebido, o executor de consultas o repassa ao executor de intra-consulta, que por sua vez efetua o redirecionamento ao compositor de resultados. Como já foi dito anteriormente, usamos o SGBD em memória conhecido como HSQLDB. Após o primeiro resultado recebido, o compositor de resultados cria uma tabela responsável por acumular os resultados parciais. No caso da consulta Q6, a criação da tabela temporária seria feita pelo seguinte comando SQL:

```
create table table_temp_1 (revenue float)
```

Para cada registro de cada resultado parcial, seria executado o seguinte comando:

```
insert into table table_temp_1 values (?)
```

A notação *?* indica os valores dos campos de resultado parcial. Quando todos os resultados parciais são coletados, o executor de intra-consulta solicita ao compositor de resultados a produção do resultado final. A composição do resultado é feita por meio de um comando SQL enviado ao HSQLDB. O comando também é gerado pelo analisador sintático. No caso da consulta Q6, o comando SQL gerado seria:

```
select sum(revenue) as revenue
from table_temp_1
```

Em seguida, o resultado final é enviado ao processador de nó que, por sua vez, responde à aplicação cliente. Desde que as consultas sejam paralelizáveis, o compositor de resultados pode compor qualquer resultado parcial. A única limitação existente é o tamanho do resultado final. Como o HSQLDB funciona em memória, todo seu trabalho é efetuado sem usar estruturas em disco. Se o tamanho do resultado final superar o tamanho existente de memória RAM, o HSQLDB não pode montar o resultado final. É enviada uma mensagem de erro para o executor de intra-consulta, que, por sua vez, cancela a execução da consulta.

IV.2.2 Limitações do Analisador Sintático

Na implementação atual do Apuama, o analisador sintático não é automático. Isto significa que as possíveis solicitações de análise e as devidas respostas devem ser carregadas manualmente e previamente pelo administrador do DBC. Esta informações estão fixadas em um arquivo XML de configuração do Apuama. Este contém a informação do comando SQL a ser analisado, o comando SQL da subconsulta SVP e o comando SQL necessário para agregação de resultados. O fragmento XML relativo a cada consulta paralelizável é da seguinte forma:

```
<sqlTransformation>
        <sqlOriginal>
                <![CDATA[ "comando SQL da consulta paralelizável" ]]>
        </sqlOriginal>
        <sqlTransformated>
                <![CDATA[ "comando SQL da subconsulta SVP
                        correspondente" ]]>
        </sqlTransformated>
        <sqlGlobalAggregator>
                <![CDATA[ "comando SQL da consulta de agregação" ]]>
        </sqlGlobalAggregator>
</sqlTransformation>
```

Se o catálogo fizer uma busca neste arquivo XML e não encontrar o comando SQL da consulta a ser analisada, o catálogo informa que esta consulta não é paralelizável.

A substituição deste processo manual pela de análise automática de consultas está planejada para desenvolvimentos futuros do Apuama. O projeto ParGRES, que também utiliza VP para processamento de consultas, já possui um analisador automático de consultas. Como ambos projetos são de software livre, o desenvolvimento futuro do Apuama poderia ter como base o progresso atual já feito no analisador sintático do ParGRES.

IV.2.3 Funcionamento do Controle de Consistência entre Réplicas

O executor de intra-consulta também controla a consistência entre as réplicas para processamento por paralelismo intra-consulta. O executor de intra-consulta só submete a subconsulta SVP quando a réplica possuir o mesmo estado de consistência estipulado.

O estado de uma réplica é definido por um contador incremental que registra cada atualização submetida ao SGBD. O incremento do contador é feito pelo processador de nó. O contador é armazenado pelo controlador de consistência. Quando o executor intra-consulta recebe uma requisição para processar uma consulta paralelizável, solicita o estado de todos os processadores de nó. Os estados são comparados e é selecionado aquele de maior valor. A seleção do maior valor indica qual nó possui a base com as atualizações mais recentes. A subconsulta SVP será executada apenas sobre um SGBD apenas quando este atingir o estado consistência. Para os SGBDs que não atingiram o estado de consistência, o executor de intra-consulta permite que o seu processador de nó execute todas as atualizações até que atinja o estado desejado. Após atingir este estado, todas as novas atualizações são bloqueadas pelo processador de nó. O executor de intra-consulta, então, envia a subconsulta SVP ao executor de consultas. Em seguida, as atualizações bloqueadas para aquele SGBD são liberadas. Como a execução subconsulta SVP em um SGBD é independente da execução em outros SGBDs, as subconsultas são liberadas assincronamente.

Deve-se ter dado um cuidado especial com a liberação de atualizações bloqueadas, pois, sem este cuidado, o mecanismo definido não seria suficiente para controlar a consistência entre as réplicas. Os diferentes fluxos de execução da solicitação de atualização e da subconsulta SVP podem gerar um problema conhecido como "*race condition*". A situação de *race condition* específica do Apuama foi apresentado na seção III.4.2 e é verificado quando um resultado inesperado ocorre dependendo da velocidade de execução de cada fluxo de execução. Quando o estado de consistência é atingido, é liberado primeiro o fluxo de execução da subconsulta SVP e depois o fluxo de execução das atualizações. O problema é que não existe nenhuma garantia de que os fluxos manterão a ordem de execução. Apesar de ser liberado posteriormente, o fluxo de execução de atualização pode acabar solicitando a atualização ao SGBD antes do início do processamento da subconsulta SVP. Este seria um o resultado inesperado. Para não ocorrer esta situação, o Apuama só libera as atualizações após a subconsulta SVP iniciar no SGBD uma transação de leitura. Alterando o estado de isolamento de consultas para leitura repetitiva (padrão de isolamento *repeatable read* definido pelo padrão SQL-92), o SGBD garante que após o início da transação o estado da base não será modificado pela execução de transações concorrentes. Com este recurso, o efeito do *race conditions* é eliminado.

IV.3 Conclusão

Neste capítulo mostramos como o Apuama e o C-JDBC compõem uma solução única de DBC. O Apuama estende o C-JDBC de forma não instrusiva, sendo conectados exclusivamente por uma interface JDBC. Sendo assim, o desenvolvimento destes dois componentes é independente. O uso do Apuama não está ligado a nenhuma versão específica do software do C-JDBC.

Foram apresentados os componentes utilizados na contrução do Apuama, suas finalidades e como interagem entre si. Na versão atual de nossa solução, os componentes relacionados diretamente com a análise de consultas não são completamente automatizados. É necessário que o administrador do DBC publique no catálogo do Apuama informações relativas às consultas que serão processadas por paralelismo intra-consulta. Para desenvolvimento futuros estão previstas contribuições com o projeto ParGRES, que já possui componentes de análise de consultas automatizados.

V Experimentos

Neste capítulo avaliamos a capacidade de processamento paralelo intra-consulta adicionado ao C-JDBC pelo Apuama em diferentes cenários. Executamos experimentos baseados no *benchmark* TPC-H. O Apuama foi submetido a cargas de trabalho de alto nível de concorrência, mesmo enquanto executava operações de atualização da base de dados.

Na seção V.1 é feita uma discussão sobre o que deve ser avaliado nos experimentos com o Apuama. A seção V.2 apresenta o *benchmark* TPC-H. Na seção V.3 descrevemos o ambiente do agrupamento em que os experimentos foram conduzidos. As seções em seguida analisam os experimentos feitos. A seção V.4 apresenta resultados de aceleração das consultas isoladas. Na seção V.5 são relatados testes de vazão usando o Apuama. O último experimento é avaliado na seção V.6, em que são analisados experimentos de vazão de consultas em conjunto com atualizações concorrentes.

V.1 O que avaliar?

A proposta do Apuama é oferecer uma solução de DBC de baixo custo para o processamento de consultas OLAP que atenda simultaneamente a muitos clientes e permita atualizações concorrentes. Deve ser livre de qualquer controle rígido de submissão de consultas e atualizações. Geralmente a infra-estrutura de DW está dedicada a poucos clientes (baixa concorrência de transações) e apenas prevê transações de atualização quando o sistema está ocioso ou indisponível para consultas. Por meio do Apuama esperamos que sistemas de suporte à decisão possam efetuar consultas OLAP concorrentes sobre dados recentemente atualizados.

Para avaliar nossa proposta, o Apuama foi submetido a experimentos que avaliam a capacidade de processamento de consultas OLAP nas seguintes situações:

- processamento de consultas OLAP sem concorrência;
- processamento de consultas OLAP concorrentes;
- processamento de consultas OLAP com requisições de atualização concorrentes.

O processamento de consultas leves – típicas do OLTP – não foi testado porque o Apuama não interfere no processamento por paralelismo inter-consulta do C-JDBC. A

eficiência para aplicações OLTP experimentada anteriormente em publicações do C-JDBC é mantida (detalhes em CECCHET *et al.* (2004)). O paralelismo inter-consulta é apenas empregado em teste de comparação com o paralelismo intra-consulta para o processamento de consultas OLAP.

Para cada experimento é medido o tempo de execução da consulta ou do lote de consultas executadas no DBC. No caso de lote de consultas, o volume de requisições e a quantidade de nós do agrupamento envolvidos são modificados em cada execução dos testes para serem obtidas as propriedades de aceleração e escalabilidade de vazão. Através destas propriedades medimos se o Apuama é capaz de atender a seu propósito.

Estão fora do escopo deste trabalho experimentos que combinam processamento simultâneo de consultas pesadas (OLAP) e leves (OLTP) e testes escalabilidade que considere o aumento da base de dados.

V.2 Benchmark

Com o objetivo de efetuar experimentos com o Apuama que se aproximem da carga de trabalho de um ambiente real de DW, nossa avaliação foi baseada na especificação do *benchmark* TPC-H. Detalhes sobre a especificação podem ser consultados no Apêndice I.

Neste trabalho usamos um subconjunto das 22 consultas definidas pelo TPC-H. São consultas que podem ser paralelizadas de acordo com a classificação sugerida na seção III.2. Assim como na especificação, identificamos as consultas pelos seus números. As consultas são as seguintes: Q1, Q3, Q4, Q5, Q6, Q12, Q14, Q21. Escolhemos estas consultas porque possuem diferentes complexidades. A Q1 acessa apenas a tabela *lineitem* e desempenha muitas operações de agregação. O predicado "*where*" de Q1 não é muito seletivo, pois em torno de 99% das tuplas são recuperadas. Esta é uma consulta muito custosa. Q3 faz a junção das tabelas *lineitem* e *orders* e uma tabela de dimensão. Diferentemente de outras consultas, o seu resultado contém um grande número de tuplas. Q4 contém uma referência à tabela *lineitem* e uma subconsulta com outra referência a *lineitem*. Q5 faz a junção de *lineitem* e *orders* e quatro tabelas de dimensão. Apenas uma operação de agregação é feita. Assim como Q1, Q6 acessa apenas a tabela *lineitem*. As principais diferenças entre elas é que Q6 tem apenas uma operação de agregação e seu predicado "*where*" é muito mais seletivo, recuperando

apenas 1,5% das tuplas. Q12 faz a junção das tabelas *lineitem* e *orders* e tem duas operações de agregação. Q14 faz a junção da tabela *lineitem* e uma tabela de dimensão. Q21 contém três referências à tabela *lineitem*. Duas destas referências fazem parte de subconsultas.

