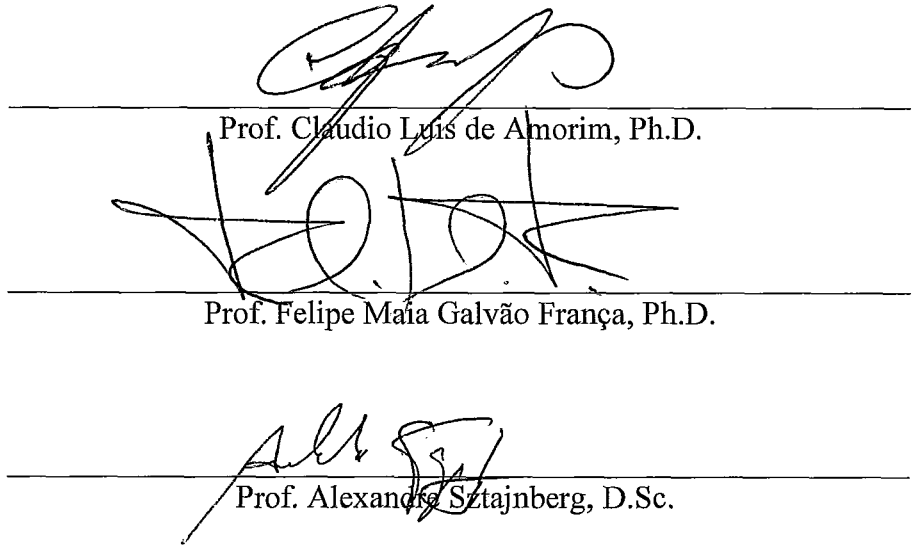
CACHEAMENTO CONSISTENTE, TRANSPARENTE E DISTRIBUÍDO DE BANCO DE DADOS EM SERVIDORES DE COMÉRCIO ELETRÔNICO

Ângelo Nascimento Vimeney

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:



Prof. Claudio Luis de Amorim, Ph.D.

Prof. Felipe Maia Galvão França, Ph.D.

Prof. Alexandre Sztajnberg, D.Sc.

RIO DE JANEIRO, RJ - BRASIL

JUNHO DE 2008

VIMENEY, ÂNGELO NASCIMENTO

Cacheamento Consistente, Transparente e Distribuído de Banco de Dados em Servidores de Comércio Eletrônico [Rio de Janeiro] 2008

XVIII, 145 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2008)

Dissertação – Universidade Federal do Rio de Janeiro, COPPE

1 - Cache para Banco de Dados

2 - Comércio Eletrônico

3 - Java

I. COPPE/UFRJ II. Título (série)

Agradecimentos

Aos professores Claudio Amorim, Felipe França e Alexandre Sztajnberg. Ao Antônio Alexandre. Ao Arthur Granado, Daniel Schmidt, Diego Dutra, João Maurício, Lauro Whately, Leonardo Pinho e aos demais colegas do LCP. Ao Andrey Coppieters, Leandro Marzulo e aos demais colegas da COPPE. Ao Eduardo Cardoso, Roberto Martins e aos demais colegas da easyCAE. À Ester Lima, aos parentes, aos amigos, à Karla Dames e a todos aqueles que de alguma forma ajudaram na realização deste trabalho.

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

CACHEAMENTO CONSISTENTE, TRANSPARENTE E DISTRIBUÍDO DE BANCO DE DADOS EM SERVIDORES DE COMÉRCIO ELETRÔNICO

Ângelo Nascimento Vimeney

Junho/2008

Orientador: Claudio Luis de Amorim

Programa: Engenharia de Sistemas e Computação

A degradação significativa no desempenho dos servidores Web, causada pelo acesso à sistemas de banco de dados, motiva a exploração de técnicas que reduzam o tempo de acesso à banco de dados como uma estratégia promissora na redução da demora experimentada pelo clientes ao navegar em servidores de comércio eletrônico. O sistema dCache é um sistema genérico, transparente e consistente de cache de consultas de banco de dados embutido em um driver JDBC. Neste trabalho, aprimoramos a arquitetura original do driver dCache e a estendemos gerando uma versão distribuída que mantém as caracteríscas desejáveis da arquitetura original e adicionalmente, torna o driver apto a ser empregado em servidores de comércio eletrônico dotados de um cluster de servidores de aplicação. Através da avaliação experimental do driver, mostramos que a sua versão distribuída é capaz de proporcionar ganhos de desempenho em relação a versão original quando o SGBD é um gargalo no servidor de comércio eletrônico. Verificamos ainda, que minimizar o tempo de acesso às caches distribuídas, é uma questão fundamental para que o driver proporcione ganhos de desempenho ainda mais expressivos.

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

TRANSPARENT, CONSISTENT AND DISTRIBUTED DATABASE CACHING FOR E-COMMERCE SERVERS

Ângelo Nascimento Vimeney

June/2008

Advisor: Claudio Luis de Amorim

Department: Computing and Systems Engineering

The significant degradation in performance of Web servers, caused by access to the database systems, motivates the use of techniques that reduce the access time to the database as a promising strategy in reducing the delay experienced by customers while browsing e-commerce sites. The dCache is a generic, transparent and consistent database queries cache system embedded in a JDBC driver. In this work, we improved the original dCache's architecture and extended it generating a distributed version that keeps the original architecture's desirable characteristics and, additionally, makes the driver able to be employed in e-commerce servers equipped with a cluster of application servers. Through the evaluation of the experimental driver, we showed that the distributed version is capable of providing gains in performance over the original version when the DBMS is a bottleneck in the e-commerce server. We noted further that minimize the access time to the caches used by the distributed version, is a key issue for the driver provides more significant speedups.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Listagens

Glossário

AJP13 *Apache JServ Protocol version 1.3.*

ANSI *American National Standards Institute.*

API *application programming interface.*

ARPANet *Advanced Research Project Agency.*

BBS *Bulletin Board System.*

BM *Browsing Mix.*

CSV *Comma Separated Values.*

DDL *Data Definition Language.*

DLM *Distributed Locks Manager.*

DNS *Domain Name System.*

DTD *Document Type Definition.*

HTML *Hyper Text Markup Language.*

HTTP *Hyper Text Transfer Protocol.*

IP *Internet Protocol.*

J2EE *Java 2 Platform Enterprise Edition.*

JDBC *Java Database Connectivity.*

JDK *Java SE Development Kit.*

JIPC *Java Inter-Process Communication.*

JRE *Java Runtime Environment.*

JSE *Java Standard Edition.*

JVM Java Virtual Machine.

LCP Laboratório de Computação Paralela.

LGPL *Library GNU Public License.*

LRU *Least Recently Used.*

OM *Ordering Mix.*

RBE *remote browser emulator.*

SGBD sistema de gerenciamento de banco de dados.

SGBDD sistema de gerenciamento de banco de dados distribuído.

SM *Shopping Mix.*

SMP *Symmetric Multiprocessing.*

SQL *Structured Query Language.*

SSH *Secure Shell.*

SSL *Secure Sockets Layer.*

TCP *Transmission Control Protocol.*

TLS *Transport Layer Security.*

TPC *Transaction Processing Performance Council.*

TPC-W *Transaction Processing Council's Web e-Commerce benchmark.*

UDP *User Datagram Protocol.*

URL *Uniform Resource Locator.*

WIPM *Web Interactions Per Minute.*

WWW *World Wide Web.*

XML *Extensible Markup Language.*

Capítulo 1

Introdução

Desde a sua criação em 1969 e, principalmente, após sua popularização no início dos anos 90, a Internet vem crescendo a cada ano, tanto em número de usuários quanto na quantidade e variedade de serviços disponíveis [1, 2, 3, 4]. Ao longo do seu acelerado crescimento, o perfil dos usuários da Internet passou de militares – usuários da *Advanced Research Project Agency* (ARPANet) – para pesquisadores – com a criação da *World Wide Web* (WWW) – até alcançar o grande público. Já os serviços oferecidos, evoluíram desde a simples transferência de arquivos e troca de mensagens fornecidas por *Bulletin Board Systems* (BBSs), passando por um grande acervo de páginas *Hyper Text Markup Language* (HTML) estáticas interconectadas, até atingirem o grau de sofisticação atual.

Hoje, pode ser encontrada na Internet uma grande quantidade de *sites* que provêem aos seus usuários conteúdo gerado dinamicamente através do emprego de uma diversidade de tecnologias [5] capazes de proporcionar uma experiência de navegação mais interativa, envolvendo desde a simples personalização de páginas *Web* de acordo com a preferência do usuário até o acesso a bases de dados *on-the-fly* (durante a navegação).

Se por um lado a geração dinâmica de conteúdo transforma Web na plataforma alvo de uma série de novas aplicações (por exemplo, as aplicações de comércio eletrônico), por outro, o emprego dessas tecnologias tende a aumentar a latência experimentada pelos usuários ao acessar páginas na Web [6, 7]. Alta latência no acesso a páginas de um *site* de comércio eletrônico implica no insatisfação dos seus clientes. Para uma empresa, a insatisfação de clientes pode, por sua vez, significar perda destes clientes para a concorrência. Uma perspectiva nada desejável quando se estima que reter apenas 5% mais clientes pode aumentar o lucro de um negócio em 100% [8].

O desempenho percebido por clientes navegando na Web vem sendo fortemente

dominado pelo tempo de processamento dos servidores Web, principalmente, quando o cliente acessa um *site* muito popular. Sugere-se que o acesso aos servidores Web representem cerca de 40% do tempo gasto em uma transação Web e que esta percentagem tende a crescer em um futuro próximo [9]. Com a disponibilidade de banda de rede crescendo cerca de duas vezes mais rápido do que a capacidade dos servidores, e com o aumento da complexidade das aplicações Web, é esperado que, cada vez mais, o "gargalo" concentre-se nos servidores [9]. Essa expectativa justifica o estudo e o emprego de técnicas que minimizem o tempo de acesso a estes servidores.

1.1 Servidores de comércio eletrônico

Embora muitos servidores Web [10, 11, 12] sejam capazes de suprir conteúdo dinâmico de forma autônoma, estes servidores são sistemas implementados visando desempenho otimizado no atendimento de pedidos de páginas com conteúdo estático, incluindo imagens e outros objetos armazenados no sistema de arquivos da própria máquina servidora. No caso do servidor Apache, atualmente utilizado por mais de 50% dos domínios da Internet [3], linguagens para geração dinâmica de conteúdo como PHP e Perl são interpretadas por módulos ligados, ou dinamicamente, ou em tempo de compilação, ao código do servidor. O uso de linguagens interpretadas evita que o programador prejudique o funcionamento do servidor, mas o fato do interpretador executar acoplado ao servidor Web impede que o interpretador seja executado em uma máquina separada, alternativa esta que forneceria um grau de isolamento mais apropriado entre servidor Web e interpretador.

No caso da linguagem Java, servidores de aplicação [13, 14, 15, 16, 17] podem prover o conteúdo dinâmico, restringindo a responsabilidade dos servidores Web ao conteúdo estático. Os servidores de aplicação são sistemas implementados visando desempenho otimizado no atendimento de pedidos de páginas geradas dinamicamente, podendo executar em uma máquina dedicada, independente do servidor Web. Esta separação, além de fornecer um isolamento mais apropriado, explora o ponto forte de cada um dos servidores: fornecer páginas estáticas (servidor Web) e fornecer páginas dinâmicas (servidor de aplicação) [6].

Servidores de comércio eletrônico são uma combinação de software e hardware projetada para executar aplicações de comércio eletrônico, as quais fornecem aos seus clientes uma forma de adquirir produtos via Internet através da navegação em uma loja virtual. Um servidor de comércio eletrônico típico é composto por um servidor Web, um servidor de aplicação e um servidor de banco de dados. Em curto prazo, estratégias de

otimização de software [18, 19, 20, 21] e hardware (aumento no número de processadores, memória, etc.) podem diminuir a latência experimentada por clientes no acesso à servidores como este. Entretanto, tais estratégias, além de custosas financeiramente, não são uma solução de longo prazo devido ao crescimento acelerado do número de clientes interessados nestas aplicações [9].

As técnicas de aprimoramentos em software e em hardware são chamadas de *software scale-up* e *hardware scale-up* [9, 22]. Essas duas técnicas visam otimizações aplicadas a uma única máquina servidora, e são chamadas genericamente de *scale-up*. Complementarmente a estas duas técnicas, uma solução mais escalável para o problema da alta latência no acesso aos servidores na Internet são as técnicas de *scale-out*. Ao contrário das técnicas de *scale-up*, as técnicas de *scale-out* procuram diminuir a latência no acesso aos servidores aumentando-se o número de máquinas servidoras empregadas na solução.