No Apêndice II, encontramos a listagem da sintaxe do comando SQL de cada uma das consultas usadas. Juntamente com as consultas também é apresentado o comando SQL correspondente de cada subconsulta gerada pelo Apuama e o comando SQL usado para agregar os resultados parciais.

Os resultados dos experimentos mostram os tempos de execução das consultas e as taxas de vazão para um número crescente de nós (de 1 a 32). Cada execução foi repetida cinco vezes e a métrica final é o valor médio obtido dessas execuções, não considerando a primeira. Os dados em *cache* do sistema operacional e do SGBD estabilizam já na segunda execução. O tempo de execução na primeira repetição é superior ao tempo das repetições subseqüentes porque os sistemas de *cache* ainda não oferecem grande contribuição para melhora de desempenho. A primeira execução é instável e por esta razão dificilmente é mensurada com precisão. Medindo o tempo de execução muitas vezes nos leva a um cenário de *cache* estável, em que o tempo de execução do experimento pode ser obtido com precisão. O problema de medirmos apenas o *cache* estável é estar considerando uma situação que pode não corresponder à realidade. No uso diário de um DW não há nenhuma garantia de que o *cache* esteja estável antes de executar uma consulta.

Em alguns casos, as métricas são normalizadas dividindo seu valor pelo valor obtido durante o mesmo experimento usando apenas um nó. Para facilitar a leitura e análise, os valores são apresentados na escala logarítmica, que oferece uma noção clara de linearidade (CROWL, 1994).

V.3 Ambiente de Execução

Executamos experimentos sobre um agrupamento de 32 nós com arquitetura de memória distribuída do grupo *Paris* de INRIA (PROJECT, 2005). Cada nó tem dois processadores Opteron de 2.2 Ghz com 2 GB de memória RAM e 30 GB de espaço livre em HD local. Os nós são interconectados por uma rede *ethernet* Gigabit. Uma instância do PostgreSQL 8.0.1 é executada em cada nó. O tamanho total da base de

dados em disco usando fator de escala 5, incluindo todas as tabelas e índices, é em torno de 11 GB. Usando este fator de escala, a cardinalidade das tabelas de fato *lineitem* e *orders* é de $30x10^6$ e $7{,}5x10^6$, respectivamente. A VP é utilizada sobre as tabelas de fato. As chaves primárias são escolhidas como VPA e índices agrupados são associados a elas. Índices também são gerados para chaves primárias das tabelas de dimensão e chaves estrangeiras de todas as tabelas. Nenhum outro índice foi criado. Como o TPC-H presume consultas *ad-hoc*, não fazemos qualquer outra otimização no esquema físico. O HSQLDB é usado para computar o resultado final.

V.4 Experimentos de Consultas Sem Concorrência

O primeiro experimento avalia a aceleração obtida com o Apuama executando consulta isoladas em diferentes configurações do agrupamento. Apesar do objetivo mestre do Apuama ser atender a alto volume de requisições OLAP concorrentemente com atualizações, o teste de execução de consultas OLAP isoladas é preliminar e necessário para validar esta implementação da SVP.

A Tabela 3 mostra os tempos de execução obtidos. A média do tempo de execução seqüencial (1 nó) das consultas é de quase 5 minutos. Com o uso do paralelismo do agrupamento, o tempo de execução dessas consultas cai para, em média, aproximadamente 4 segundos usando 32 nós. A consulta Q4 é a que obteve a melhor redução de tempo. O seu tempo de execução seqüencial é de pouco mais de 5 minutos. Já com 4 nós há uma forte redução no tempo de execução para 3,85 segundos. Com 32 nós a consulta Q4 reduz seu tempo para menos de 1 segundo. A consulta Q1 é a que obteve a aceleração mais leve. O tempo de execução seqüencial é quase 6 minutos. Mesmo assim, com 32 nós, o seu tempo de execução é de menos de 10 segundos.

		Q1	*Q3*	*Q4*	*Q5*	*Q6*	*Q12*	*Q14*	*Q21*	*Média*
Número de Nós	*1*	355.70	284.41	316.68	255.77	189.24	290.25	198.17	418.22	288.56
	2	180.71	166.26	171.25	137.43	100.90	155.06	107.76	214.91	154.29
	4	75.63	104.12	3.85	111.23	11.70	91.65	75.09	128.23	75.19
	8	36.65	10.41	2.00	11.67	5.97	8.41	6.78	35.89	14.72
	16	18.59	7.58	1.07	3.38	3.13	4.38	3.39	18.60	7.51
	32	9.38	4.96	0.55	2.05	1.55	2.16	1.74	9.42	3.98

Tabela 3 – Tempo de execução de consultas em segundos em cada configuração do agrupamento

Para melhor analisar comparativamente a aceleração das consultas, vamos observar o gráfico da . Este gráfico apresenta os tempos de execução normalizados. Isto

significa que todos os tempos apresentados correspondem à razão entre o tempo medido e o tempo de execução seqüencial da consulta, possibilitando a análise de aceleração. Os valores obtidos foram multiplicados por 128 para facilitar a leitura do gráfico. Com 2 nós, o tempo de execução de todas as consultas é reduzido para quase 50% quando comparadas com o tempo de execução com apenas um nó. Com 4 nós, o tempo de execução da consulta é diminuído para 30% a 50% em todas as consultas, com exceção da consulta Q4 que diminuiu para 1.23% do tempo original. De 8 a 32 nós, os fragmentos obtidos pela VP são suficientemente pequenos para caber na memória RAM disponível, resultando em aceleração superlinear. Para essas configurações podemos observar que, depois da primeira execução, nenhum acesso a disco é feito.

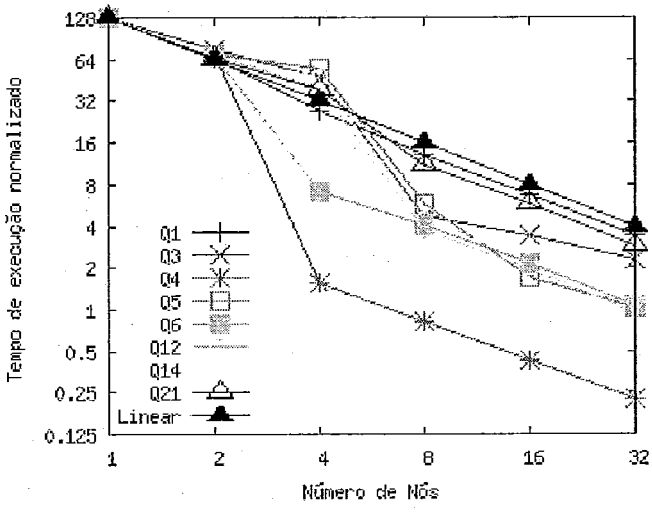

Figura 6 – Experimentos de aceleração de consultas – tempo de execução normalizados

Podemos concluir que o Apuama estende com sucesso o C-JDBC para reduzir o tempo de consultas OLAP. Com o emprego da técnica SVP a aceleração é quase linear para todas as configurações do agrupamento sendo superlinear na maioria das situações. Como não usamos lotes de consultas neste experimento, a escalabilidade de vazão não é medida.

Neste experimento, era esperada aceleração linear para as consulta Q1, Q4, Q6, Q12 por possuírem apenas referências a tabelas que foram fragmentadas virtualmente. As consultas Q3, Q5, Q14 e Q21, por possuírem referências a tabelas que não foram fragmentadas, deveriam ter aceleração apenas próxima do linear. No entanto, isto não é

precisamente o que foi constatado. O efeito do fragmento virtual todo em memória leva todas as consultas a uma aceleração superlinear. Além disso, a presença de mais referências a tabelas não fragmentadas não determina claramente que uma consulta terá menor aceleração de que outra consulta que referencia apenas tabelas virtualmente fragmentadas. A seguir analisamos com mais profundidade o comportamento de cada uma das consultas OLAP.

V.4.1 Análise de Desempenho Detalhado das Consultas

Nesta seção analisamos em detalhe o plano de execução e o uso de recursos computacionais por parte de cada uma das consultas. Como o processamento da consulta em cada SGBD é representado pela subconsulta SVP, selecionamos SGBD do agrupamento de PCs e monitoramos sua execução. Os gráficos apresentados mostram dados obtidos pelas estatísticas internas do PostgreSQL e pela ferramenta *vmstat* (VMSTAT, 2005) que, além de outras informações, coleta dados de E/S do nó. Neste trabalho as estatísticas do PostgreSQL são acessadas por meio da visão *pg_statio_all_tables*. Neste trabalho usamos os seguintes dados desta visão:

- *heap_blocks_hit*: blocos de dados lidos no *cache* do SGBD ;
- *heap_blocks_read*: blocos de dados solicitados ao sistema operacional;
- *index_blocks_hit*: blocos de índice encontrados no *cache* do SGBD;
- *index_blocks_read*: blocos de dados solicitados ao sistema operacional;
- *total_blocks_read*: total de blocos solicitados ao sistemas operacional, ou seja, corresponde a soma de *heap_blocks_read* e *index_blocks_read*.

O tamanho do bloco do PostgreSQL é de 8 kb. A informação apresentada nos gráfico mostra o bloco com 1 kb, o que corresponde a 8 vezes o valor obtido nas estatísticas. Isto é feito para normalizar com o tamanho do bloco de 1kb medidos pelo *vmstat* nas operações de E/S do sistema operacional. No gráfico, os dados obtidos pelo *vmstat* são os seguintes:

- *disk_blocks_read*: blocos solicitados ao disco pelo sistema operacional;
- *disk_blocks_write*: blocos escritos no disco pelo sistema operacional;

Note que a diferença entre os blocos solicitados ao sistema operacional (*total_blocks_read*) e os blocos efetivamente lidos em disco (*disk_blocks_read*) corresponde aos blocos que foram encontrados no *cache* do sistema operacional no momento da execução da subconsulta SVP. Por conta da diminuição do tamanho do

fragmento virtual com o uso de uma quantidade maior de nós, é esperada uma diminuição linear ou quase linear de blocos lidos pelo sistema operacional (*total_blocks_read*).

Nos gráficos a seguir podemos observar que o *cache* do sistema operacional muitas vezes é responsável direto pela aceleração superlinear da consulta OLAP. Apesar de não estarmos analisando o desempenho da transação de atualização, o SGBD efetua operações de escrita em disco durante o processamento das consultas. Isto está relacionado à construção de estruturas temporárias de dados do PostgreSQL necessárias para execução de algumas consultas.

Os gráficos são apresentados na escala logarítmica e o eixo vertical indica a quantidade de blocos usando potência na base 2.