Na figura 1.1, observamos que a as técnicas de *scale-out* se dividem em *global scale-out* (quando os servidores estão localizações geográficas distintas) e *local scale-out* (quando os servidores estão em uma única rede local). A técnica *local scale-out* ainda permite duas sub classificações: *cluster-based* e *distributed* (distribuída). Na abordagem distribuída, os endereços IP dos servidores Web são visíveis pela aplicação cliente. A escolha do servidor alvo para tratamento de uma requisição de um cliente é normalmente realizada durante o processo de resolução do nome do *site*, pelo mecanismo de *Domain Name System* (DNS). Na abordagem baseada em clusters, mais moderna, os endereços dos servidores não são visíveis pela aplicação cliente. Apenas um endereço é visível, o qual corresponde a um *switch* Web responsável por escalonar os pedidos dos clientes entre os servidores Web disponíveis.

No decorrer deste trabalho, estaremos interessados na técnica de cluster de servidores de comércio eletrônico. A figura 1.2 mostra a arquitetura de um cluster de servidores de comércio eletrônico típico. Os clientes comunicam-se com o *switch* Web através do protocolo *Hyper Text Transfer Protocol* (HTTP). O *switch* Web pode ser um hardware especializado ou um software rodando em uma máquina dedicada. O *switch* pode analisar o tráfego em diferentes níveis da pilha de protocolo de rede, decidindo qual o servidor Web a ser utilizado para cada requisição, incluindo nesta escolha, algum mecanismo de balanceamento de carga [23].

Os servidores Web recebem as requisições HTTP repassadas pelo *switch* e atendem diretamente os pedidos de páginas estáticas. Os pedidos de páginas dinâmicas são encaminhados ao servidores de aplicação através de algum protocolo conector. Note que,

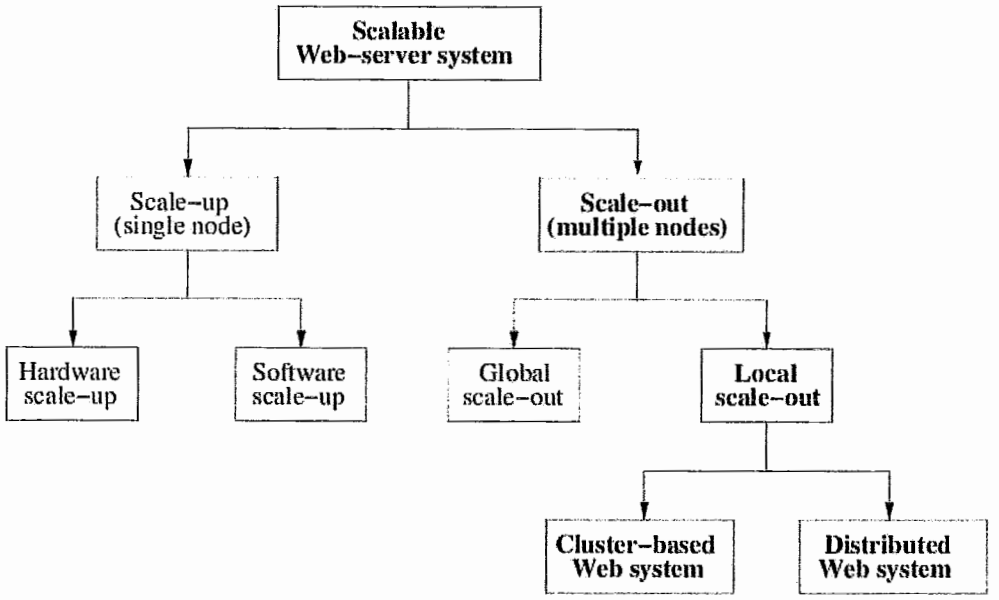


Figura 1.1: Diferentes abordagens para tornar servidores Web escaláveis.

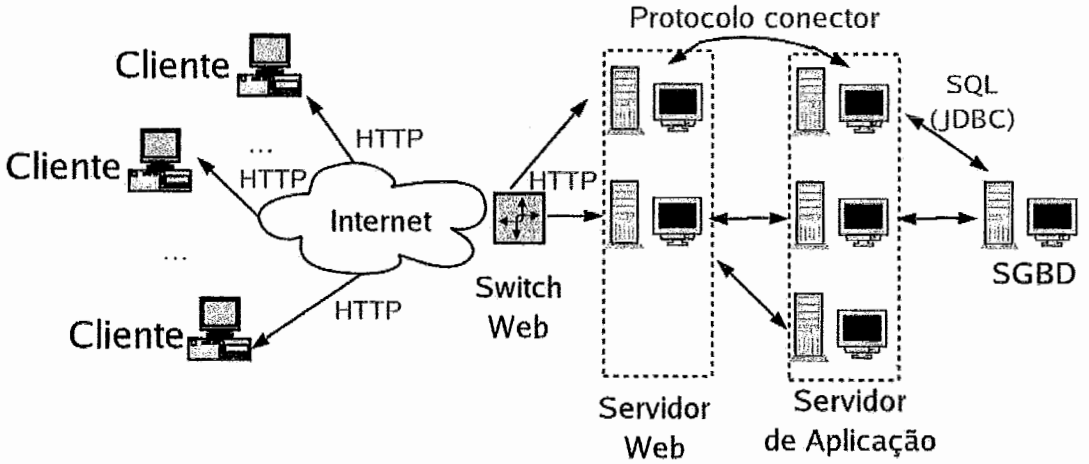


Figura 1.2: Arquitetura de um servidor de comércio eletrônico.

neste ponto, fica clara mais uma vantagem da separação entre servidores Web e servidores de aplicação: um segundo nível de balanceamento de carga pode ser utilizado na escolha de qual servidor de aplicação será utilizado para tratar o pedido de página dinâmica repassado por um servidor Web. Os servidores de aplicação, por sua vez, atendem os pedidos de páginas dinâmicas, eventualmente, acessando o sistema de gerenciamento de banco de dados (SGBD). No caso dos servidores de aplicação Java, a comunicação entre os servidores de aplicação e o SGBD se dá por intermédio da *application programming interface* (API) *Java Database Connectivity* (JDBC) [24, 25].

1.2 Diminuindo o gargalo no acesso ao SGBD

O uso de tecnologias de geração dinâmica de conteúdo pode, por si só, aumentar em até 5 vezes o tempo de acesso a um servidor Web, quando comparado com o tempo de acesso a páginas estáticas de mesmo tamanho. Quando estas tecnologias são utilizadas para realizar acesso a um sistema de banco de dados, o aumento pode chegar até 8 vezes [6, 26]. A degradação significativa no desempenho dos servidores Web, causada pelo acesso a sistemas de banco de dados, motiva a exploração de técnicas que reduzam o tempo de acesso à banco de dados como uma estratégia promissora na redução da demora experimentada pelo clientes ao navegar em servidores de comércio eletrônico.

O uso de sistemas de gerenciamento de bancos de dados distribuídos (SGBDDs) é um exemplo de técnica de redução no tempo de acesso à banco de dados tradicionalmente[1] restrita a grandes organizações, devido ao alto custo de um sistema de banco de dados deste tipo. Uma alternativa (ou um complemento) à utilização de sistemas de bancos de dados distribuídos é a utilização de sistemas de caches para minimizar a quantidade de acessos ao SGBD. A partir do levantamento bibliográfico realizado, percebemos que os sistemas de cache admitem, pelo menos, três classificações: quanto à sua generalidade, quanto à forma de disponibilização e quanto aos objetos armazenados. A classificação quanto à generalidade possui duas categorias:

- *Caches implementadas especificamente para uma aplicação.* As caches desta categoria possuem a vantagem de poderem ser otimizadas para a aplicação para a qual são implementadas. Isso porque o desenvolvedor conhece as peculiaridades da sua aplicação e pode explorar melhor os pontos onde o sistema de cache potencialmente trará os maiores ganhos de desempenho. No entanto, além do custo

[1] Já existem opções gratuitas de SGBDD (pelo menos, o *MySQL Cluster* [27]), porém, essa técnica de redução no tempo de acesso ao banco de dados não foi estudada por este trabalho.

de desenvolvimento de um novo sistema de cache (ou adaptação do sistema já existente) para cada nova aplicação, essa abordagem traz as desvantagens inerentes à qualquer sistema não-reutilizável, como o maior custo de manutenção e maior incidência de erros de programação [28, 29].

- *Caches genéricas.* As caches desta categoria são implementadas para atender a um conjunto de aplicações que obedecem a um, ou alguns, pré-requisitos impostos pelo sistema de cache em questão. As caches desta categoria não podem ser tão otimizadas quanto as da primeira categoria mas, em compensação, possuem as vantagens inerentes a um artefato de software reutilizável, como o menor custo de manutenção e menor incidência de erros de programação. Embora o desenvolvedor não precise escrever o sistema de cache inteiro para a sua aplicação, para utilizar caches desta categoria pode ainda ser necessário algum esforço de programação para adaptar a aplicação às exigências impostas pelo sistema de cache escolhido.

As caches podem ser separadas em três categorias quanto à forma de disponibilização para a aplicação:

- *Cache implementada no código-fonte da aplicação.* Embora não seja obrigatório, caches implementadas especificamente para uma aplicação são fortes candidatas a entrarem nesta categoria pois, uma vez que não podem ser reutilizadas por outras aplicações, o fato de fazerem parte do código-fonte da aplicação à qual se destinam não limita a sua aplicabilidade e ainda pode otimizar o seu desempenho.
- *Cache implementada em uma biblioteca ou framework.* As caches desta categoria obrigam o programador a adaptar a aplicação à API disponibilizada pela biblioteca ou aos padrões impostos pelo *framework*. Essas adaptações, além de poluírem o código-fonte com uma lógica que não é inerente ao problema alvo da aplicação, precisam ser realizadas toda vez que deseja-se migrar de uma determinada sistema de cache para outro [30][2].
- *Cache implementada em driver ou em algum middleware entre a aplicação e o SGBD.* Ao contrário das caches implementadas no código-fonte da aplicação e

[2]A referência [30] (*Hibernate*) trata, na verdade, de um *framework* objeto-relacional, não se propondo em princípio, a ser um sistema de cache para banco de dados. Entretanto, além dele possuir o conceito de seção (que funciona como uma espécie de cache) ele pode ser configurado para utilizar o que ele chama de *second level cache*. Utilizando este recurso, vários sistemas de cache [31, 32, 33, 34, 35] (de terceiros e, inclusive, não necessariamente escritos para atuarem especificamente como caches de banco de dados) podem ser conectados para fornecer mais um nível de cache para o *framework*.

das implementadas em biblioteca ou *framework*, as caches desta categoria [36, 37][38][3] possuem a vantagem de poderem ser transparentes para as aplicações[4]. Além disso, por se localizarem em uma camada intermediária entre as aplicações e o SGBD, estas caches possuem também a vantagem de poderem prover consistência entre diversas aplicações que estejam modificando um mesmo banco de dados. Em sistemas de cache implementados no nível da aplicação, esta propriedade não pode ser garantida pois o sistema de cache não tem controle sobre as modificações realizadas por outras aplicações.

Os sistemas de cache de banco de dados podem ser classificados quanto ao tipo de informação que é armazenada em cache em duas categorias:

- *Caches de tabelas.* Os sistemas de cache desta categoria procuram evitar o acesso da aplicação ao SGBD espelhando as tabelas do banco de dados na memória primária. Assim, as consultas que antes obrigariam o SGBD a recuperar dados do disco, podem ser realizadas a partir da leitura da réplica das informações do disco armazenadas na memória principal.
- *Caches de consultas.* Ao contrário das caches de tabelas, as caches de consultas não são focadas nas tabelas do banco de dados, mas sim, nas consultas realizadas pela aplicação. Sistemas de cache desta categoria procuram evitar o acesso ao SGBD reaproveitando as relações obtidas a partir da execução de consultas anteriormente realizadas pela aplicação.

1.3 O driver dCache

Em [40], é proposto, implementado e avaliado o sistema *dCache*. O sistema dCache é um sistema genérico de cache de consultas de banco de dados embutido em um driver JDBC. Por se tratar de um driver JDBC, o dCache é apropriado para aplicações escritas em Java. O driver dCache intercepta as invocações realizadas pela aplicação à API JDBC tentando atender as requisições da aplicação com os dados armazenados em cache, evitando assim o acesso ao SGBD. Caso o dCache não seja capaz de atender autonomamente a solicitação

[3]Nesta referência é estudado um mecanismo genérico de cache de blocos de disco que, apesar de não se tratar propriamente de um sistemas de cache de banco de dados, pode beneficiar diversos componentes de servidor Web, inclusive, minimizando o gargalo no acesso a bancos de dados.

[4]Fora a transparência para as aplicações, os sistemas de cache ainda poderiam ser classificados em transparentes ou não para o banco de dados. Em [39] é apresentado um sistema de cache implementado em *middleware* que apesar de ser transparente para a aplicação, requer a criação de uma nova coluna em todas as tabelas do banco de dados as quais se pretende armazenar em cache.

da aplicação, é então realizado o acesso ao SGBD. A API JDBC é uma API padrão para acesso de aplicações Java à fontes de dados. Como o driver dCache implementa esta API, aplicações Java que usam JDBC não precisam ser alteradas para utilizarem o driver dCache, que pode então ser considerado transparente para estas aplicações. O driver dCache também é transparente para o banco de dados, ou seja, não exige qualquer modificação nos bancos de dados que são através dele acessados.

O dCache utiliza uma cache de comandos *Structured Query Language* (SQL) para armazenar comandos previamente submetidos ao driver, uma cache de relações para permitir o reaproveitamento de relações geradas pela execução de comandos SQL previamente executados e uma cache de tuplas para permitir o reaproveitamento de tuplas comuns entre relações. O driver dCache se preocupa tanto com comandos somente leitura (consultas SQL) quanto com comandos de escrita, atualizando e invalidando a cache quando necessário. Graças aos mecanismos de atualizações e invalidações, o driver dCache pode ser definido como um sistema de cache *consistente* para banco de dados.

Ainda em [40], o driver dCache é avaliado experimentalmente através do benchmark *Transaction Processing Council's Web e-Commerce benchmark* (TPC-W) [41], que modela uma livraria virtual. Os resultados obtidos demonstraram que o dCache é capaz de proporcionar ganhos de desempenho em servidores de comércio eletrônico, principalmente quando a carga de trabalho submetida ao benchmark é formada em maioria por operações do tipo somente leitura e, sobretudo, quando o servidor de banco de dados é sobrecarregado devido a consultas complexas e/ou envolvendo muitos registros. Neste cenário, o driver dCache chega a proporcionar ganhos de desempenho de quase 130% (medidos em interações Web por minuto)[5].

Em decorrência dos resultados obtidos com a avaliação do dCache, pelo menos três trabalhos futuros são propostos em [40]:

1. *Exploração de outros mecanismos de invalidação.* Em [40] é verificado que mesmo uma taxa pequena de operações de escrita pode causar um grande número de faltas (*misses*) no acesso à cache em função do modelo de invalidações adotado, o qual elimina de uma vez todas as entradas da cache relacionadas com determinada consulta invalidada. É sugerido, então, um modelo de invalidações com mais granulosidade, que elimine da cache apenas as tuplas que de fato não podem ser reaproveitadas.
2. *Exploração de outras engines de armazenamento.* Os experimentos realizados

[5]O benchmark TPC-W e as métricas a ele relacionadas são explicadas em maiores detalhes na seção 3.1.

em [40] utilizaram o SGBD *MySQL* [42] versão 4.1.4, com o motor de armazenamento (*engine*) MyISAM [43]. Esta *engine* é considerada rápida [40], mas não dá suporte ao isolamento de transações (em [40] o isolamento transacional é realizado apenas por *locks* na aplicação e pelo mecanismo de suspensão de uso da cache[6]). Por isso, é sugerido o estudo do impacto do isolamento transacional centrado no SGBD.

3. *Exploração do paralelismo entre servidores de aplicação.* A arquitetura do driver dCache proposta e implementada em [40] suportava a utilização do driver dCache em servidores de comércio eletrônico dotadas de apenas um servidor de aplicação. A constatação de que o desempenho do driver dCache é melhor quando a taxa de ocupação do servidor de banco de dados é alta leva [40] a sugerir como direção futura de investigação, a inclusão no driver de um mecanismo de propagação das atualizações e invalidações para outras instâncias do driver em servidores de aplicação paralelos.

1.4 Objetivo e organização

O presente trabalho dá continuidade ao estudo do sistema dCache, tentando responder às questões deixadas em aberto por [40]. O trabalho pode ser divido em duas fases. Na primeira fase, iremos propor e avaliar experimentalmente modificações na arquitetura e implementação apresentada em [40], mantendo entretanto, a natureza centralizada do sistema. A primeira fase será responsável por agregar aprimoramentos na arquitetura e implementação original do driver, com o objetivo de melhorar o desempenho do dCache quando ele é aplicado em servidores de comércio eletrônico dotados de apenas um servidor de aplicação.

Chamaremos a arquitetura resultante das modificações realizadas na primeira fase de *nova arquitetura centralizada*, ou por simplificação, *arquitetura centralizada*. Dentre as modificações agregadas à nova arquitetura centralizada, está a utilização um novo mecanismo de invalidações, conforme sugerido por [40]. O termo "centralizada" refere-se ao fato de que esta versão da arquitetura pode ser aplicada em servidores de comércio eletrônico dotados de apenas um servidor de aplicação.

Na segunda fase, criaremos uma versão distribuída do driver dCache, inspirada na sugestão de exploração do paralelismo entre servidores de aplicação feita por [40].

[6] O mecanismo de suspensão de uso da cache será explicado em 2.1.5.

Para tanto, iremos propor uma extensão da arquitetura centralizada, implementar a arquitetura proposta e avaliá-la experimentalmente. A versão distribuída permitirá que o sistema dCache possa ser empregado em clusters de comércio eletrônico dotados de diversos servidores de aplicação mantendo estes servidores de aplicação consistentes. Adicionalmente, esperamos que a colaboração entre as diversas instâncias do dCache resulte em maiores taxas de acertos em cache, revertendo-se em ganhos de desempenho mais acentuados do que os apresentados pela versão original.

Nossas propostas de arquiteturas centralizada e distribuída precisarão preservar as características desejáveis já presentes no driver apresentado por [40]. Ou seja, pretende-se que a novas versões continuem possuindo caches consistentes, genéricas o suficiente para serem utilizadas por qualquer aplicação escrita em Java, transparentes para as aplicações e para o banco de dados. A avaliação das novas arquiteturas propostas será realizada utilizando-se uma *engine* de armazenamento com suporte a transações, conforme também proposto por [40].

As arquiteturas centralizada e distribuída serão apresentadas no capítulo 2. Durante a apresentação da arquitetura centralizada, vamos apontar as diferenças com relação à versão original (proposta por [40]) e ressaltar os pontos que precisarão sofrer modificações para dar suporte ao comportamento distribuído. A apresentação da arquitetura distribuída consistirá na descrição das modificações que foram realizadas nos pontos destacados. No mesmo capítulo, apresentaremos também aspectos interessantes das implementações das duas novas arquiteturas, focando nas diferenças entre elas.

No capítulo 3, descreveremos o ambiente experimental utilizado para avaliar as duas arquiteturas, apresentaremos o benchmark utilizado para a avaliação e analisaremos os resultados gerados pelos experimentos. Veremos que as modificações implementadas na arquitetura centralizada do driver proporcionam ganhos de desempenho com relação a arquitetura original proposta em [40]. Veremos que a arquitetura distribuída pode proporcionar tanto ganhos de desempenho quanto perda de desempenho, em função da carga de trabalho submetida ao sistema em teste. Verificaremos que as caches distribuídas, utilizadas na arquitetura distribuída do driver, são um grande gargalo nesta arquitetura.

No capítulo 4, serão propostos trabalhos futuros com o objetivo de aprimorar o funcionamento das caches distribuídas, melhorando então, o desempenho da arquitetura distribuída do driver. No mesmo capítulo, também serão propostos outros trabalhos com o objetivo de conhecer melhor outras possíveis limitações do driver e melhorar o seu desempenho, independentemente dos aprimoramentos nas caches distribuídas.

Nossas conclusões serão apresentadas no capítulo 5. No apêndice B, consta um

manual de utilização do driver dCache. Detalhes sobre a instrumentação realizada no sistema para obtenção dos dados analisados no capítulo de avaliação experimental são apresentados no apêndice A. O apêndice C traz um resumo sobre o pacote JGroups, utilizado na implementação da arquitetura distribuída do driver. O apêndice D traz um resumo sobre o pacote JNetwork, desenvolvido neste trabalho e utilizado também na implementação da arquitetura distribuída. Finalmente, no apêndice E, é apresentado um resumo sobre o monitor de desempenho Sysusager, também desenvolvido neste trabalho e utilizado na avaliação experimental do driver.

Capítulo 2

Arquitetura e implementação do driver

Apresentaremos neste capítulo as arquiteturas centralizada e distribuída do driver dCache. Começaremos descrevendo a arquitetura centralizada e destacaremos as diferenças entre a versão proposta por este trabalho e a versão estudada em [40]. Em seguida, descreveremos a arquitetura distribuída, proposta a partir da extensão da arquitetura centralizada previamente definida. Daremos destaque às modificações necessárias para transformar a arquitetura centralizada em distribuída e apresentaremos também questões relativas à implementação das duas arquiteturas.

2.1 Arquitetura centralizada

A arquitetura centralizada do driver dCache foi realizada visando a obtenção de ganhos de desempenho em aplicações de comércio eletrônico para a Web, sem a necessidade de alteração dessas aplicações ou do banco de dados. Esta arquitetura assumiu as seguintes premissas:

1. Existe um padrão de acesso que envolve uma taxa maior de dados recebidos por parte dos usuários do que dados enviados, como ocorre normalmente em sistemas para a Web. Isso se reflete em uma quantidade maior de leituras do que de escritas no banco de dados.
2. Aplicações de comércio eletrônico utilizam a pesquisa de itens em catálogos. Alterações de informações de catálogo são feitas com freqüência bastante menor do que a freqüência dos acessos feitos a essas informações pelos usuários e são realizadas pelos administradores do sistema utilizando uma interação via Web - tipicamente uma interação para cada item alterado.

3. Os usuários dessas aplicações fazem buscas por palavras-chave para encontrar itens que lhes interessam. Essas buscas seguem uma distribuição estatística típica daquelas que regem o comportamento humano em grandes populações, como é o caso da distribuição de Zipf e demais distribuições ditas de cauda pesada, como a distribuição de Pareto. Com isso, mesmo uma cache pequena pode melhorar significativamente o desempenho dessas aplicações [7].
4. A maior parte das escritas ao banco de dados devida às interações dos usuários diz respeito a dados que têm pouco impacto global no sistema, e mais impacto nas leituras no banco de dados de interesse do próprio usuário.

Para auxiliar na descrição da arquitetura do driver dCache, utilizaremos exemplos de comandos SQL no decorrer deste capítulo. Os resultados obtidos pela execução destes comandos SQL pressupõem a existência de uma base de dados composta por apenas uma tabela `PESSOA`, cujos dados são apresentados na figura 2.1. A tabela pessoa contém quatro colunas: a primeira é a sua chave primária, a segunda contém o nome de pessoas, a terceira a idade dessas pessoas e a quarta, o identificador de um suposto chefe de cada uma das pessoas. Os valores armazenados na quarta coluna são chaves estrangeiras (*foreign keys*) que apontam para a própria tabela `PESSOA`.

PESSOA			
ID	NOME	IDADE	CHEFE_ID
01	João	20	05
02	Maria	25	05
03	Pedro	30	06
04	José	35	06
05	Fátima	40	NULL
06	Paulo	45	NULL

Figura 2.1: Dados na tabela fictícia `PESSOA`

O driver dCache é constituído por três componentes: um *catálogo de metadados de bancos de dados*, um *parser de comandos SQL* e um *núcleo de processamento de comandos SQL* (ou simplesmente, *núcleo de processamento*) (figura 2.2). O catálogo de metadados contém representações de todas as estruturas de banco de dados (*schemas*) que estão sendo acessadas através do driver. Toda vez que é aberta, através do driver dCache, uma conexão para um banco de dados que até então não havia sido acessado, é realizada a leitura da estrutura deste novo banco, que então é armazenada no catálogo de metadados. Veremos adiante que o catálogo provê ao parser de comandos SQL e ao

núcleo de processamento informações úteis para que estes desempenhem suas atividades.