Subconsulta SVP de Q1

O plano de consulta da subconsulta SVP de Q1 é o mesmo para todas as configurações de agrupamento. A única tabela de fato envolvida é a *lineitem* e esta é lida com o uso do índice agrupado. O gráfico da Figura 7 mostra o uso de recurso computacional pela subconsulta SVP. Para processar a subconsulta SVP, o PostgreSQL solicita aproximadamente 6 Gb em blocos ao sistema operacional (em torno de $1{,}5*2^{22}$ blocos). A quantidade de blocos solicitados decresce linearmente até atingir pouco mais de 191 mb em blocos com o uso de 32 nós. Isto demonstra o uso efetivo do índice agrupado para executar a subconsulta SVP. Com 4 nós, a quantidade de blocos solicitados passa para 1,5 Gb e por conta disto a curva de *disk_blocks_read* mostra que o *cache* do sistema operacional começa a ter influência no desempenho da subconsulta SVP. O tamanho de 1,5 Gb já é menor do que a quantidade de memória RAM disponível (quase 1.6 Gb). No entanto, observamos que o tempo de execução não cai bruscamente com o uso de 4 nós. Como a subconsulta SVP possui uma grande quantidade de agregações, o seu tempo de processamento é limitado pelo processador (*cpu-bound*). Durante a execução da subconsulta SVP, verificamos que a ocupação de um processador é quase máxima (cada nó do agrupamento possui 2 processadores). Todos estes fatores associados justificam a aceleração próxima a linear da consulta Q1 para todas as configurações do agrupamento.

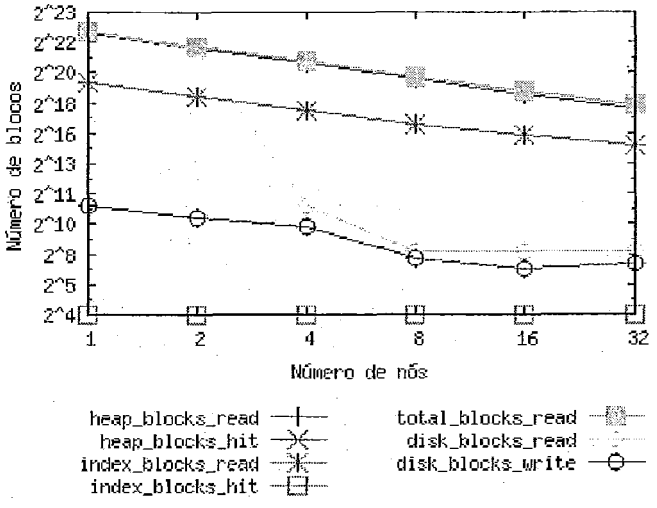

Figura 7 – O uso de recursos computacionais pela subconsulta SVP de Q1

Subconsulta SVP de Q3

O plano de consulta da subconsulta SVP de Q3 é o mesmo quando o agrupamento tem entre 2 e 8 nós (plano *i*), mas é diferente com 16 (plano *ii*) e com 32 também (plano *iii*). Participam dessa consulta as tabelas *lineitem*, *orders* e *customer*. Ambas as tabelas de fato são lidas por meio do índice agrupado em todos os planos gerados. Através do gráfico da Figura 8 podemos constatar que o uso do índice agrupado garante a redução linear de blocos de dados solicitados pelo PostgreSQL (*heap_blocks_read*). A variação entre os planos de consulta está basicamente nas operações de junção entre as tabelas. No plano *i*, a junção das tabelas é feita por *merge join*, para depois fazer *hash join* entre o resultado parcial e a tabela *customer*. Enquanto isso, no plano *ii*, as tabelas *orders* e *customer* sofrem operação de *hash join* para então depois ser feito o *merge join* com *lineitem*. Já no plano *iii*, a diferença em relação ao plano *ii* é o uso do *nested loop join* ao invés de *merge join*. Estas diferenças têm efeito direto na construção de resultados parciais e, conseqüentemente, nas operações de escrita. De 2 a 8 nós, a curva *disk_blocks_write* é quase linear. Com 16 nós, as operações de escrita são reduzidas em 8 vezes. O volume de escrita é desprezível com 32 nós.

O gráfico mostra que há uma queda brusca do número de operações de leitura em disco (*disk_blocks_read*) quando passamos a usar 8 nós. Fica evidente que isto proporciona aceleração, pois dobrando a quantidade de 4 nós temos um tempo de execução aproximadamente 10 vezes menor. De 8 a 32 nós, a curva de aceleração é

menos acentuada, mas continuamos observando aceleração superlinear. Como a consulta Q3 tem baixa seletividade, o tempo de agregação dos resultados parciais (pouco mais de 1 segundo) torna-se significativo frente ao tempos de execução com 8, 16 e 32 nós (entre 11 e 4 segundos no total).

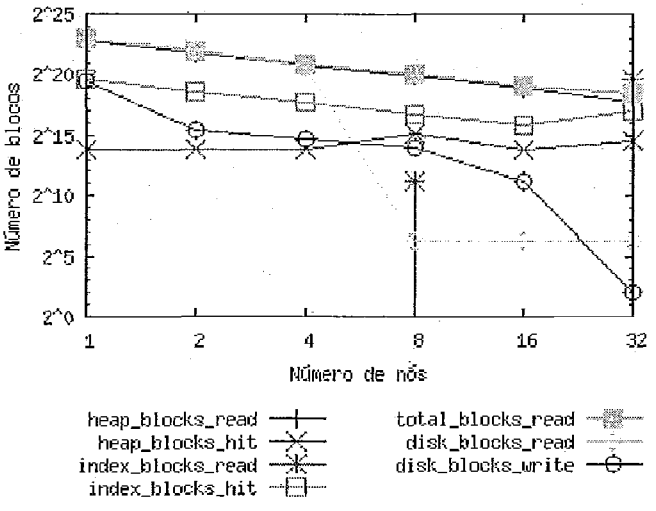

Figura 8 – O uso de recursos computacionais pela subconsulta SVP de Q3

Subconsulta SVP de Q4

A consulta Q4 envolve as tabela *orders* e possui uma subconsulta sobre a tabela *lineitem*. O plano de consulta da subconsulta SVP de Q4 é o mesmo para todas as configurações do agrupamento. Para cada tupla de *orders*, é executado a subconsulta sobre *lineitem*. Ambas são acessadas com o uso índice agrupado. O gráfico da Figura 9 mostra que o número de blocos de dados solicitados pelo PostgreSQL cai linearmente entre todas as configurações (aproximadamente 4Gb, 2Gb, 1Gb, 490Mb, 240Mb e 120Mb entre 1 e 32 nós). Com 4 nós a quantidade blocos solicitada é quase da mesma dimensão da memória RAM do nó, causando uma queda brusca nas operações de leitura feitas pelo sistema operacional (*disk_blocks_read*). Como Q4 executa uma subconsulta sobre *lineitem* para cada tupla lido de *orders*, o ganho de desempenho deve ser mais do que linear quando os fragmentos virtuais couberem em memória. Por esta razão, a aceleração da consulta Q4 quando passamos de 2 para 4 nós é enorme: mais de 44 vezes menor.

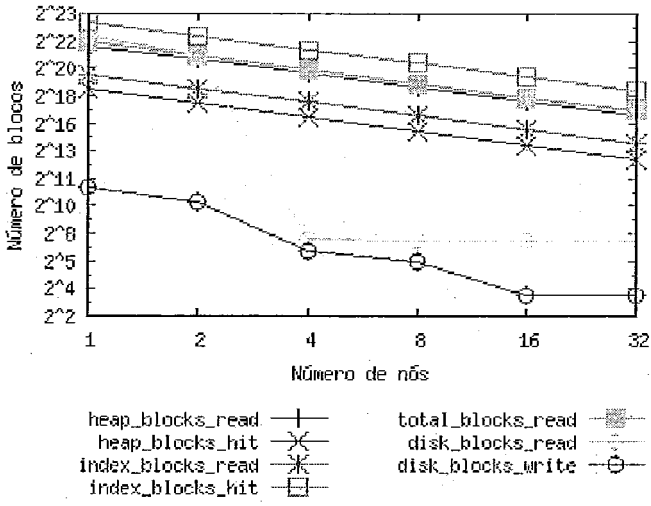

Figura 9 – O uso de recursos computacionais pela subconsulta SVP de Q4

Subconsulta SVP de Q5

A subconsulta SVP de Q5 é constituída por junções entre as tabelas *lineitem*, *orders*, *customer*, *supplier*, *nation* e *region*. O plano de consulta para a subconsulta SVP varia de acordo com a quantidade de nós do agrupamento. Em todos os casos *lineitem* e *orders* são lidos através do agrupamento de índice. A ordem em que as junções são feitas entre as tabelas varia entre as configurações do agrupamento. Com 2, 16 e 32 nós, a junção mais custosa é o *merge join* entre *lineitem* e o resultado da junção de *orders* com algumas tabelas de dimensão. Para 4 e 8 nós, a operação de *merge join* é feito diretamente sobre *lineitem* e *orders*, para depois então juntar com as tabelas de dimensão. Com 8 nós, a quantidade de blocos de leitura solicitados ao PostgreSQL (curva *disk_blocks_read* do gráfico da Figura 10) cai para menos de 2Gb, e a aceleração superlinear pode ser notada. Constatando as curvas *heap_blocks_hit* e *index_blocks_hit* para 8 e 16 nós observamos um grande aumento no aproveitamento do *cache* do PostgreSQL. Como resultado, o tempo de processamento entre 8 e 16 nós cai em aproximadamente 3 vezes. Entre 16 e 32 nós o tempo de processamento cai para um pouco menos da metade.

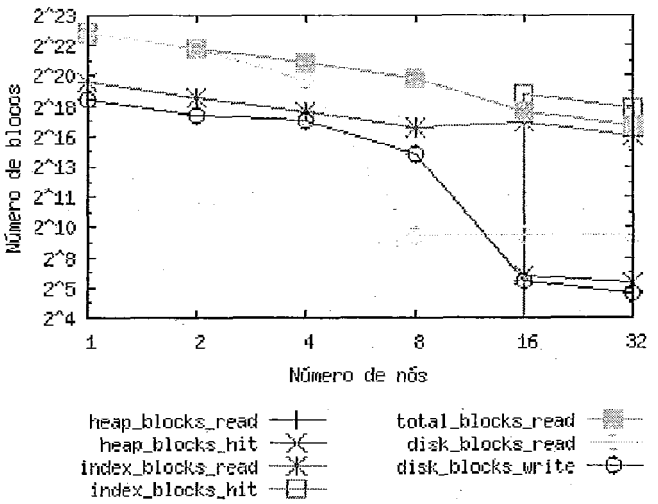

Figura 10 – O uso de recursos computacionais pela subconsulta SVP de Q5

Subconsulta SVP de Q6

A subconsulta SVP de Q6 é simples e faz apenas uma operação de agregação sobre a tabela de fato *lineitem*. As tuplas dessa tabela são acessadas pelo índice agrupado. O plano de consulta da subconsulta de Q6 é constante para todas as configurações do agrupamento. Por conta disto, podemos notar no gráfico da Figura 11 que a quantidade de blocos solicitados pelo PostgreSQL (*disk_blocks_read*) decresce proporcionalmente ao aumento de nós do agrupamento. Com quatro nós, a quantidade de blocos solicitados pelo PostgreSQL é pouco mais de 1.7 Gb. Este tamanho já cabe em memória e então há uma brusca queda na quantidade de blocos lidos em disco pelo sistema operacional quando passamos de 2 para 4 nós. Por conta destes fatores, a aceleração do tempo de execução da consulta Q6 é quase linear entre 1 e 2 nós e entre 4 e 32 nós.