As aplicações que utilizam o driver dCache enviam-no comandos SQL em forma de texto através da API JDBC. O parser é responsável por interpretar os comandos SQL recebidos transformando-os em objetos mais facilmente manipuláveis pelo driver. O núcleo de processamento é o responsável por executar os comandos SQL que foram previamente tratados pelo parser. Quando o núcleo de processamento se vê incapaz de executar um comando SQL com seus próprios recursos, recorre a um *driver JDBC escravo*. O driver JDBC escravo (ou simplesmente, *driver escravo*) é qualquer driver JDBC, eleito pelo usuário da aplicação, capaz de se comunicar com o SGBD em uso pela aplicação. O driver escravo será o único a efetivamente se comunicar com o SGBD, repassando-o comandos SQL que não puderam ser executados de forma autônoma pelo driver dCache.

A figura 2.2 mostra as camadas existentes em um sistema qualquer que utiliza o driver dCache para acessar um SGBD. No primeiro nível, observamos a aplicação cliente do driver dCache. Através da interface JDBC, ela acessa o driver dCache, no nível dois. Também através da interface JDBC, o driver dCache se comunica com o driver escravo, no nível três. No quarto e último nível, reside o SGBD, que se comunica apenas com o driver escravo.

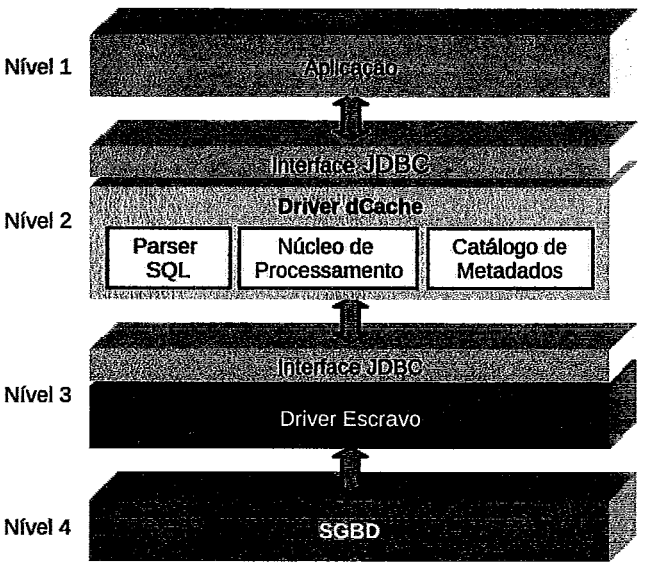


Figura 2.2: Camadas de um sistema usando o driver dCache.

Para minimizar o uso do parser, o núcleo de processamento de comandos SQL do driver dCache utiliza uma *cache de comandos SQL*. A cache de comandos SQL armazena comandos SQL já transformados pelo parser, evitando que estes precisem ser novamente interpretados. Visando minimizar o uso do driver escravo, e assim conseqüen-

temente minimizar o acesso ao SGBD, o núcleo de processamento ainda conta com mais quatro estruturas de dados. São elas: a *cache de relações*, a *cache de tuplas*, a *tabela de modificações locais* e a *tabela de modificações globais*.

2.1.1 As caches de relações e de tuplas

A cache de relações armazena relações provenientes de comandos SQL do tipo SELECT (deste ponto em diante, chamaremos comandos SQL do tipo SELECT de *consultas SQL* ou *consultas*) anteriormente executados pelo núcleo de processamento. Antes de acessar o driver escravo, o núcleo de processamento verifica se já existe uma relação na cache de relações que atenda a consulta SQL que está sendo executada. Em caso positivo, a relação existente será reutilizada. Em caso negativo, o driver escravo será acionado para fornecer uma nova relação, que será utilizada para atender a consulta corrente e também será armazenada na cache para que consultas futuras dela possam se beneficiar.

O driver dCache sempre opera sobre comandos SQL parametrizados[1]. Seja `C1 = "SELECT NOME,IDADE FROM PESSOA WHERE IDADE >= ?"` uma consulta SQL. A consulta `C1`, por exemplo, é um comando SQL parametrizado. A interrogação encontrada no final da consulta sinaliza que naquela posição entrará um argumento que, neste caso, representa a idade mínima de interesse da consulta. Enquanto os parâmetros de uma consulta parametrizada não são fixados, não é possível afirmar qual a relação que será retornada pela consulta. Acessando, com a consulta `C1` e o argumento `35`, nossa base de dados de exemplo, seria retornada a relação `R1`, mostrada na figura 2.3. Já acessando a mesma base de dados, com a mesma consulta `C1`, porém com argumento `25`, seria retornada a relação `R2` (mostrada na figura 2.4), a qual possui duas tuplas a mais que `R1`.

R1	
NOME	IDADE
José	35
Fátima	40
Paulo	45

Figura 2.3: Relação obtida através da consulta `C1` com argumento `35`

[1] A aplicação cliente pode utilizar comandos SQL não parametrizados normalmente (usando o método `java.sql.Connection#createStatement()`, por exemplo), porém, por simplicidade de projeto, internamente esses comandos serão tratados como comandos parametrizados com "zero parâmetros".

Portanto, para acessar a cache de relações, é utilizada como chave, uma consulta parametrizada e os seus argumentos. Com esse par de informações, é possível determinar univocamente qual a relação de interesse. A cache de relações armazena dois tipos de relações: *relações de tuplas* e *relações de ponteiros para tuplas*. Diferentemente das relações de tuplas, as relações de ponteiros, contém, ao invés de tuplas, ponteiros que apontam para tuplas encontradas dentro da cache de tuplas. A razão para se manter tuplas em uma cache separada é minimizar o espaço ocupado por tuplas duplicadas e, principalmente, evitar o acesso ao driver escravo para compor as tuplas de uma relação.

Observando as relações `R1` e `R2`, é fácil verificar a existência de tuplas duplicadas. A consulta `C1` constitui um exemplo de consulta SQL para a qual o núcleo de processamento do driver dCache decidiria armazenar duas relações de ponteiros na cache de relações, e não duas relações de tuplas. A figura 2.5 mostra o estado da cache de relações e da cache de tuplas do driver dCache após a execução da consulta `C1` com argumento `35` e depois com argumento `25`. Podemos observar que na cache de relações existem duas entradas: A primeira representa a relação `R1`, obtida pelo par `(C1,35)`. Por isso contém três linhas, uma para cada tupla de `R1`. A segunda representa `R2`, obtida pelo par `(C1,25)` contendo cinco linhas, uma para cada tupla de `R2`. Na cache tuplas, existem também cinco entradas, representando as três tuplas em comum entre `R1` e `R2` e as duas tuplas restantes da relação `R2`.

Já vimos um exemplo no qual o núcleo de processamento do driver dCache escolheria armazenar relações de ponteiros na cache de relações, mas ainda não explicitamos qual o critério utilizado para a tomada dessa decisão. Consideramos que consultas SQL contendo agregações (funções como SUM, AVERAGE ou COUNT) geram relações com tuplas de baixo potencial de reutilização entre relações distintas. Por esse motivo, não seria produtivo preparar a cache de tuplas para receber este tipo de conteúdo. Sendo assim, relações de tuplas são usadas quando, e somente quando, a consulta SQL contiver

R2	
NOME	IDADE
Maria	25
Pedro	30
José	35
Fátima	40
Paulo	45

Figura 2.4: Relação obtida através da consulta `C1` com argumento `25`

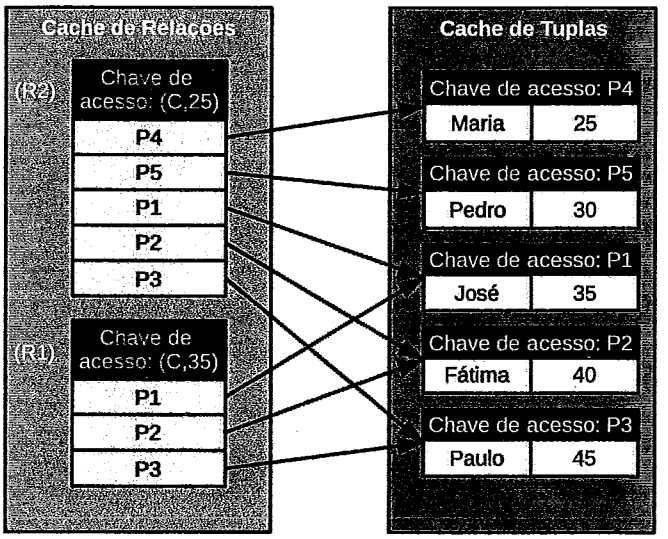


Figura 2.5: Conteúdo das caches de relações e tuplas

agregações. Nos demais casos, relações de ponteiros são preferidas.

Continuando observando a figura 2.5, notamos que a natureza dos ponteiros `P1`, `P2`, `P3`, `P4` e `P5` usados nas relações armazenadas na cache de relações ainda não foi definida. Esses ponteiros são utilizados como chave para acesso a cache de tuplas, logo, para conhecermos a sua natureza, precisamos conhecer qual o formato da chave de acesso à cache de tuplas. Para identificar univocamente uma tupla precisamos duas informações: uma lista de nomes de colunas na forma `NOME_TABELA.NOME_COLUNA` e uma lista de pares `(CHAVE_PRIMÁRIA,VALOR_DA_CHAVE_PRIMÁRIA)`. Para identificar a primeira tupla da relação `R1`, por exemplo, seria utilizada a seguinte lista de colunas e lista de chaves: `((PESSOA.NOME, PESSOA.IDADE), (PESSOA.ID=01))`. Assim, a chave de acesso à cache de tuplas é da forma `(L,I)`, onde `L` é uma lista de nomes de colunas e `I` é uma lista de pares chave primária / valor usados para fixar a linhas destas colunas.

2.1.2 Reescrita de comandos SQL

Dissemos que, para a identificação de tuplas dentro da cache de tuplas, é necessário o uso das chaves primárias das tabelas que contribuem para a composição da tupla. No entanto, nem todas as consultas enviadas para o driver selecionam as chaves primárias das tabelas envolvidas. Sendo assim, o parser do driver dCache precisa reescrever as consultas recebidas adicionando as chaves primárias de todas as tabelas que aparecem na cláusula FROM. O mecanismo de reescrita de consultas é transparente para os usuários do driver, ou seja, as colunas extra são visíveis apenas no escopo do driver. Para viabilizar

a reescrita, o parser precisa ter conhecimento da estrutura do banco de dados que está sendo acessado (precisa saber minimamente quais são as chaves primárias das tabelas). Para isso, é consultado o catálogo de metadados do banco de dados apropriado. Veremos mais adiante que o mecanismo de reescrita de comandos SQL também é útil no acesso às tabelas de modificações locais e globais do driver dCache.

Um caso interessante é o de consultas SQL onde a mesma tabela do banco de dados aparece mais de uma vez com apelidos (*alias*) diferentes. Este recurso é utilizado comumente para realizar junções (*joins*) de tabelas com elas próprias (*self-joins*). Na nossa base de exemplo, poderíamos usar a seguinte consulta `C2` para obter uma listagem contendo o nome de subordinados e seus respectivos chefes:

```
SELECT S.NOME, C.NOME
FROM PESSOA S, PESSOA C
WHERE S.CHEFE_ID = C.ID
```

Nesta consulta, a mesma tabela `PESSOA` aparece sob o apelido `S` (referenciando o subordinado) e sob o apelido `C` (referenciando o chefe). A nova consulta `C2'`, resultante da reescrita de `C2` efetuada pelo parser dCache, seria:

```
SELECT S.NOME, C.NOME, S.ID, C.ID
FROM PESSOA S, PESSOA C
WHERE S.CHEFE_ID = C.ID
```

Repare que o parser não acrescenta apenas uma chave primária da tabela `PESSOA`, e sim, duas, uma para cada "instância" da tabela `PESSOA`. A figura 2.6 mostra a relação `R3`, obtida a partir da execução da consulta `C2'` na base de exemplo. Graças ao acréscimo dessas das duas chaves primárias, o núcleo de processamento de comandos SQL do driver dCache pode, por exemplo, acessar a cache de tuplas para tentar obter a primeira tuplas da relação `R3`, utilizando para isso, a chave `((PESSOA.NOME, PESSOA.NOME), (PESSOA.ID=1, PESSOA.ID=5))`.

2.1.3 Tratando operações de escrita

As caches de relações e tuplas permitem que o núcleo de processamento do driver dCache atenda satisfatoriamente à consultas SQL desde que não sejam executados comandos SQL do tipo INSERT, UPDATE e DELETE (chamaremos esses comandos de *comandos SQL*

R3			
NOME	NOME	ID	ID
João	Fátima	1	5
Maria	Fátima	2	5
Pedro	Paulo	3	6
José	Paulo	4	6

Figura 2.6: Relação obtida através da consulta `C2'`.

de escrita, ou simplesmente, *comandos de escrita*). Porém, os comandos SQL de escrita também precisam ser tratados pelo driver. A execução de comandos de escrita altera o conteúdo das tabelas no banco de dados fazendo com que as caches de relações e de tuplas corram o risco de se tornar inconsistentes caso o driver não lhes dispense um tratamento apropriado. Uma opção de tratamento seria esvaziar as caches de relações e de tuplas toda vez que um comando de escrita fosse executado. No entanto, conforme mostrado por [44], essa opção acarretaria uma baixa taxa de acertos na cache, mesmo considerando-se a premissa de uma pequena quantidade de operações de escrita.

Uma alternativa seria atualizar o conteúdo das caches de relações e de tuplas toda vez que um comando de escrita fosse executado. Essa opção requer do núcleo de processamento do driver dCache a capacidade de determinar, a partir de qualquer comando de escrita, quais as entradas das caches que precisariam ser atualizadas. Consideramos que a complexidade desta análise e os acessos à cache dela decorrentes tornariam o tratamento de comandos de escrita demasiadamente custosos computacionalmente, o que poderia acarretar um baixo desempenho do driver. Optamos por uma abordagem que requer uma análise simplificada de comandos SQL e que mistura a opção de eliminação de entradas da cache com a opção de atualização dessas entradas. A abordagem escolhida conta com dois mecanismos distintos para o tratamento de comandos SQL de escrita: o *mecanismo de atualizações* e o *mecanismo de invalidações*. O mecanismo de atualizações será utilizado para lidar com comandos SQL do tipo UPDATE, enquanto o mecanismo de invalidações será utilizado para tratar tanto comandos SQL do tipo UPDATE quanto os comandos SQL do tipo INSERT e DELETE.

Já descrevemos o papel das caches de comandos SQL, das caches de relações e de tuplas, mas ainda não havíamos descrito o papel da tabela de modificações locais e da tabela de modificações globais. Enquanto as caches de relações e tuplas estão diretamente relacionadas com as consultas SQL, as tabelas de modificações locais e globais estão diretamente relacionadas com os comandos de escrita. Para cada *schema* de banco de

dados, o driver dCache possui apenas uma cache de comandos SQL, apenas uma cache de relações, apenas uma cache de tuplas e apenas uma tabela de modificações globais, independentemente da quantidade de transações em curso. Ou seja, essas estruturas de dados são compartilhadas por todas as transações relativas a um mesmo *schema*.

As tabela de modificações locais, por outro lado, são únicas para cada transação em curso no driver dCache. Em outras palavras, não existe apenas uma tabela de modificações locais, e sim, uma para cada transação em curso. Todas as tabelas, ou colunas de tabelas, que, dentro de uma transação, sofrem modificações por conta da execução de comandos de escrita, são registradas na tabela de modificações locais daquela transação. Quando é solicitado ao driver dCache o *commit* da transação, as modificações são então transferidas para a tabela de modificações globais, tornando-se visíveis para as demais transações. Caso seja solicitado o *rollback* da transação, todas as entradas da tabela de modificações locais são eliminadas. A figura 2.7 mostra a coexistência destas diversas estruturas de dados dentro de uma instância do driver dCache.

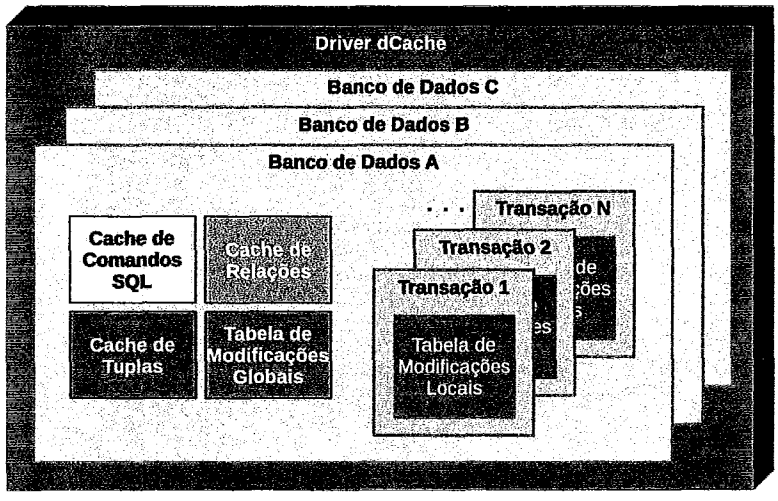


Figura 2.7: Principais estruturas de dados em uma instância dCache

Atualizações

A execução de comandos SQL do tipo UPDATE, adiciona entradas da forma *(TABELA, COLUNA, CHAVE_PRIMÁRIA, VALOR)* na tabela de modificações locais da transação que executa o comando UPDATE. Uma quádrupla desta forma é capaz de identificar uma única célula de uma tabela do banco de dados. A primeira posição da quádrupla indica qual tabela, dentre as possivelmente diversas tabelas do banco de dados que está sendo acessado pela transação em curso, sofreu a modificação. A segunda posição indica

qual a coluna da tabela que foi modificada. A terceira posição, fixa a linha que sofreu a modificação. Por fim, a quarta posição indica o valor mais atualizado para a célula modificada. Na seção 2.1.2 vimos que o conhecimento das chaves primárias de todas as tuplas pertencentes a uma relação é fundamental para o acesso a cache de tuplas. Observando o formato de uma entrada da tabela de modificações locais, verificamos a necessidade de utilização de valores de chave primária de uma tupla, agora para acesso a esta tabela. Vale lembrar que o mecanismo de reescrita de comandos SQL já prove ao núcleo de processamento de comandos SQL os valores destas chaves primárias. Logo, fica viabilizado o acesso à tabela de modificações locais.

A execução de comandos SQL dos tipos INSERT e DELETE, por sua vez, adiciona entradas mais simples na tabela de modificações locais. As entradas adicionadas nestes casos indicam apenas a tabela do banco de dados que sofreu a inserção ou a remoção. Veremos que entradas deste segundo tipo não são usadas pelo mecanismo de atualização, mas pelo mecanismo de suspensão temporária do uso da cache, que será descrito na seção 2.1.5. A tabela de modificações globais admite formatos similares aos usados pela tabela de modificações locais. Quando uma transação realiza um *commit*, as entradas na sua tabela de modificações locais são eliminadas e entradas equivalentes são geradas na tabela de modificações globais.

O mecanismo de atualização consiste na utilização, por parte do núcleo de processamento de comandos SQL do driver dCache, dos registros das modificações realizadas nas células das tabelas do banco de dados, em detrimento de dados previamente armazenados nas caches de relações e de tuplas. Pode-se dizer que, para atender uma consulta SQL, o núcleo de processamento do driver dCache precisa verificar uma hierarquia de até quatro camadas de dados, sendo elas (das mais prioritárias para as menos prioritárias): a tabela de modificações locais da transação em curso, a tabela de modificações globais, as caches de relações e de tuplas e, por fim, o driver escravo. Essa hierarquia é mostrada na figura 2.8.

Através da API JDBC, é sempre solicitado ao driver o valor de uma coluna selecionada pelo comando SELECT por vez[2]. Se um valor atualizado para a coluna solicitada pode ser encontrado na tabela de modificações locais da transação em curso, a busca está finalizada. Caso contrário, o valor para a coluna será buscado nas demais camadas, e assim por diante, até que driver dCache não tenha outra escolha a não ser solicitar o valor para o driver escravo, o que possivelmente[3] resultará em uma consulta ao SGBD.

[2]Nos referimos aos *getters* da classe `java.sql.ResultSet` (por exemplo o método `getString(int)`) que retornam apenas uma célula de um *result set* posicionado em uma tupla por vez.

[3]Utilizamos "possivelmente" pois o driver escravo poderia também possuir algum mecanismo de cache

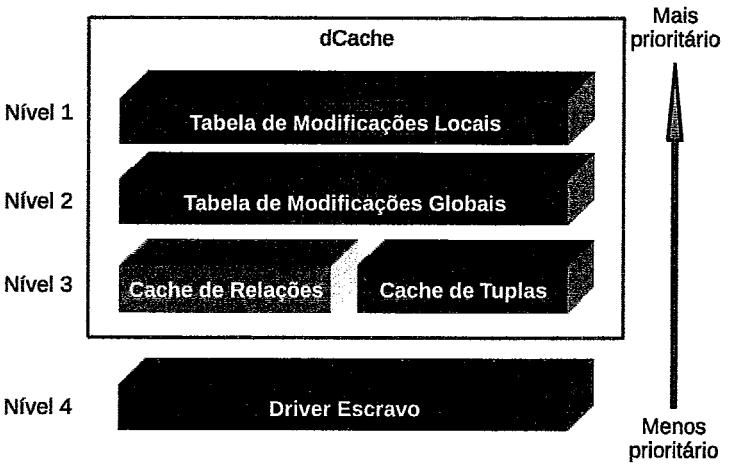


Figura 2.8: Hierarquia de dados do driver dCache

Uma pequena variação à busca hierárquica descrita é o caso dos resultados de funções de agregação, os quais são buscados a partir da terceira camada. Graças ao formato de dados utilizado pelas tabelas de modificações locais e globais, não faria sentido buscar nem na primeira, nem na segunda camada, valores da coluna da relação obtida por uma consulta SQL como "`SELECT SUM(IDADE) FROM PESSOA`", afinal, essa coluna não pertencente a nenhuma tabela do banco de dados sendo acessado.

Invalidações

Para entendermos o mecanismo de invalidações se torna importante a apresentação do conceito de tabelas e colunas críticas para uma consulta SQL. Consideramos que uma tabela é crítica para uma consulta SQL se, e somente se, a execução de comandos SQL do tipo INSERT ou DELETE afetando esta tabela pode alterar a relação resultante da execução da consulta SQL em questão. Na consulta "`SELECT * FROM PESSOA`", por exemplo, a tabela `PESSOA` é uma tabela crítica pois, claramente, dela inserindo-se ou removendo-se registros, a relação resultante da execução da consulta seria alterada. Analogamente, consideramos que uma coluna é crítica para uma consulta SQL se, e somente se, a execução de comandos UPDATE afetando esta coluna pode alterar a relação resultante da execução da consulta SQL. Na consulta "`SELECT * FROM PESSOA WHERE IDADE > 35`", por exemplo, a coluna `IDADE` é uma coluna crítica pois realizando-se um comando SQL do tipo UPDATE que afete a coluna `IDADE`, a relação resultante da

que prevenisse o acesso ao SGBD.

execução da consulta possivelmente seria alterada[4].

Como havíamos dito, os formatos das entradas da tabela de modificações globais são similares aos da tabela de modificações locais. Porém, essas entradas possuem um campo adicional. A tabela de modificações globais, além de armazenar uma referência para a célula (ou tabela) que sofreu a modificação, armazena um rótulo de tempo, marcando o instante de tempo em que a célula (ou tabela) sofreu a modificação. Assim, a execução de um comando SQL do tipo UPDATE, adicionaria uma entrada da forma `(TABELA, COLUNA, CHAVE_PRIMARIA, VALOR, RÓTULO_TEMPO)` na tabela de modificações globais, quando a transação em curso realizasse *commit*. Já a execução de um comando SQL do tipo INSERT ou DELETE adicionaria uma entrada da forma `(TABELA, RÓTULO_TEMPO)` quando a transação em curso realizasse *commit*.

Já descrevemos o funcionamento da cache de relações, mas ainda não havíamos mencionado que as relações armazenadas na cache de relações (sejam elas relações de tuplas ou de ponteiros para tuplas) também possuem rótulos de tempo. Enquanto que os rótulos de tempo na tabela de modificações globais indicam o instante em que foi realizada uma modificação, no caso das relações armazenadas na cache de relações, estes rótulos de tempo indicam o instante em que a relação foi adicionada na cache de relações. Veremos que o mecanismo de invalidação é baseado na comparação entre os rótulos de tempo das relações que estão armazenadas na cache relações com os rótulos de tempo das entradas registradas na tabela de modificações globais.