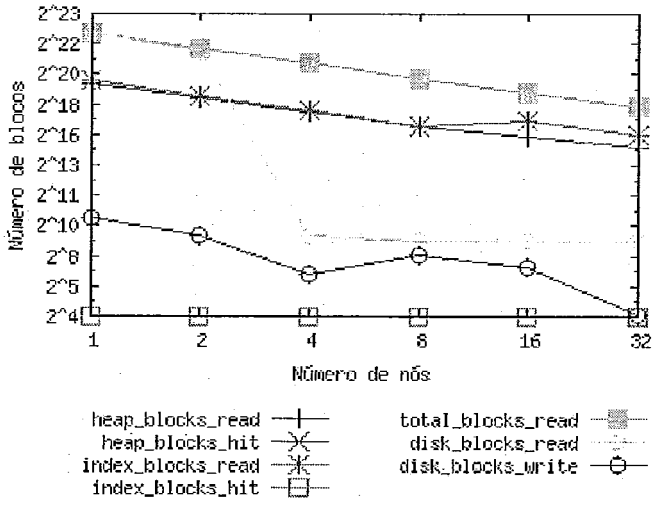

Figura 11 – O uso de recursos computacionais pela subconsulta SVP de Q6

Subconsulta SVP de Q12

A consulta Q12 é de alta seletividade e composta pelas tabelas de fato *lineitem* e *orders*. O plano de consulta gerado a partir de subconsulta SVP de Q12 é o mesmo para qualquer configuração do agrupamento. As tabelas são lidas pelo índice agrupado e a junção entre elas é feito por *merge join*.

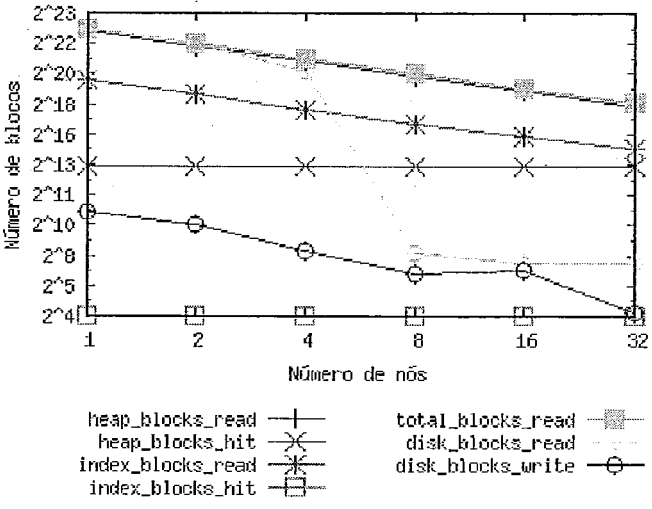

Figura 12 – O uso de recursos computacionais pela subconsulta SVP de Q12

O gráfico da Figura 12 mostra que a diminuição da quantidade de blocos solicitados pelo PostgreSQL é linear conforme esperado (*total_blocks_read*):

aproximadamente 8 GB, 4 GB, 2 GB, 1 GB, 535 Mb e 271 Mb. Com 2 nós, a quantidade mais precisa de blocos solicitados é 2.090 Mb, ligeiramente superior a quantidade de memória RAM disponível. Como 4 nós, ou seja, lendo 1 GB de dados, os fragmentos virtuais cabem em memória. Por esta razão, a aceleração do tempo de execução da consulta entre 4 e 8 nós é de mais de 10 vezes.

Subconsulta SVP de Q14

A consulta Q14 é composta da tabela de fato *lineitem* e da tabela de dimensão *part*. O plano de consulta da subconsulta SVP de Q14 mostra que a tabela *lineitem* é acessada via índice agrupado. O plano de consulta um para entre 2, 4 e 8 nós, e outro diferente para 16 e 32 nós. Nos planos de consultas para 2, 4 e 8 nós, as tabelas *lineitem* e *part* sofrem junção por *merge join*. Para 16 e 32 nós, a junção entre as tabelas é feita por *nested loop*. Essa mudança no plano de execução não é significativa para o desempenho porque a junção ocorre entre uma tabela de fato e uma tabela de dimensão. A cardinalidade da tabela *part* é muito inferior a *lineitem*. A redução de blocos solicitados (*total_blocks_read*), como pode ser visto no gráfico da Figura 13, é linear pelo menos até 32 nós. De 4 para 8 nós, a quantidade de blocos solicitados cai, aproximadamente, de 3.5 GB para 1.7 GB. A partir daí, o fragmento virtual é menor do tamanho da memória disponível, permitindo uma grande queda na quantidade de operações de leitura de disco feitas pelo sistema operacional (*disk_blocks_read*). A aceleração no processamento da consulta entre 4 e 8 nós é de pouco mais de 11 vezes.

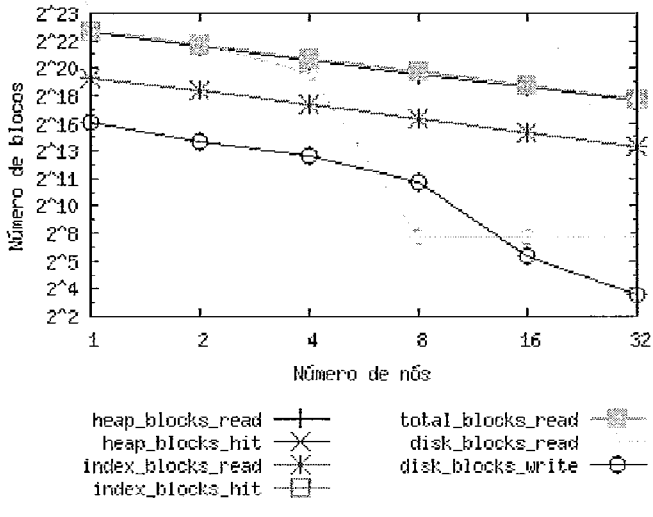

Figura 13 – O uso de recursos computacionais pela subconsulta SVP de Q14.

Subconsulta SVP de Q21

A consulta Q21 tem referências às tabelas de fato *lineitem* e *orders* e às tabelas de dimensão *supplier* e *nation* e a duas subconsultas. Cada subconsulta tem uma referência a tabela *lineitem*. Em ambos subplanos de consulta e para todas as configurações do agrupamento, a tabela *lineitem* é acessada pelo índice agrupado. Estas subconsultas são executadas para cada tupla da tabela *lineitem* externa. O plano de execução da subconsulta SVP de Q21 muda de acordo com a quantidade de nós envolvidos no agrupamento. Novamente, analisamos a operação de agregação entre as tabelas de fato. Para 2, 4 e 16 nós, *lineitem* e *orders* são juntos por *merge join*. Com 8 e 32 nós é feito junção por *neste loop join* entre *lineitem* e o resultado da junção de *orders* com algumas tabelas de dimensão.

Podemos ver na Figura 14, que, de 4 para 8 nós, há uma aceleração de pouco de mais 4,5 vezes no tempo de execução da consulta. Isto é devido a quantidade de blocos solicitados pelo PostgreSQL quando o agrupamento passa a usar 8 nós. Aproximadamente 820 MB de blocos podem ser armazenados no *cache* do sistema operacional. Curiosamente a redução de tempo de execução não é tão acentuada como vemos em outras consultas. Isto pode ocorrer devido ao uso intenso de *cache* do PostgreSQL que já é feito com 2 e 4 nós. Como a consulta Q21 tem três referências à tabela *lineitem*, muitos blocos solicitados para as três referência à tabela podem ser compartilhados pelo PostgreSQL.

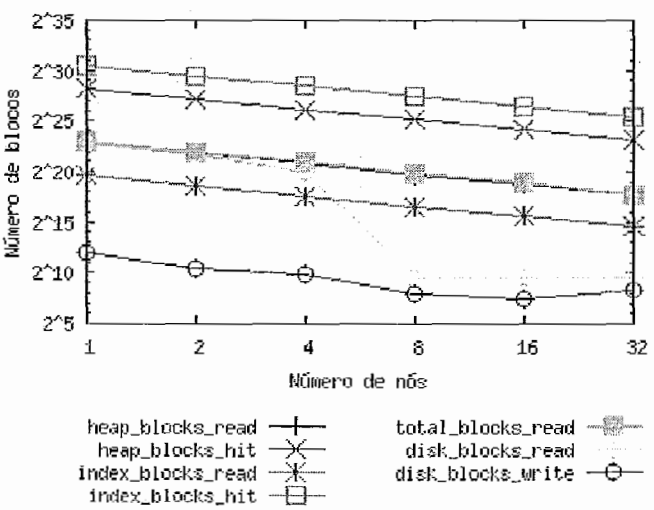

Figura 14 – O uso de recursos computacionais pela subconsulta SVP de Q21

Avaliação Geral

A respeito da análise detalhada da aceleração das consultas, existem algumas características encontradas:

- o plano de consulta pode mudar de acordo com o tamanho do fragmento virtual; isto pode alterar o padrão de aceleração das consultas;
- quando os fragmentos virtuais cabem em memória, existe uma grande aceleração das consultas por causa da ausência de operações de E/S;
- se o processamento da subconsulta for limitado pelo processador (*cpu-bound*), o efeito da ausência de operações de E/S não é notado na curva de aceleração;
- quando o tempo de execução da subconsulta é tão pequeno que fica na mesma escala de grandeza do tempo para composição dos resultados parciais, a curva de aceleração é atenuada;

V.4.2 Benefício da Estabilidade da SVP

O desempenho do disco de armazenamento na maioria das situações é o fator limitante de desempenho para aplicações de banco de dados. O tempo de acesso a um bloco de disco aproximadamente equivale à execução de milhões de instruções pelo processador. Por esta razão, o uso eficiente de arquitetura de *cache* implementado tanto por hardware (*cache L1, L2*) quanto por software (*cache* do sistema operacional e do SGBD) minimiza o atraso gerado pelo uso do disco. A técnica de VP utilizada neste trabalho também precisa que o uso do disco seja otimizado para que a aceleração de consultas OLAP seja notada. Na seção III.1 sugerimos desabilitar a escolha da varredura seqüencial de tabelas no otimizador de plano de consultas do PostgreSQL. Favorecer o uso do índice agrupado é crucial para a eficiência do processamento de consultas de alto de custo pelo Apuama. O gráfico da Figura 15 mostra a razão entre o tempo de execução da subconsulta SVP de cada consulta OLAP com e sem a varredura seqüencial habilitada. Habilitar a varredura seqüencial pelo otimizador de consultas não quer dizer que obrigatoriamente serão lidas sequencialmente, mas apenas que o otimizador tem opção de fazer esta escolha.