Toda vez que, em benefício da execução de uma consulta SQL solicitada por uma transação em curso, o núcleo de processamento de comandos SQL do driver dCache obtém uma relação a partir da cache de relações, é realizado um teste de validade. O objetivo do teste de validade é garantir que os valores armazenados em uma relação obtida a partir da cache de relações podem ser utilizados pela transação em curso sem o risco de estarem inconsistentes. O teste de validade consiste na comparação do rótulo de tempo associado a relação obtida a partir da cache de relações com os rótulos encontrados na tabela de modificações globais, para cada uma das tabelas e colunas críticas da consulta SQL em execução. A existência de algum rótulo de tempo na tabela de modificações globais maior do que o rótulo de tempo da relação obtida da cache de relações indica que a relação em cache pode estar inconsistente. Neste caso, a relação é eliminada da

[4]Na implementação do parser de comandos SQL, são consideradas colunas críticas para uma determinada consulta C, todas as colunas que aparecem nas cláusulas WHERE, ORDER BY e GROUP BY da consulta C. São consideradas tabelas críticas, todas as tabelas que aparecem na cláusula FROM da consulta C.

cache de relações e em seu lugar é inserida uma nova relação (chamamos este processo de invalidação da relação).

Vamos mostrar o mecanismo de invalidação em ação usando o código de exemplo 2.1. Na linha 1, estamos assumindo que a variável `con` contém uma referência para uma instância da classe `java.sql.Connection`, a qual está conectada ao nosso banco de dados de exemplo via driver dCache. Ainda na mesma linha, a conexão em questão é utilizada para criar uma consulta parametrizada (um objeto da classe `java.sql.PreparedStatement`) a partir do texto "`SELECT * FROM PESSOA`". A execução desta linha faria com que o parser do driver dCache interpretasse o texto enviado pela aplicação, transformando em uma estrutura compilada mais facilmente manipulável pelo núcleo de processamento de comandos SQL.

```
1  PreparedStatement ps = con.prepareStatement("SELECT * FROM PESSOA");
2  ResultSet rs = ps.executeQuery();
3
4  while(rs.next()){
5     String nome = rs.getString('NOME');
6     int idade = rs.getInt('IDADE');
7     int chefe = rs.getInt('CHEFE_ID');
8     System.out.println(nome+" "+idade+" "+chefe);
9  }
10
11 PreparedStatement update = con.prepareStatement("DELETE FROM PESSOA WHERE
       ID=?");
12 update.setInt(1,1);
13 update.executeUpdate();
14 con.commit();
15
16 ps.executeQuery();
```

Listagem 2.1: Consultando e removendo registros da base de dados

A chamada ao método `executeQuery()` (linha 2) fará com que o driver dCache acesse a cache de relações para tentar reaproveitar uma relação correspondente ao resultado esperado pela consulta que está sendo executada. Vamos supor que a cache de relações não contivesse tal relação. O núcleo de processamento seria obrigado a criar uma nova relação, a qual seria armazenada na cache para reutilização futura. No momento do armazenamento da nova relação na cache de relações, ela ganharia um rótulo de tempo, vamos dizer, `1`. As linhas 4 até 9 são usadas para obter e imprimir o nome, idade e chefe das seis pessoas que foram selecionadas pela consulta executada em nossa base de exemplo.

Na linha 11, é criado um comando do tipo DELETE, parametrizado, executado na linha 13. A execução deste comando adiciona uma entrada na tabela de modificações locais da transação em curso. Na linha 14, a tabela de modificações locais é esvaziada e a tabela de modificações globais, por sua vez, recebe a entrada `(PESSOA, 2)`, indicando que a tabela `PESSOA` sofreu uma modificação no instante de tempo `2`. Quando, na linha 16, a consulta "`SELECT * FROM PESSOA`" é repetida, a cache de relações é novamente consultada. Agora, a relação gerada anteriormente será encontrada na cache e será submetida ao teste de validade.

Como parte do teste de validade, o mecanismo de invalidações verificará que a tabela `PESSOA` é uma tabela crítica para a consulta executada, por isso, seu rótulo de tempo precisa ser comparado com o rótulo de tempo da relação obtida a partir da cache. Como vimos, a relação foi armazenada na cache com rótulo de tempo `1`, enquanto que o rótulo de tempo armazenado na tabela de modificações globais para a tabela `PESSOA` foi `2`. Como `2` é maior do que `1`, e os rótulos de tempo são sempre gerados de forma crescente, o mecanismo de invalidações conclui que a tabela `PESSOA` sofreu uma modificação depois de ter sido armazenada em cache e, por isso, seu conteúdo possivelmente está inconsistente. De fato, a execução do comando DELETE iniciada na linha 11 removeu um registro da tabela `PESSOA`, por isso, seria esperado que a execução da consulta "`SELECT * FROM PESSOA`" retornasse uma tupla a menos do que as seis contidas na relação invalidada. Assim, a relação é descartada e outra é criada (através do acesso ao driver escravo) para atender à consulta SQL em execução.

2.1.4 Leitura de valores sob demanda

Um outro mecanismo utilizado pelo núcleo de processamento de comandos SQL do driver dCache para minimizar o acesso ao driver escravo é a *leitura de valores sob demanda*. A leitura de valores sob demanda é um mecanismo que tentar adiar ao máximo o acesso ao driver escravo através do preenchimento das relações e das tuplas armazenadas nas caches de relações e tuplas somente quando estes valores são de fato requisitados ao núcleo de processamento através da API JDBC.

Voltemos ao código de exemplo 2.1. Havíamos dito que a consulta executada na linha 2 faz com que núcleo de processamento crie uma nova relação e acrescente-a na cache de relações. Esta relação ainda não conteria nenhuma tupla. As tuplas desta relação serão acrescentadas sob demanda. Este comportamento é ditado pelo mecanismo de leitura de valores sob demanda, aplicado às tuplas de uma relação. Chamaremos as relações

as quais ainda não possuem todas as suas tuplas de *relações incompletas*. Chamaremos de *relações completas* aquelas que já possuem todas as suas tuplas.

A chamada ao método `next()` (linha 4) fará com que o mecanismo de leitura de valores sob demanda acrescente uma primeira tupla na relação incompleta. A tupla acrescentada à relação, entretanto, contém apenas as informações indispensáveis para o correto funcionamento do driver dCache: neste caso o valor da chave primária da primeira tupla lida. Os valores das demais colunas selecionadas pela consulta (`NOME`, `IDADE` e `CHEFE_ID`) ainda não foram atribuídos na tupla pelo núcleo de processamento. Os valores das colunas faltantes serão atribuídos somente nas linhas 5, 6 e 7, quando eles são requisitados pela aplicação cliente. A cada invocação ao método `next()` executada pelo laço `while` uma tupla é acrescentada à relação, até que ela ganhe seis tuplas, tornando-se completa.

2.1.5 Suspensão temporária de uso da cache

O *mecanismo de suspensão temporária de uso da cache* será o responsável por evitar a ocorrência de dois problemas que serão explicada a seguir: problema da contaminação das caches e o problema da modificação local ignorada. Assim como o mecanismo de atualização, o mecanismo de suspensão temporária de uso da cache utiliza a tabela de modificações locais das transações em curso. Toda vez que uma transação executa uma consulta na qual, pelo menos uma de suas tabelas e/ou colunas críticas encontra-se na tabela de modificações locais daquela transação, é realizada a suspensão de uso da cache.

A suspensão de uso da cache consiste no impedimento do uso da cache de relações para obtenção de relações que atenderiam a execução de consultas SQL que possuem tabelas e/ou colunas críticas registradas na tabela de modificações locais daquela transação. A verificação da presença de tabelas e/ou colunas críticas na tabela de modificações globais da transação visa limitar as suspensões de uso da cache somente aos casos em que, de fato, os dois problemas mencionados podem se manifestar.

O problema da contaminação das caches

Contando com as caches de relações e de tuplas, e com as tabelas de modificações locais e globais, o driver dCache possui autonomia suficiente para executar muitas consultas SQL sem que o driver escravo sequer tome conhecimento de que estas consultas foram solicitas pela aplicação cliente. Já a execução de comandos SQL de escrita sempre precisa ser realizada com o conhecimento do driver escravo. Ou seja, mesmo registrando

modificações nas tabelas de modificações locais e globais, o driver dCache submete os comandos de escrita à execução pelo driver escravo. Esse procedimento visa manter o driver escravo sempre atualizado, pois ele precisará executar as consultas SQL que o driver dCache não terá condições de realizar de forma autônoma, e queremos que ele gere resultados consistentes.

Assim, quando uma transação realiza modificações no banco de dados (modificações estas que ainda não sofreram *commit*, ou seja, precisam ser visíveis apenas pela transação em curso), duas entidades passam a possuir conhecimento dessas modificações: a tabela de modificações locais desta transação e o driver escravo. Suponhamos então, que uma transação execute uma consulta SQL e que o driver dCache tenha condições de executar de forma autônoma esta consulta, fornecendo para esta transação uma relação incompleta, obtida a partir da cache de relações. Em seguida, a suposta transação realiza modificações nas tabelas e/ou colunas críticas da consulta realizada.

A transação em curso poderá utilizar as tuplas possivelmente já existentes nesta relação incompleta, com a certeza de que, graças ao mecanismo de atualização, os valores modificados serão lidos corretamente da sua tabela de modificações locais. Entretanto, o problema da *contaminação das caches* emerge quando esta transação tenta utilizar tuplas ainda não existentes nesta relação incompleta: O mecanismo de leitura de valores sob demanda iria acessar o driver escravo para obter as tuplas faltantes e, uma vez que o driver escravo já possui conhecimento das modificações efetuadas pela transação em curso, forneceria valores modificados pela transação.

A primeira vista, este comportamento pode parecer coerente, no entanto, precisamos lembrar que a relação que recebe estes valores modificados continuará disponível na cache de relações para reutilização por outras transações, que terão acesso a informações que deveriam ser visíveis apenas pela transação que realizou as modificações. Para que o driver dCache possa dar suporte ao nível 1 do padrão *American National Standards Institute (ANSI)* (*Read Committed*) de isolamento entre transações, é necessário que as modificações efetuadas por uma transação não sejam visíveis pelas demais até que a primeira realize *commit*.

O problema da modificação local ignorada

Na seção 2.1.3, vimos que o mecanismo de invalidações previne que uma determinada transação reutilize relações oriundas da cache de relações quando uma ou mais tabelas e/ou colunas críticas da consulta associada a esta relação sofreram modificações. O me-

canismo de invalidações, entretanto, utiliza-se da tabela de modificações globais para determinar se alguma tabela e/ou coluna crítica sofreu modificações. Ou seja, modificações no escopo de uma única transação (registradas na tabela de modificações locais), não são consideradas pelo mecanismo de invalidações. O problema da *modificação local ignorada* consiste na possibilidade de uma transação utilizar, inadvertidamente, relações oriundas da cache de relações que teriam ficado inconsistentes (no escopo desta transação) graças a modificações que esta transação realizou apenas no seu escopo.

2.1.6 Regiões críticas

Vimos que o núcleo de processamento de comandos SQL do driver dCache conta com cinco estruturas de dados para realizar suas atividades: a cache de comandos SQL, a cache de relações, a cache de tuplas, a tabela de modificações globais e a tabela de modificações locais. Vimos que as quatro primeiras estruturas de dados são compartilhadas por todas as transações em curso no driver, enquanto que a tabela de modificações locais contém informações úteis para uma única transação. Vamos analisar quais os cuidados que precisam ser tomados pela arquitetura do driver para evitar que o acesso concorrente a essas quatro estruturas de dados compartilhadas gere problemas de consistência para o driver e, adicionalmente, também verificaremos qual o impacto das restrições de acesso a essas estruturas no nível de isolamento entre as transações.

No caso da cache de comandos SQL, da cache de relações e da cache de tuplas, a única exigência feita pela arquitetura do driver é que a implementação das mesmas saiba lidar corretamente com leituras e/ou escritas concorrentes. Ou seja, não existirão problemas de concorrência do ponto de vista arquitetural, se não existirem problemas de concorrência no nível de granulosidade correspondente a operações de leitura e escrita de entradas individuais na cache. Já a tabela de modificações globais precisa ser analisada com mais detalhes. A tabela de modificações globais é utilizada pelo núcleo de processamento em três circunstâncias: durante o processo de *commit* de transações, durante os testes de validade e pelo mecanismo de atualização, quando o valor de uma célula de uma tupla é solicitada pela aplicação cliente. O uso concorrente da tabela de modificações globais por estas três seções de código pode transformá-las em regiões críticas. Vamos estudar, começando pelo processo de *commit*, cada uma destas potenciais regiões críticas, e verificar em quais circunstâncias será necessária a imposição de acesso em regime de exclusão mútua.

Descrevemos no decorrer das seções anteriores, de forma independente, algumas

atividades que o núcleo de processamentos de comandos SQL do driver dCache realiza durante o processo de *commit* de uma transação. Na seção 2.1.3, vimos que, durante o processo de *commit*, a tabela de modificações globais do driver dCache é atualizada e que são eliminadas as entradas da tabela de modificações locais da transação em curso. Na seção 2.1.3, vimos que o núcleo de processamento de comandos SQL do driver dCache utiliza um rótulo de tempo para registrar o momento que as tabelas e colunas modificadas pela transação em curso passam pelo processo de *commit*. Na seção 2.1.5 foi dito que, ainda durante o processo de *commit*, o driver escravo é informado sobre as modificações realizadas pela transação em curso. Agora será importante sintetizar as etapas do processo de *commit* de uma transação do driver dCache, o que é feito pela listagem abaixo:

1. Obtenção de um novo rótulo de tempo;
2. Para cada tabela que recebeu uma inserção ou remoção de tupla, atualização do registro associado na tabela de modificações globais utilizando-se o rótulo de tempo obtido;
3. Para cada coluna atualizada, escrita do valor atualizado na tabela de modificações globais, associando-o ao rótulo de tempo obtido.
4. Esvaziamento da tabela de modificações locais da transação em curso;
5. Notificação do driver escravo sobre as modificações realizadas pela transação em curso.

Vamos supor que duas transações em curso no driver dCache alterem as colunas `NOME` e `IDADE` de uma mesma tupla da tabela `PESSOA`. A primeira transação atualiza o nome para `MARCOS` e a idade para `60`. A segunda transação atualiza o nome para `TADEU` e a idade para `70`. Durante o processo de *commit*, ambas transações precisarão atualizar a tabela de modificações globais, porém, cada uma com um valor diferente para as mesmas colunas. Se as duas realizam o *commit* concorrentemente, poderia ocorrer a situação em que a primeira transação escreve o valor `MARCOS` na tabela de modificações globais, em seguida, a segunda transação sobrescreve esta coluna com o valor `TADEU` e escreve na coluna `IDADE` o valor `70`. A primeira transação volta a executar e sobrescreve então o valor `60` na coluna `IDADE`. O valores associados à tupla modificada ao final do *commit* das duas transações concorrentes são `TADEU` e `60`, um par de informações que não corresponde nem ao desejado pela primeira transação, nem ao desejado pela segunda transação. Este exemplo mostra que, para garantir as propriedades de atomicidade e consistência das transações [45, 46, 47] do driver dCache, é necessário que o código de *commit* seja tratado como uma região crítica a ser acessada em exclusão mútua.

Na seção 2.1.3, vimos que o núcleo de processamento de comandos SQL do

driver dCache, antes de reutilizar uma relação oriunda da cache de relações, realiza um teste de validade. Também foi mostrado que este teste de validade é composto pelas seguintes etapas:

1. Obtenção das tabelas e colunas críticas da consulta que dá origem a relação a ser reutilizada;
2. Comparação do rótulo de tempo associado à relação a ser reutilizada com os rótulos de tempo associados às tabelas e colunas críticas da consulta.

Neste ponto, é importante observar que os rótulos de tempo associados às tabelas e colunas críticas da consulta foram gerados exatamente pela etapa 1 do processo de *commit* descrito acima, e gravados na tabela de modificações globais nas etapas 2 e 3, também do processo de *commit*. Imaginemos duas transações executando concorrentemente no driver dCache. A primeira, em processo de *commit*, atualiza dados na tabela PESSOA. A segunda, realiza um teste de validade em que a mesma tabela PESSOA é uma tabela crítica. Digamos que a primeira transação atualize o rótulo de tempo na tabela de modificações globais antes da segunda transação realizar a comparação entre os rótulos de tempo (etapa 2 do teste de validade). Caso o processo de *commit* da primeira transação seja abortado por um motivo qualquer (uma falha no acesso ao driver escravo, por exemplo), a segunda transação poderá invalidar uma relação obtida a partir da cache de relações desnecessariamente. Este comportamento, apesar de aumentar a taxa de faltas no acesso a cache de relações, não gera inconsistências no driver e portanto não torna necessária a imposição de exclusão mútua entre os testes de validade e o processo de *commit*.

Digamos agora, que a segunda transação realize a comparação entre os rótulos de tempo antes da primeira transação atualizar o rótulo de tempo na tabela de modificações globais. A segunda transação considerará que a relação obtida a partir da cache de relações é valida, embora a primeira transação esteja prestes a realizar atualizações na tabela PESSOA. Caso a relação obtida seja uma relação completa, o efeito produzido pela execução concorrente do processo de *commit* e do teste de validade seria equivalente ao caso em que são executados, seqüencialmente, primeiro o processo de *commit* e depois o teste de validade.

Caso a relação obtida seja incompleta, poderão existir nesta relação, tuplas antigas (com valores anteriores a atualização realizada pelo processo de *commit*) e tuplas com valores atualizados pelo processo de *commit* (oriundas do mecanismo de leitura de valores sob demanda). Um vez que a API JDBC especifica com flexibilidade a sensibilidade

dos objetos `ResultSet` [5] às modificações realizadas por transações concorrentes[6], este comportamento também não é justificativa para a imposição de exclusão mútua entre os testes de validade e o processo de *commit*. Por fim, como os testes de validade realizam somente leituras na tabela de modificações globais, não existe necessidade de imposição de exclusão mútua nos testes de validade entre si. Concluímos então que, apesar do uso da tabela de modificações globais, os testes de validade não constituem uma região crítica.

Na seção 2.1.3, foi dito que o mecanismo de atualização busca na tabela de modificações globais um valor atualizado para a célula da tupla que está sendo lida pela aplicação cliente. Esses valores atualizados são escritos nesta tabela na etapa 3 do processo de *commit* descrito acima. Vamos mostrar que a execução concorrente da busca na tabela de modificações globais efetuada pelo mecanismo de atualização e o processo de *commit* de transações pode acarretar o fenômeno de leituras sujas [50, 45, 46, 51].

Vamos supor a execução concorrente de duas transações `T1` e `T2`. A transação `T1` busca na tabela de modificações globais, o valor mais atualizado para a coluna `IDADE` de uma determinada tupla da tabela `PESSOA`. A transação `T2`, executando a etapa 3 do processo de *commit*, atualiza o valor da coluna `IDADE` da mesma tupla da tabela `PESSOA`. Digamos que `T2` realize a escrita na tabela de modificações globais antes de `T1` realizar a leitura. Assim, em `T1` é obtido o valor atualizado. Suponhamos então, que durante a etapa 5 do processo de *commit*, ocorra um erro de acesso ao banco de dados e a transação `T2` seja obrigada a realizar *rollback*. Conclusão: `T1` terá lido um valor atualizado pela transação `T2` que foi abortada, caracterizando assim a leitura suja. Desta forma, para evitar leituras sujas, é necessário que a busca de valores atualizados pelo mecanismo de atualização e o processo de *commit* das transações sejam realizados em exclusão mútua.

2.1.7 Níveis de isolamento

Já sabemos que o processo de *commit* de uma transação do driver dCache transfere as modificações visíveis apenas por esta transação para a tabela de modificações globais, onde outras transações em curso obterão valores mais atualizados para atender

[5] Os objetos da classe `java.sql.ResultSet` encapsulam as relações obtidas a partir da cache de relações. A partir desta relação encapsulada, provêem os dados solicitados pela aplicação cliente.

[6] A partir da versão 2.0 da API JDBC [48] é fornecida, através da interface `DatabaseMetaData`, uma forma de determinar a habilidade de um objeto `ResultSet` em detectar as modificações realizadas por outras transações. Por exemplo, os métodos `othersUpdatesAreVisible`, `othersDeletesAreVisible`, e `othersInsertsAreVisible` podem ser utilizados com este propósito. A questão em discussão está diretamente relacionada com o conceito de materialização de resultados discutido por [49].

às execução das suas consultas. Vamos mostrar que este comportamento dá margem a existência dos fenômenos de leituras que não se repetem e leituras fantasmas [50, 45, 46, 51] no driver dCache. Vamos novamente supor a execução concorrente de duas transações `T1` e `T2`. Vamos supor que essas transações realizem as seguintes operações, na ordem apresentada:

1. A transação `T1` executa um comando SQL do tipo SELECT, realizando a leitura da coluna `IDADE` de uma determinada tupla da tabela `PESSOA`. Digamos que a consulta seja "`SELECT IDADE FROM PESSOA WHERE ID=1`".
2. A transação `T2` executa um comando SQL do tipo UPDATE, atualizando a coluna `IDADE` da mesma tupla da tabela `PESSOA`. Digamos que o comando de escrita seja "`UPDATE PESSOA SET IDADE=23 WHERE ID=1`".
3. A transação `T2` realiza *commit*.
4. A transação `T1` repete a consulta realizada no passo 1.

Tomando como referência nossa base de dados de exemplo, a consulta realizada pela transação `T1` no passo 1 retornaria o valor 20. No passo 3, as modificações realizadas por `T2` são escritas na tabela de modificações globais, ou seja, existirá uma entrada nesta tabela indicando que o valor mais atualizado para coluna `IDADE` da tabela `PESSOA` para a chave primária 1 é 23. Quando, após o passo 4, a transação `T1` tentar utilizar o valor da coluna `IDADE` obtido a partir da repetição da consulta, o mecanismo de atualização irá dar preferência ao valor mais atualizado para esta célula, ou seja, 23. Com isso, a mesma transação `T1` obteve os valores 20 e 23 para a mesma célula do banco de dados, caracterizando o fenômeno das leituras que não se repetem.

Vamos agora supor que `T1` e `T2` executam os mesmos passos, a menos do passo 2, que será substituído por:

- A transação `T2` executa um comando SQL do tipo DELETE. Digamos que o comando seja "`DELETE FROM PESSOA WHERE ID=1`".

Ao executar o passo 1, o núcleo de processamento teria adicionado, na cache de relações, uma relação correspondente ao resultado da consulta realizada por `T1`. O *commit* da transação `T2`, no passo 3, atualizará o rótulo de tempo associado à tabela `PESSOA` na tabela de modificações globais. Quando, no passo 4, ao executar a consulta repetida por `T1`, o núcleo de processamento encontrar na cache de relações a relação adicionada anteriormente, será realizado um teste de validade. Graças à atualização do rótulo de tempo na tabela de modificações globais, o teste de validade detectará que a tabela `PESSOA` foi alterada e obrigará a invalidação da relação obtida da cache. Com isso, uma nova relação precisará ser gerada para atender a repetição da consulta. A nova

relação, já refletindo as alterações repassadas ao driver escravo pelo *commit* da transação `T2`, não conterá a tupla cuja chave primária vale 1. Conclusão: a mesma transação `T1` primeiramente obtém uma relação contendo uma tupla e depois obtém uma relação vazia, caracterizando o fenômeno das leituras fantasmas.

Como o nível de isolamento 3 do padrão ANSI (*Serializable*), além de não admitir leituras sujas, também não admite nem leituras que não se repetem, nem leituras fantasmas, podemos dizer que a arquitetura do driver dCache apresentada não dá suporte a este nível de isolamento. Como o nível de isolamento 2 do padrão ANSI (*Repeatable Read*), além de não admitir leituras sujas, também não admite nem leituras que não se repetem, podemos dizer que a arquitetura também não dá suporte a este nível de isolamento. Restaram ao driver dCache os níveis 0 (*Read Uncommitted*) e 1 (*Read Committed*). O nível 0 permite a ocorrência dos três fenômenos. Logo, a arquitetura do dCache dá suporte automaticamente a este nível de isolamento. Já o nível 1 não admite leituras sujas. Assim, o suporte a este nível de isolamento fica condicionado à imposição ou não, por parte da implementação do driver, de exclusão mútua entre o commit das transações e leitura da tabela de modificações globais pelo mecanismo de atualização.