Geralmente a varredura seqüencial de toda a tabela é mais eficiente comparado a ler toda a tabela usando índice agrupado. Isto é verificado quando as consultas Q1, Q6,

Q12, Q14 são executadas com apenas 1 nó. A relação entre o tempo de execução com o uso da varredura seqüencial e o com uso do índice agrupado é 0,85, 0,65, 0,56, 0,68 vezes, respectivamente. No entanto, estes casos não são importantes em nosso trabalho porque com apenas 1 nó ainda não estamos paralelizando as consultas. Para as configurações a partir de 2 nós, o processamento de todas as subconsultas são mais lentas se executadas com a varredura seqüencial habilitada ou, então, são processadas no mesmo tempo. As subconsultas de Q4 e Q21 seriam processadas com eficiência pelo Apuama mesmo que a varredura seqüencial do PostgreSQL não fosse desabilitada. A razão entre o tempo de execução das suas subconsultas é aproximadamente 1 para todas as configurações. Com 2, 4 ou 8 nós, o otimizador opta por um plano de execução que inclui varredura seqüencial da tabela de fato para as subconsultas SVP de quase todas as consultas. Com 32 nós, a subconsulta de Q14 chega quase a ser 97 vezes pior se executada com varredura seqüencial habilitada e a subconsulta de Q3, 70 vezes pior. Concluímos, portanto, que desabilitar a varredura seqüencial para as subconsultas é necessário e obrigatório para obter-se bom desempenho no emprego da técnica SVP no PostgreSQL.

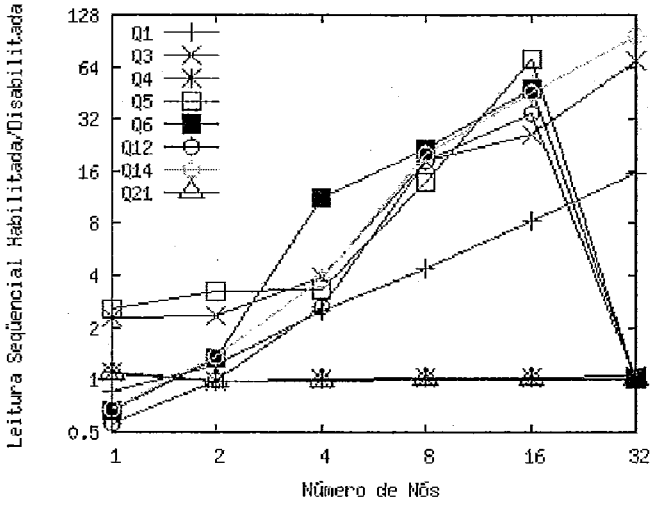

Figura 15 – Razão entre o tempo de execução da subconsulta com o varredura seqüencial habilitada e desabilitada

V.5 Experimentos de Lotes Consultas Concorrentes Apenas de Leitura

No próximo experimento, lotes de consulta apenas de leitura são executados em paralelo contra o DBC. Todos os lotes são compostos pelas mesmas 8 consultas, no entanto ordenadas de maneira diferente de acordo com a especificação TPC-H. Os lotes especificados pelo TPC-H tiveram que ser adaptados porque este inclui todas as 22 consultas. No Apêndice III, são listados os lotes com apenas as consultas OLAP usadas neste trabalho.

As consultas de cada lote são submetidas seqüencialmente, ou seja, a consulta Q_i é enviada apenas depois da consulta Q_{i-1} terminar. Isto é como o TPC-H simula o usuário tomador de decisão formulando novas consultas baseado em resultados de consultas anteriores. As consultas de diferentes lotes são submetidas em paralelo.

Como o experimento com a execução de consultas OLAP isoladas mostrou aceleração superlinear para a maioria das configurações, no experimento com execução concorrente de consultas também a espectativa de bom desempenho. O bom resultado depende da capacidade do SGBD de processar concorrentemente um conjunto de n subconsultas SVP concorrentemente em igual ou menor tempo de execução do que executá-las de forma serial.

A Figura 16 mostra a taxa de vazão em consultas por minutos obtidas durante a execução de 3 lotes concorrentes de consulta em diferentes configurações do agrupamento. Também mostra uma curva de aceleração de vazão que seria atingida se um ganho exatamente linear fosse obtido. O número de lotes está de acordo com a especificação TPC-H, que recomenda este nível de concorrência para base de dados com fator de escala 5. Para todas as configurações, a vazão ascende superlinearmente. Com dois nós, a vazão é bem próxima do linear. Com quatro nós, a vazão é quase duas vezes maior do que se um ganho linear fosse obtido. De 8 a 32 nós, a razão da vazão entre o que é esperado e o obtido pelo Apuama é constante em torno de 6 vezes, portanto mostrando excelente desempenho.

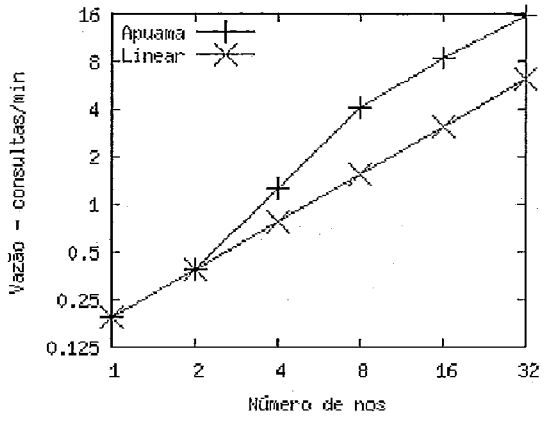

Figura 16 – Experimento de aceleração com 3 lotes de consulta apenas de leitura

A Figura 17 mostra o tempo de execução em experimentos que o Apuama processa lotes de consulta apenas de leitura. Neste experimento, o número de lotes de consultas concorrentes é igual ao número de nós que estão sendo usados. Neste caso, o ideal é que o tempo de execução fosse o mesmo para todas as configurações do agrupamento, como a curva denominada "linear" mostra. Como ocorre no experimento anterior, o desempenho obtido com 2 nós é melhor que o esperado. Com 4 nós, o desempenho é mais que duas vezes melhor do que o esperado. De 8 a 32 nós, o desempenho é sempre em torno de 3 vezes melhor que o esperado, mostrando ótima escalabilidade de vazão. Portanto, o Apuama pode ser usado com sucesso para reduzir o tempo de consultas OLAP e aumentar a vazão do sistema em cenários OLAP típico e também na presença de alta concorrência.

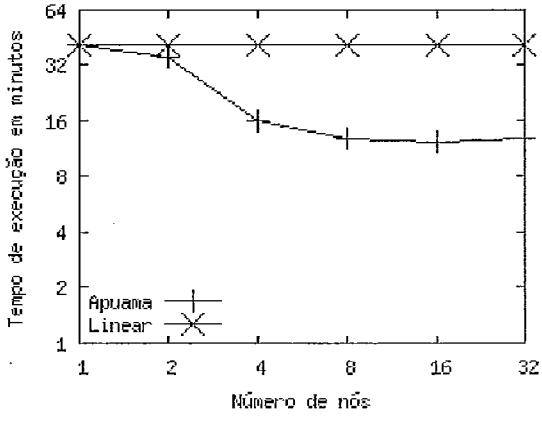

Figura 17 – Experimento de escalabilidade de vazão com *n* lotes de consulta apenas de leitura, em que *n* é igual ao número de nós

V.5.1 Comparação de Desempenho do Paralelismo Intra-consulta com o Inter-consulta

O paralelismo inter-consulta é usado com sucesso para sistemas OLTP, principalmente pelos que têm uma grande quantidade de clientes simultâneos. Para verificar se esta técnica pode ser melhor do que a SVP em testes com lotes de consultas, nesta seção repetimos os experimentos de vazão de consultas OLAP usando apenas o paralelismo inter-consulta provido pelo C-JDBC. O gráfico da Figura 18 mostra uma curva que equivale à razão entre o tempo de execução com o paralelismo inter-consulta e o tempo e execução com o paralelismo intra-consulta. Os tempos correspondem à submissão ao agrupamento de 3 lotes de consultas variando o número de nós envolvidos. O gráfico indica que com dois nós o paralelismo intra-consulta já é aproximadamente 2 vezes mais rápido que o paralelismo inter-consulta. Com 4 nós a razão passa para 4 vezes. Para um número maior de nós esta diferença tende a crescer porque o desempenho do C-JDBC não pode melhorar. Como existem apenas 3 lotes, o C-JDBC pode, no máximo, envolver três nós no processamento da consulta simultaneamente. Enquanto isso, o paralelismo intra-consulta consegue sempre envolver todos os nós do agrupamento. Para 8 nós, a diferença de ganho de desempenho passa para aproximadamente 12 vezes melhor. Para 16 e 32 nós a diferença continua em um crescimento quase linear: aproximadamente 25 e 46 vezes melhor, respectivamente. Dado isso, concluímos que o paralelismo inter-consulta é bem inferior ao paralelismo intra-consulta para o processamento de consultas OLAP em um ambiente típico de DW.

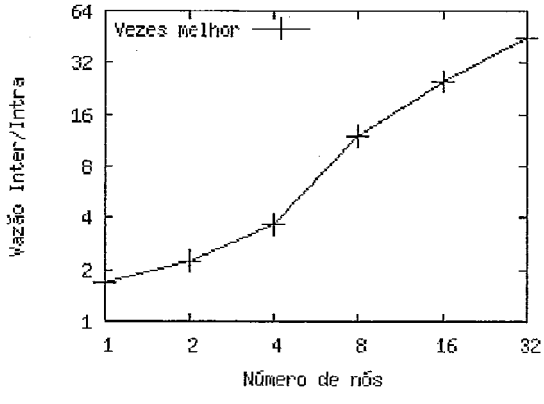

Figura 18 – Comparação entre paralelismo inter-consulta e intra-consulta para aceleração de execução de 3 lotes

No próximo experimento comparativo, a quantidade de lotes é igual à quantidade de nós do agrupamento. Nesta situação, ao contrário da última, o paralelismo inter-consulta ocupa a quantidade máxima dos nós disponíveis. Apesar do paralelismo inter-consulta ser geralmente mais eficiente neste experimento do que em relação ao anterior, a curva da razão entre o tempo das duas técnicas de paralelismo da Figura 19 mostra que o cenário continua favorável ao uso do paralelismo intra-consulta. Com 2 nós, o desempenho entre paralelismo inter-consulta e intra-consulta são próximos. De 4 a 32 nós a razão entre os tempos com o uso de cada uma das técnicas é de aproximadamente 4 vezes. Sendo assim, mesmo nos testes de escalabilidade de vazão o paralelismo intra-consulta mostra-se superior ao paralelismo inter-consulta para processamento de consultas OLAP.

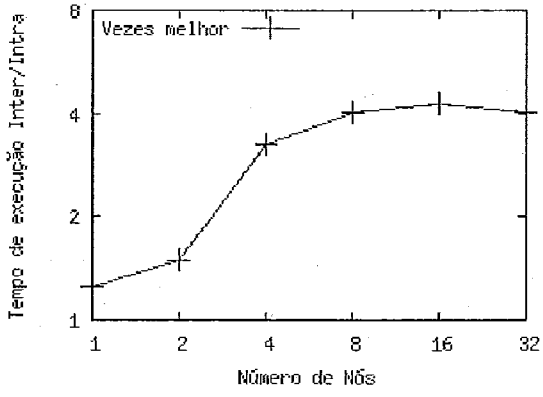

Figura 19 – Comparação entre paralelismo inter-consulta e intra-consulta para escalabilidades com *n* lotes de consulta apenas de leitura, em que *n* é igual ao número de nós

V.6 Experimentos de Lotes de Consulta com Transação Concorrente de Atualizações

Nos experimentos a seguir, combinamos os lotes de consultas apenas de leitura com um lote de atualização. O lote de atualização influi negativamente no desempenho geral do sistema. É um tipo de requisição que não é beneficiado pelo uso do paralelismo, porque a mesma requisição de atualização deve ser repetidamente processada em cada SGBD. Processar *n* atualizações em apenas um SGBD é menos custoso do que processar *n* atualizações em um grupo de SGBDs. Neste experimento

medimos qual é o impacto do protocolo de propagação de resultados no desempenho geral do sistema.

O lote de atualização consiste em 52.500 transações para todas as configurações do agrupamento. Primeiro, um lote de atualizações insere tuplas 30.000 (*SF**4*1500) na tabela *lineitem* e 7.500 tuplas (*SF**1500) na tabela *orders*. Em seguida, as mesmas tuplas que foram inseridos são removidos por 7.500 transações de remoção (*SF**1500) em *lineitem* e por 7.500 transações de remoção (*SF**1500) em *orders*.

A Figura 20 mostra a vazão em consultas por minutos obtidos durante o processamento concorrente de 3 lotes apenas de leitura e um lote de atualização. Novamente é mostrada uma curva correspondente à vazão que seria atingida se o ganho linear fosse obtido. De 2 a 8 nós, o desempenho do Apuama é quase linear. De 8 a 32 nós, o protocolo de consistência faz com que o atraso de propagação de atualizações afete o desempenho geral do Apuama. Não há quase nenhum ganho de desempenho de 16 para 32 nós.

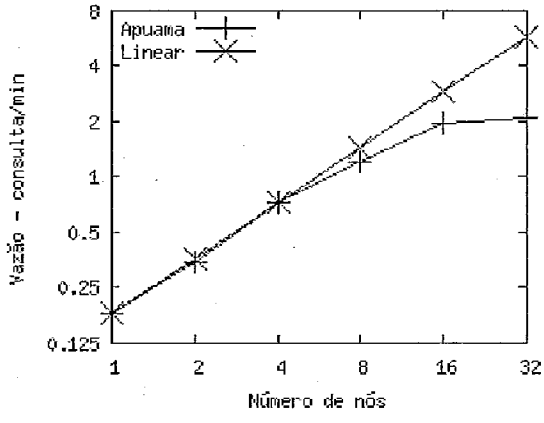

Figura 20 – Experimentos de aceleração de vazão de consultas com 3 lotes concorrentes a 1 lote de atualização

A Figura 21 mostra a escalabilidade do Apuama com um lote de consultas de atualização. Aqui o número de consulta apenas de leitura é igual ao número de nós enquanto temos apenas um lote de atualização. Pode ser notado um ganho de desempenho até 16 nós. No entanto, com 32 nós, o desempenho é quase o mesmo quando com 4 nós. Isto se justifica pelo tempo necessário para sincronizar as réplicas quanto um grande de número de nós é empregado.

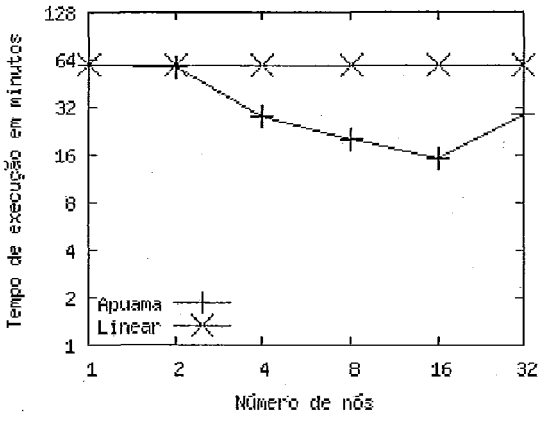

Figura 21 – Experimentos de escalabilidade de consultas com *n* lotes concorrentes a 1 lote de atualização, em que *n* é igual ao número de nós

V.7 Conclusão

Em resumo, esses experimentos mostraram que o Apuama provê excelente desempenho no processamento de carga de trabalho apenas de leitura. Isto é verdade tanto para cenários típicos OLAP quanto para aqueles com um alto grau de concorrência. Com carga de trabalho mista (constituída de consulta apenas de leitura e atualizações), bom desempenho pode ser obtido até com 16 nós. Com 32 nós há degradação de desempenho. O resultado com 32 nós é inferior a algumas configurações do agrupamento de PCs com uma menor quantidade de nós. Políticas de propagação de atualização alternativas devem ser investigadas em desenvolvimento futuro. De forma geral, o sistema de DW não precisa ficar indisponível para o usuário final enquanto a operação de renovação de dados é realizada. Portanto, nos cenários típicos OLAP, quando essas operações ocorrem de tempos em tempos, podemos concluir que o Apuama é uma boa solução para sistemas de suporte à decisão.

VI Conclusão

O Apuama compõe uma solução de DBC de baixo custo capaz de reduzir o tempo de processamento de consultas OLAP e atualizar concorrentemente as bases de dados. O Apuama é de fácil migração porque os SGBDs não precisam ser modificados, não requer hardware específico e as bases de dados não precisam ser fragmentadas. O Apuama foi implementado como uma extensão do C-JDBC, um conhecido DBC em software livre para aplicações OLTP. O Apuama estende o C-JDBC adicionando a capacidade de processamento de consultas OLAP por paralelismo intra-consulta. O código-fonte do C-JDBC não precisou ser modificado para ser estendido. O resultado é um poderoso e único DBC que pode apoiar aplicações OLTP e OLAP. Atualmente, não existe outro DBC que ofereça estas características. Apuama é capaz de processar concorrentemente um grande volume de requisições de leitura e escrita. Implementamos o paralelismo intra-consulta usando SVP, uma técnica de fragmentação virtual que se mostrou eficiente no processamento de consultas OLAP e que pode ser combinada com outras técnicas já conhecidas. O mecanismo implementado não é intrusivo e se baseia exclusivamente em SQL e no *driver* JDBC. Portanto, permite que seja usado em diferentes SGBDs.

Para avaliar nossa solução, implementamos o Apuama sobre um agrupamento de PCs com 32 nós e executamos experimentos com consultas típicas do *benchmark* TPC-H. Quando processamos consultas isoladamente, a aceleração superlinear foi obtida na maioria das situações. Quando processamos lotes paralelos de consulta apenas de leitura, o ganho de desempenho do Apuama é superlinear para todas as configurações do agrupamento, até em cenário com alto nível de concorrência. Com carga de trabalho mista que combina lotes paralelos de consulta apenas de leitura e atualização de uma grande porção de dados, o ganho de desempenho também é muito bom, sendo superlinear em muitas situações. Em testes que incluem atualização de um grande número de nós, o Apuama apresentou deterioração de desempenho por conta do protocolo de consistência de réplicas. Mesmo nesta configuração, a escalabilidade do Apuama é superlinear em situações de alto grau de concorrência. Esse tipo de avaliação não foi encontrado em outros trabalhos da literatura.

Portanto, concluímos que o Apuama é uma solução de alto desempenho adequada para sistemas de suporte à decisão que atende a muitos clientes simultaneamente, mesmo que a base de dados seja atualizada freqüentemente.

VI.1 Contribuições

Esta pesquisa foi iniciada e motivada com base em recente tese de doutorado em que foi proposto o SmaQ (LIMA, 2004). O SmaQ é uma solução eficiente de DBC para processamento de consultas OLAP de leitura e capaz de balancear, com sucesso, dinamicamente, a carga de processamento. Entretanto, o SmaQ não prevê atualizações da base de dados. Em paralelo à elaboração desta dissertação, foi desenvolvido o projeto ParGRES (MATTOSO *et al.*, 2005b), que é uma evolução do SmaQ e prevê requisições de atualização. Contudo, o ParGRES não foi preparado para atender simultaneamente a muitas requisições, como ocorre em aplicações OLTP. Nossa investigação com o Apuama nos levou a níveis de concorrência de processamento de consultas OLAP não atingidos anteriormente, até mesmo com transações simultâneas de atualização. Além desta dissertação, nossa abordagem foi publicada em 2006 no *workshop QLQP* do congresso *EDBT* (MIRANDA *et al.*, 2006). Como nossa motivação é oferecer uma solução de DBC disponível e de baixo custo, o código-fonte do Apuama está publicado livremente em *http://www.cos.ufrj.br/~bmiranda/apuama*, protegido sob a licença de software livre LGPL. Planejamos em breve migrar a hospedagem de nosso código-fonte para um site de projetos de software, como por exemplo, o SourceForge.Net (SOURCEFORGET.NET, 2006). O Apuama, por si só, não é uma solução de DBC, pois é resultado de combinação com C-JDBC. Esta separação permite que as técnicas de paralelismo de consultas OLAP sejam incrementadas independentemente da evolução do C-JDBC. Um sinal que isto é possível, é que o Apuama pode ser alternativamente combinado com o Sequoia (SEQUOIA, 2006), uma nova solução de DBC para aplicações OLTP. O Sequoia faz parte de um novo esforço em software livre patrocinado por uma empresa privada que teve como base o código fonte do C-JDBC.

O Apuama contém uma implementação do algoritmo de paralelismo intra-consulta através da fragmentação SVP que reduz eficientemente o tempo de processamento de consultas. Usando nossa abordagem, a paralelização pode ser empregada em outros SGBDs, não apenas com o PostgreSQL.

Uma outra contribuição desta dissertação é a análise dos trabalhos relacionados. Detalhamos os principais DBCs e SGBDs paralelos existentes e suas características mais importantes. Por meio deste estudo, é possível identificar oportunidades de novos trabalhos e vislumbrar combinação de técnicas oferecidas por diferentes soluções.

VI.2 Trabalhos Futuros

Como desdobramento dos resultados obtidos, futuramente poderiam ser feitos novos experimentos com o objetivo de evoluir as funcionalidades providas pelo Apuama. Os nossos experimentos testaram a capacidade do Apuama de processamento paralelo de consultas OLAP e de atualizações concorrentes usando um *benchmark* para ambiente OLAP. No entanto, apesar de o Apuama acelerar aplicações OLAP e OLTP, os experimentos não levaram em conta a combinação de carga de trabalho de ambos os tipos de aplicação. Sugerimos como trabalho futuro a elaboração de uma avaliação em um cenário de atendimento de aplicações OLAP e OLTP. Entretanto, atualmente não existe a especificação de um *benchmark* para um cenário em que o SGBD atende simultaneamente a requisições típicas de aplicações OLAP e OLTP.

O paralelismo intra-consulta SVP é sensível à distorção da distribuição do valor do atributo de fragmento, assim como ocorre na fragmentação física. Por causa da distorção, poderia ser atribuído mais carga de trabalho para uns SGBDs em detrimento de outros. Entretanto, esta distorção é naturalmente amenizada, porque o atributo de fragmentação geralmente corresponde à chave primária da tupla que, por sua vez, teria uma distribuição de valores satisfatória. Porém, caso a consulta envolva diversas operações a distorção pode provocar uma redução significativa no ganho do desempenho. A fragmentação virtual nos proporciona algumas alternativas de solução, pois o tamanho de cada fragmento pode ser estabelecido dinamicamente. De forma preventiva, o Apuama poderia enviar algumas requisições aos SGBDs com o objetivo de analisar previamente a distribuição de valores da chave primária. Em seguida o Apuama iria redimensionar o tamanho de cada fragmento tentando minimizar a distorção. Numa abordagem corretiva, o Apuama poderia gerar um número maior de fragmentos virtuais do que o número de nós. Cada SGBD iria processar um fragmento virtual por vez e o Apuama atribuiria mais fragmentos virtuais àqueles que terminassem antes de outros o processamento de suas subconsultas SVP.

A replicação total usada pelo Apuama pode ser um limitador de desempenho caso o tamanho da base de dados seja muito grande em relação ao dispositivo de armazenamento. Soluções bem sucedidas como a de FURTADO *et al.* (2005) que utilizam a replicação parcial da base de dados poderiam ser avaliadas junto ao Apuama e indicar melhores abordagens de replicação para o Apuama.

O experimento efetuado com Apuama em ambientes com muitas consultas OLAP, transações de atualizações concorrentes e muitos nós envolvidos indica uma degradação de desempenho. Novos experimentos poderiam ser feitos para identificar o limitador de desempenho no protocolo de propagação e sugerir alternativas. A capacidade do PostgreSQL de processar muitas consultas e atualizações simultaneamente também seria analisada.

Realizar experimentos com outros SGBDs, livres ou comerciais, iria nos oferecer uma visão melhor dos pontos fortes e fracos do uso de cada SGBD com o Apuama e ainda identificar o perfil de desempenho que podemos esperar dos SGBDs em geral.

Referências bibliográficas

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

AUERBACH, K., DORIN, R., WINTER, R., 2005, "2005 TopTen Program", url: http://www.wintercorp.com/vldb/2005_topten_survey/telecon/TopTen%202005_Final.pdf, consultada em 11/09/2005.

BERNSTEIN, P. A., HADZILACOS, V., GOODMAN, N., 1986, "Concurrency Control and Recovery in Database Systems", Versão Eletrônica, Addison-Wesley Longman Publishing Co., Inc.

CECCHET, E., 2004a, "C-JDBC: a Middleware Framework for Database Clustering" *In: Proceedings of IEEE Data Engineering Bulletin* 27(2): pp. 19-26.

CECCHET, E., 2004b, "RAIDb: Redundant Array of Inexpensive Databases", *In: Proceedings of IEEE/ACM International Symposium on Parallel and Distributed Applications (ISPA)*, Hong Kong, China, pp. 115-125.

CECCHET, E., MARGUERITE, J., ZWAENEPOEL, W., 2004, "C-JDBC: Flexible Database Clustering Middleware", *In: Proceedings of USENIX Annual Technical Conference, Freenix Track*, Boston, MA, USA, pp. 9-18.

CHAUDHURI, S., DAYAL, U., 1997, "An Overview of Data Warehousing and OLAP Technology" *In: ACM SIGMOD Record* 26(1): pp. 65-74.

COULON, C., PACITTI, E., VALDURIEZ, P., 2004, "Scaling Up the Preventive Replication of Autonomous Databases in Cluster Systems", *In: Proceedings of 6th International Conference on High Performance Computing for Computational Science (VECPAR)*, Valencia, Spain, pp. 170-183.

CROWL, L. A., 1994, "How to Measure, Present, and Compare Parallel Performance" *In: IEEE Parallel & Distributed Technology: Systems & Technology* 2(1): pp. 9-25.

CRUANES, T., DAGEVILLE, B., GHOSH, B., 2004, "Parallel SQL Execution in Oracle 10g", *In: Proceedings of the ACM SIGMOD International Conference on Management of Data*, Paris, France, pp. 850-854.

DB2, 2005, "DB2 ICE", url: http://ibm.com/software/data/db2/linux/ice, consultada em 11/09/2005.

FURTADO, C., LIMA, A. A. B., PACITTI, E., VALDURIEZ, P., MATTOSO, M., 2005, "Physical and Virtual Partitioning in OLAP Database Clusters", *In: 17th International Symposium on Computer Architecture and High Performance Computing* Rio de Janeiro, Brazil, pp. 143-15.

GANÇARSKI, S., NAACKE, H., PACITTI, E., VALDURIEZ, P., 2002, "Parallel Processing with Autonomous Databases in a Cluster System", *In: Proceedings of International Conference on Cooperative Information Systems (CoopIS)*, Los Angeles, California, pp. 410-428.

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

GRAY, J., HELLAND, P., O'NEIL, P., SHASHA, D., 1996, "The Dangers of Replication and a Solution" *In: ACM SIGMOD Record* 25(2): pp.

HSQL, 2005, "HSQL Database Engine", url: http://hsqldb.org/, consultada em 11/09/2005.

INMON, W. H., 2002, "Building the Data Warehouse", Wiley Computer Publishing, John Wiley & Sons, Inc.

JDBC, 2005, "JDBC", url: java.sun.com/products/jdbc, consultada em 11/09/2005.

KEMME, B., 2000, "Database Replication for Clusters of Workstations", Swiss Federal Institute of Technology Zürich, Zürich, Switzerland, Ph. D. thesis.

LAHIRI, T., SRIHARI, V., CHAN, W., MACNAUGHTON, N., CHANDRASEKARAN, S., 2001, "Cache Fusion: Extending Shared-Disk Clusters with Shared Caches", *In: Proceedings of the 27th International Conference on Very Large Data Bases*, Roma, Italy, pp. 683-686

LIMA, A. A. B., 2004, "Intra-Query Paralelism in Database Clusters", PESC/COPPE, Rio de Janeiro, Brazil, UFRJ, D.Sc.

LIMA, A. A. B., MATTOSO, M., VALDURIEZ, P., 2004a, "Adaptive Virtual Partitioning for OLAP Query Processing in a Database Cluster", *In: Proceedings of 19th Brazilian Symposium on Databases (SBBD)*, Brasilia, Brazil, pp. 92-105.

LIMA, A. A. B., MATTOSO, M., VALDURIEZ, P., 2004b, "OLAP Query Processing in a Database Cluster", *In: Proceedings of the 10th Euro-Par Conference*, Pisa, Italy, pp. 355-362.

MATTOSO, M., ZIMBRÃO, G., LIMA, A. A. B., BAIÃO, F., BRAGANHOLO, V., AVELEDA, A., MIRANDA, B., ALMENTERO, B. K., COSTA, M. N., 2005a, "ParGRES: Middleware para Processamento Paralelo de Consultas OLAP em Clusters de Banco de Dados", *In: Simpósio Brasileiro de Banco de Dados - Sessão de Demos*, Uberlândia, Brasil, Recebeu o prêmio da terceira colocação dentre os softwares apresentados.

MATTOSO, M., ZIMBRÃO, G., LIMA, A. A. B., BAIÃO, F., BRAGANHOLO, V., AVELEDA, A., MIRANDA, B., ALMENTERO, B. K., COSTA, M. N., 2005b, "ParGRES: Uma Camada de Processamento paralelo de Consultas sobre o PostgreSQL", *In*, Porto Alegre, Brasil, pp. 259-264.

MIRANDA, B., LIMA, A. A. B., VALDURIEZ, P., MATTOSO, M., 2006, "Apuama: Combining Intra-query and Inter-query Parallelism in a Database Cluster", *In: EDBT Workshop Proceedings, 11th International Workshop on Foundations of Models and Languages for Data and Objects (FMLDO)*, Munique, Alemanha, pp.

MYSQL, 2005, "MySQL 5.0 Documentation", url: http://mysql.com, consultada em 11/09/2005.

ÖZSU, M. T. and VALDURIEZ, P., 1999, "Principles of Distributed Database Systems", 2 ed., New Jersey, Prentice Hall.

PACITTI, E., ÖZSU, T. and COULON, C., 2003, "Preventive Multi-master Replication in a Cluster of Autonomous Databases", *In: International Conference on*

Parallel and Distributed Computing (Euro-Par), Klagenfurt, Austria, pp. 318-327.

PACITTI, E., VALDURIEZ, P., GAËTAN GAUMER, E., COULON, C., 2005, "RepDB*", url: http://www.sciences.univ-nantes.fr/lina/ATLAS/RepDB, consultada em 12/04/2005.

PARGRES, 2005, "ParGRES Database cluster", url: http://forge.objectweb.org/projects/pargres/, consultada em 05/02/2005.

POSTGRESQL, 2005, "PostgreSQL 8.0.1 Documentation", url: http://postgresql.org, consultada em 11/09/2005.

PROJECT, P., 2005, "Paris Project", url: http://www.irisa.fr/paris/General/cluster.htm, consultada em 11/09/2005.

RÖHM, U., BÖHM, K., SCHEK, H.-J., SCHULDT, H., 2002, "FAS - A Freshness-Sensitive Coordination Middleware for a Cluster of OLAP Components", *In: Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, Hong Kong, China, pp. 754-765.

SCHUMAN, E., 2004, "At Wal-Mart, World's Largest Retail Data Warehouse Gets Even Larger (eWeek article)", url: http://www.eweek.com/article2/0,1895,1675960,00.asp, consultada em 11/09/2005.

SEQUOIA, 2006, "Sequoia Documentation", url: http://sequoia.continuent.org, consultada em 15/02/2006.

SOURCEFORGET.NET, 2006, "SourceForge.Net", url: http://sourceforge.net, consultada em 15/02/2006.

TPC-H, 2005, "TPC-H Benchmark", url: http://tpc.org/tpch, consultada em 11/09/2005.

TPC, 2005, "Transaction Processing Performance Council (TPC)", url: http://tpc.org, consultada em 11/09/2005.

VMSTAT, 2005, "VMSTAT TOOL", url: http://procps.sourceforge.net, consultada em 11/09/2005.

Apêndice I

O TPC-H é um *benchmark* definido pelo TPC (*Transaction Processing Performance Council*) (TPC, 2005). O TPC é uma organização formada pelas principais empresas da indústria de software, principalmente as relacionadas a banco de dados, e que há muito tem seus *benchmarks* aceitos e utilizados pela comunidade científica.

Utilizamos o *benchmark* TPC-H (TPC-H, 2005) que descreve a simulação de uma carga de trabalho de uma aplicação de suporte à decisão. As consultas OLAP geradas a partir desta aplicação têm características *ad-hoc*, ou seja, não são conhecidas previamente. A aplicação permite que o tomador de decisão dinamicamente escolha que tipo de informação deseja visualizar. Por esta razão, nenhuma alteração no esquema físico da base de dados pode ser feita baseada no perfil de consultas que serão efetuadas.

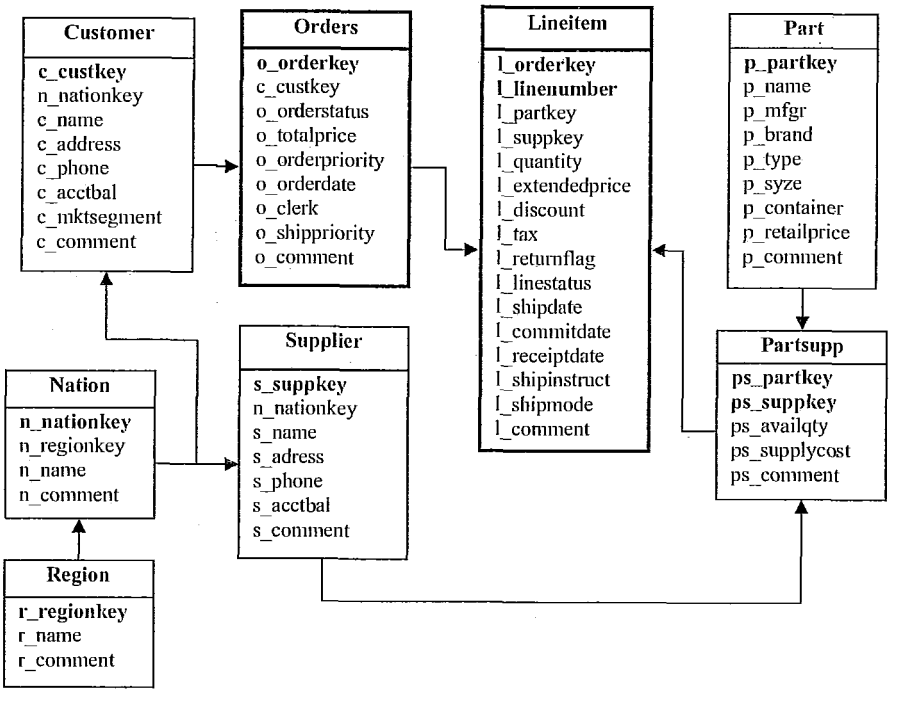

Figura 22 – Esquema da base de dados *benchmark* TPC-H; as bordas em negrito indicam tabelas de fato

Como mostra a Figura 22, o *benchmark* TPC-H define um esquema com 8 tabelas, sendo 2 tabelas de fato (*lineitem e orders*) e 6 tabelas de dimensão (*part, partsupp, supplier, customer, nation e region*). O TPC-H determina as cardinalidades

das tabelas através um de fator de escala (*scale factor – SF*), com exceção das tabelas *nation* e *region*. Cabe ao usuário do *benchmark* escolher fator de escala para seu experimento. As cardinalidades das tabelas são as seguintes: *lineitem* = *SF**6.000.000, *orders* = *SF**1.500.000, *partsupp* = *SF**800.000, *part* = *SF**200.000, *supplier* = *SF**10.000, *customer* = *SF**150.000, *nation* = 25 e *region* = 5.

O TPC-H é composto por 22 consultas OLAP e uma transação concorrente que atualiza aproximadamente 10% da base de dados, inserindo e removendo tuplas das duas tabelas de fato.

Apêndice II

As consultas Q1, Q3, Q4, Q5, Q6, Q12 e Q14 especificadas pelo *benchmark* TPC-H utilizadas em nossos experimentos estão listadas abaixo seguidas de suas respectivas traduções de comando SQL para subconsulta do SVP e agregação de resultados parciais.

Consulta Q1: Original

```
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 s
        um_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 Q1: Subconsulta

```
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,
        sum(l_extendedprice) as avg_price,
        sum(l_discount) as avg_disc,
        count(*) as count_order
from lineitem
where l_shipdate <= date '1998-12-01' - interval '90 day'
```

```
        and l_orderkey >= ? and l_orderkey < ?
        group by l_returnflag,      l_linestatus
```

Consulta Q1: Agregação

```
select l_returnflag, l_linestatus,
        sum(sum_qty) as sum_qty,
        sum(sum_base_price) as sum_base_price,
        sum(sum_disc_price) as sum_disc_price,
        sum(sum_charge) as sum_charge,
        sum(avg_qty)/sum(count_order) as avg_qty,
        sum(avg_price)/sum(count_order) as avg_price,
        sum(avg_disc)/sum(count_order) as avg_disc,
        sum(count_order) as count_order
from %TEMP
group by l_returnflag,      l_linestatus
```

Consulta Q3: Original

```
select
        l_orderkey,
        sum(l_extendedprice * (1 - l_discount)) as revenue,
        o_orderdate,
        o_shippriority
from
        customer,
        orders,
        lineitem
where
        c_mktsegment = 'BUILDING'
        and c_custkey = o_custkey
        and l_orderkey = o_orderkey
        and o_orderdate < date '1995-03-15'
        and l_shipdate > date '1995-03-15'
group by
        l_orderkey,
        o_orderdate,
        o_shippriority
order by
        revenue desc,
        o_orderdate
```

Consulta Q3: Subconsulta

```
select l_orderkey,
        sum(l_extendedprice * (1 - l_discount)) as revenue,
```

```
        o_orderdate,
        o_shippriority
from    customer, orders, lineitem
where c_mktsegment = 'BUILDING'
        and c_custkey = o_custkey
        and l_orderkey = o_orderkey
        and o_orderdate < date '1995-03-15'
        and l_shipdate > date '1995-03-15'
        and o_orderkey >= ? and o_orderkey < ?
        and l_orderkey >= ? and l_orderkey < ?
group by l_orderkey, o_orderdate, o_shippriority
order by
        revenue desc,
        o_orderdate
```

Consulta Q3: Agregação

```
select l_orderkey,
        sum(revenue) as revenue,
        o_orderdate,
        o_shippriority
from %TEMP
group by l_orderkey, o_orderdate, o_shippriority
order by
        revenue desc,
        o_orderdate
```

Consulta Q4: Original

```
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 Q4: Subconsulta

```
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 >= ? and o_orderkey < ?
and exists ( select *
         from lineitem
         where l_orderkey = o_orderkey
         and l_commitdate < l_receiptdate
         and l_orderkey >= ? and l_orderkey < ? )
group by o_orderpriority
```

Consulta Q4: Agregação

```
select o_orderpriority, sum(order_count) as order_count
from %TEMP
group by o_orderpriority
```

Consulta Q5: Original

```
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
        revenue desc
```

Consulta Q5: Subconsulta

```
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 >= ? and o_orderkey < ?
        and l_orderkey >= ? and l_orderkey < ?
        group by n_name
```

Consulta Q5: Agregação

```
select n_name, sum(revenue) as revenue
from %TEMP
group by n_name
```

Consulta Q6: Original

```
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 0.06 - 0.01 and 0.06 + 0.01
        and l_quantity < 24
```

Consulta Q6: Subconsulta

```
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 0.06 - 0.01 and 0.06 + 0.01
        and l_quantity < 24
        and l_orderkey >= ? and l_orderkey < ?
```

Consulta Q6: Agregação

```
select sum(revenue) as revenue
from %TEMP
```

Consulta Q12: Original

```
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 Q12: Subconsulta

```
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 >= ? and o_orderkey < ?
    and l_orderkey >= ? and l_orderkey < ?
    group by l_shipmode
```

Consulta Q12: Agregaçãp

```
select l_shipmode,
    sum(high_line_count) as high_line_count,
    sum(low_line_count) as low_line_count
from %TEMP
group by l_shipmode
```

Consulta Q14: Original

```
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 Q14: Subconsulta

```
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 >= ? and l_orderkey < ?
```

Consulta Q14: Agregação

```
select sum(promo_revenue_1)/sum(promo_revenue_2) as promo_revenue
from %TEMP
```

Consulta Q21: Original

```
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
        numwait desc,
        s_name
```

Consulta Q21: Subconsulta

```
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 >= ?
                     and l2.l_orderkey < ? )
        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 >= ? and l3.l_orderkey < ? )
and s_nationkey = n_nationkey
and n_name = 'SAUDI ARABIA'
and o_orderkey >= ? and o_orderkey < ?
and l1.l_orderkey >= ? and l1.l_orderkey < ?
group by s_name
order by
        numwait desc,
        s_name
```

Consulta Q21: Agregação

```
select
        s_name,
        sum(numwait) as numwait
from
        %TEMP
group by
        s_name
order by
        numwait desc,
        s_name
```

Apêndice III

Os lotes de consultas utilizados para experimentos de vazão segundo o *benchmark* TPC-H foram adaptados neste trabalho. Os lotes, aqui, apenas incluem as consultas Q1, Q3, Q4, Q5, Q6, Q12 e Q14. Como empregamos até 32 lotes em nossos experimentos, foram adaptados os 32 primeiros lotes:

Lote 1: Q21, Q3, Q5, Q6, Q12, Q14, Q1, Q4

Lote 2: Q6, Q14, Q5, Q12, Q1, Q4, Q3, Q21

Lote 3: Q5, Q4, Q6, Q1, Q14, Q21, Q12, Q3

Lote 4: Q5, Q21, Q14, Q12, Q6, Q4, Q1, Q3

Lote 5: Q21, Q4, Q6, Q14, Q3, Q1, Q5, Q12

Lote 6: Q3, Q6, Q4, Q12, Q1, Q5, Q14, Q21

Lote 7: Q21, Q4, Q1, Q3, Q5, Q6, Q14, Q12

Lote 8: Q1, Q5, Q12, Q14, Q4, Q3, Q6, Q21

Lote 9: Q3, Q6, Q21, Q4, Q1, Q12, Q14, Q5

Lote 10: Q6, Q12, Q1, Q21, Q14, Q3, Q4, Q5

Lote 11: Q14, Q1, Q4, Q5, Q12, Q3, Q21, Q6

Lote 12: Q1, Q12, Q6, Q4, Q5, Q21, Q3, Q14

Lote 13: Q21, Q3, Q1, Q12, Q6, Q4, Q5, Q14

Lote 14: Q5, Q4, Q1, Q21, Q14, Q3, Q12, Q6

Lote 15: Q14, Q12, Q6, Q21, Q3, Q5, Q1, Q4

Lote 16: Q1, Q3, Q6, Q5, Q14, Q4, Q12, Q21

Lote 17: Q3, Q5, Q21, Q14, Q1, Q4, Q6, Q12

Lote 18: Q14, Q4, Q5, Q21, Q6, Q3, Q1, Q12

Lote 19: Q4, Q12, Q14, Q5, Q21, Q3, Q6, Q1

Lote 20: Q14, Q4, Q1, Q12, Q5, Q3, Q21, Q6

Lote 21: Q14, Q21, Q12, Q4, Q1, Q5, Q6, Q3

Lote 22: Q14, Q21, Q4, Q1, Q12, Q5, Q6, Q3

Lote 23: Q14, Q12, Q21, Q1, Q6, Q5, Q4, Q3

Lote 24: Q3, Q14, Q21, Q6, Q4, Q1, Q5, Q12

Lote 25: Q1, Q14, Q5, Q21, Q3, Q4, Q12, Q6

Lote 26: Q5, Q21, Q14, Q4, Q6, Q3, Q1, Q12

Lote 27: Q14, Q21, Q6, Q4, Q5, Q3, Q1, Q12

Lote 28: Q1, Q12, Q21, Q14, Q3, Q4, Q5, Q6

Lote 29: Q12, Q6, Q1, Q5, Q3, Q21, Q14, Q4

Lote 30: Q5, Q3, Q1, Q14, Q21, Q12, Q6, Q4

Lote 31: Q21, Q3, Q4, Q1, Q5, Q12, Q6, Q14

Lote 32: Q12, Q4, Q3, Q5, Q6, Q1, Q21, Q14