2.1.8 Limitações arquiteturais

O driver dCache possui três limitações arquiteturais:

1. Apenas os níveis de isolamento 0 e 1 do padrão ANSI são suportados pela arquitetura do driver dCache (essa limitação já foi detalhada na seção 2.1.7);
2. Modificações no *schema* dos bancos de dados não são suportados pela arquitetura do driver dCache;
3. Todas as alterações no banco de dados precisam ser realizadas através da mesma instância do driver dCache.

Dissemos que o catálogo de metadados armazena informações sobre os *schemas* de bancos de dados que estão sendo acessados via driver dCache. A obtenção dessas informações é realizada através da leitura dos metadados do banco por intermédio do driver escravo. Uma vez que esta leitura é realizada no primeiro acesso a um *schema*, o driver dCache assume que já conhece a sua estrutura e não realiza re-leituras. Com isso, eventuais modificações no *schema* realizadas através de *Data Definition Language (DDL)* (como a criação de novas tabelas, acréscimo ou remoção de colunas de tabelas já existentes), não serão percebidas pelo driver. Desta forma, *schemas* acessados pelo driver dCache não podem sofrer alterações estruturais.

Os mecanismos de atualização e invalidações utilizados pelo driver dCache para manutenção da consistência entre as informações mantidas em cache e as informações armazenadas no SGBD pressupõem que o driver dCache tem conhecimento de todas as alterações que são solicitadas ao SGBD. Ou seja, uma instância do driver dCache não espera que, por exemplo, a sua aplicação cliente (ou uma segunda aplicação) realize uma conexão através de outro driver ou outra instância do driver dCache, ao mesmo SGBD, e realize alterações nos dados. Caso essas alterações fosses efetuadas, a instância do driver dCache não teria como atualizar ou invalidar os valores armazenados nas suas caches, pois não foi notificada sobre as alterações feitas por terceiros.

Modificações realizadas por procedimentos armazenados (*stored procedures*), funções e *triggers* também se enquadram em modificações que ocorrem à revelia do driver dCache. Logo, estes recursos não podem ser utilizados em conjunto com o driver dCache. Apesar do código dos procedimentos armazenados ficar armazenado dentro do banco de dados, a chamada a estes procedimentos é realizada através da API JDBC (mais especificamente por intermédio da classe `java.sql.CallableStatement`). Assim, uma solução que poderia ser adotada pelo driver dCache para evitar inconsistência na chamada a procedimentos armazenados seria esvaziar todas as caches e tabelas de modificações quando a aplicação cliente solicitasse a execução de um destes procedimentos. Essa solução viabilizaria o uso do driver por aplicações que dependem desse recurso, entretanto, dependendo da quantidade de vezes que os procedimentos armazenados seriam chamados, o desempenho da aplicação poderia ser prejudicado devido às faltas (*misses*) no acesso às caches que seriam geradas pelo esvaziamento dessas estruturas de dados.

2.1.9 Diferença entre as versões

A arquitetura centralizada descrita ao longo desta seção é baseada na arquitetura proposta por [40], porém possui duas diferenças. A primeira, se refere à cache de tuplas. Como dissemos anteriormente, uma vez processadas pelo parser de comandos SQL, as consultas dão origem a objetos mais facilmente manipuláveis pelo núcleo de processamento. Em [40], esses objetos são chamados de *consultas compiladas*. Enquanto na arquitetura atual a cache de tuplas é única para cada *schema* (figura 2.7), em [40] a cache de tuplas é única para cada consulta compilada.

Voltemos ao exemplo da consulta `C1 = "SELECT NOME, IDADE FROM PESSOA WHERE IDADE >= ?"`, usado na seção 2.1.1. O processamento de `C1` pelo parser

de comandos SQL dará origem a uma consulta compilada. Tanto as tuplas da relação `R1` (gerada por `C1` com o parâmetro `35`), quanto as tuplas da relação `R2` (gerada com o parâmetro `25`), na arquitetura definida por [40], poderão ser armazenadas na mesma cache de tuplas: a cache de tuplas associada à consulta `C1`. A execução de outra consulta `C3 = "SELECT NOME, IDADE FROM PESSOA"` dará origem a uma segunda consulta compilada, com sua própria cache de tuplas. É claro que a relação gerada pela execução de `C3` contém tuplas idênticas às de `R1` e `R2`. No entanto, essas tuplas idênticas não poderão ser reaproveitas, pois fazem parte da cache de tuplas associada a outra consulta compilada. Conseqüentemente, a cache de tuplas associadas às consultas compiladas `C1` e `C3` conterão tuplas repetidas.

Com a cache de tuplas única para cada *schema*, conforme definido pela arquitetura centralizada proposta pelo presente trabalho, diversas consultas compiladas podem compartilhar tuplas. Essa modificação traz três vantagens com relação a estrutura de caches usada em [40]:

- Elimina o desperdício de memória representado pelo armazenamento de tuplas repetidas em caches de consultas compiladas distintas;
- Uma vez que consultas compiladas distintas podem se beneficiar de tuplas já armazenadas na cache por consultas anteriores, aumenta a probabilidade de acerto na cache de tuplas, minimizando a necessidade de acesso ao driver escravo;
- Minimiza as faltas no acesso à cache de tuplas devido à invalidações. Em [40], a invalidação de uma consulta compilada eliminava todas as tuplas armazenadas na cache associada à consulta compilada invalidada. Com a nova arquitetura centralizada, as tuplas não são eliminadas da cache de tuplas quando uma relação é invalidada. Elas permanecem disponíveis para outras relações não invalidadas. Caso algum dos campos de uma tupla em cache tenha sofrido UPDATE, o mecanismo de atualização será capaz de manter o driver consistente.

A segunda diferença entre as versões se refere ao mecanismo de invalidações. Tomando como referência nossa base de dados de exemplo, é fácil perceber que a execução do comando `UPDATE PESSOA SET IDADE = 21 WHERE ID = 01` deveria causar a invalidação da relação obtida pela consulta `C4 = SELECT * FROM PESSOA WHERE IDADE = 20`. Em [40], o mecanismo de invalidações se preocupa em tratar apenas os comandos SQL do tipo INSERT e DELETE. Ou seja, invalidações que deveriam ser causadas por execuções de comandos SQL do tipo UPDATE são desconsideradas pelo driver, podendo gerar resultados inconsistentes para a aplicação cliente.

A nova arquitetura do driver dCache eliminou esta limitação utilizando para isso

o conceito de tabelas e colunas críticas. Utilizando este conceito, o driver é capaz de perceber a necessidade de invalidação da relação usada no exemplo acima, mantendo o driver em estado consistente. Segundo a definição apresentada anteriormente, a coluna `IDADE` é crítica para a consulta `C4`. A execução do comando UPDATE associaria um rótulo de tempo atualizado para a coluna `IDADE`, que recebeu um novo valor exatamente na tupla de interesse de `C4`. Ao executar novamente a consulta `C4`, o teste de validade verificaria que a coluna crítica `IDADE` foi modificada, e invalidaria corretamente a relação obtida para `C4` em uma execução anterior.

Vale notar que, enquanto a nova estrutura de caches apresentada confere ao driver um comportamento que, presumidamente, irá melhorar o seu desempenho com relação à implementação realizada em [40], o tratamento das invalidações devido a comandos SQL do tipo UPDATE colaborará para a degradação do desempenho do driver. Isso porque, com as invalidações devido a comandos SQL do tipo UPDATE, acertos equivocados na cache de relações se transformarão em invalidações, obrigando o driver dCache a recorrer ao driver escravo.

2.2 Implementação

A implementação da nova arquitetura centralizada do driver dCache, assim como a implementação de qualquer outro driver JDBC, consiste basicamente, na implementação das interfaces definidas pela API JDBC. Ao descrevermos a implementação do driver, entretanto, vamos preferir dar enfoque ao modelo de classes utilizado para representar as estruturas mais peculiares do driver dCache, em detrimento das classes mais diretamente relacionadas à conformidade com a API JDBC. Descreveremos, por exemplo, como foram implementadas as caches do driver, em detrimento da descrição da implementação de interfaces JDBC como a `java.sql.Connection` ou `java.sql.Statement`. A descrição da implementação realizada por esta seção dará enfoque, complementarmente, aos aspectos importantes para o entendimento das modificações que foram necessárias para a implementação da versão distribuída do driver dCache, implementação a qual será descrita na seção 2.4.

2.2.1 Principais classes e interfaces

Foi definida a interface `Cache` para representação de uma cache qualquer. A interface `Cache` possui apenas três métodos: `insert(CacheItem)`, `access(Object)` e

`clear()`. O primeiro, insere um novo item na cache. Os itens a serem armazenados na cache precisam implementar a interface `CacheItem`. O segundo, retorna um item armazenado em cache, dada a sua chave de acesso, e o terceiro, elimina todas as entradas da cache. A interface `CacheItem`, por sua vez, possui apenas um método (`getKey()`), responsável por retornar a chave de acesso associada àquele item. A interface `Cache` possui uma implementação `LocalCache`[7] construída com base na classe `LinkedHashMap` [52], fornecida pelo próprio *Java SE Development Kit* (JDK). O método `removeEldestEntry` da classe `LinkedHashMap` foi sobrescrito para fornecer às caches usadas no driver o comportamento *Least Recently Used* (LRU).

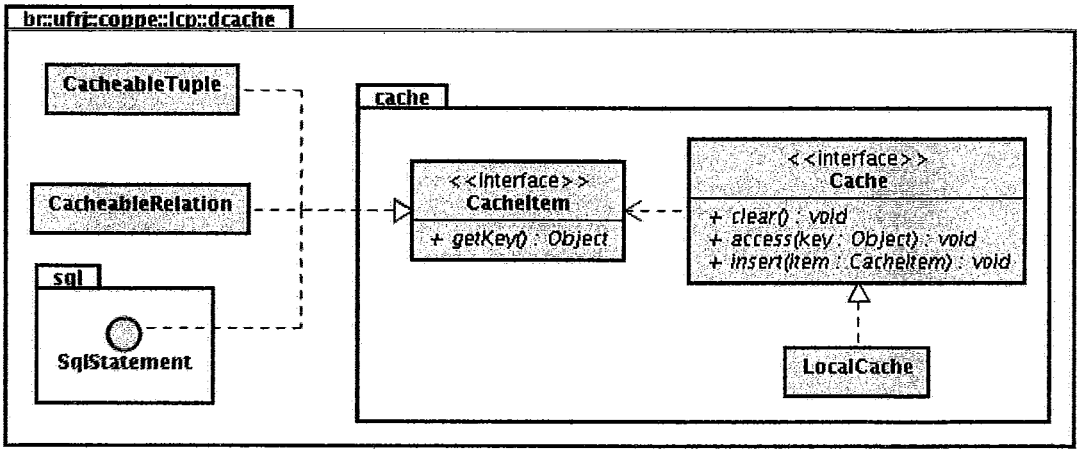


Figura 2.9: Implementando as três caches do driver dCache.

A figura 2.9 mostra a classe `LocalCache`, as interfaces `Cache` e `CacheItem`. Adicionalmente, ainda aparecem na figura duas implementações da interface `CacheItem` e uma interface herdeira, a serem explicadas em seguida. Na implementação do driver são utilizadas três instâncias da classe `LocalCache`: uma como cache de comandos SQL, uma como cache de relações e outra como cache de tuplas. Veremos agora como os comandos SQL, as relações e as tuplas são implementadas de forma que possam ser armazenadas nas instâncias da `LocalCache`.

O parser de comandos SQL possui um papel muito bem definido e pouco acoplado ao restante de toda a funcionalidade do driver. O parser de comandos SQL poderia ser visto como um serviço utilizado não só pelo driver, mas também por qualquer outro aplicativo que precisasse transformar texto SQL em objetos "compilados". Para simplificar a implementação, extensão e manutenção do driver, uma das primeiras modificações

[7] O prefixo *"Local"* foi usado para fazer distinção com a implementação de cache distribuída a ser descrita na seção 2.4.3.

realizadas, com relação a implementação fornecida por [40], foi a separação do parser de comandos SQL em um módulo independente. O único compromisso que o parser precisa assumir para o correto funcionamento do driver é que os objetos gerados a partir dos comandos SQL compilados pelo parser precisam implementar interfaces definidas pelo driver dCache. O driver dCache se comunica com o parser apenas através destas interfaces, não se importando com suas implementações. As interfaces são mostradas na figura 2.10.

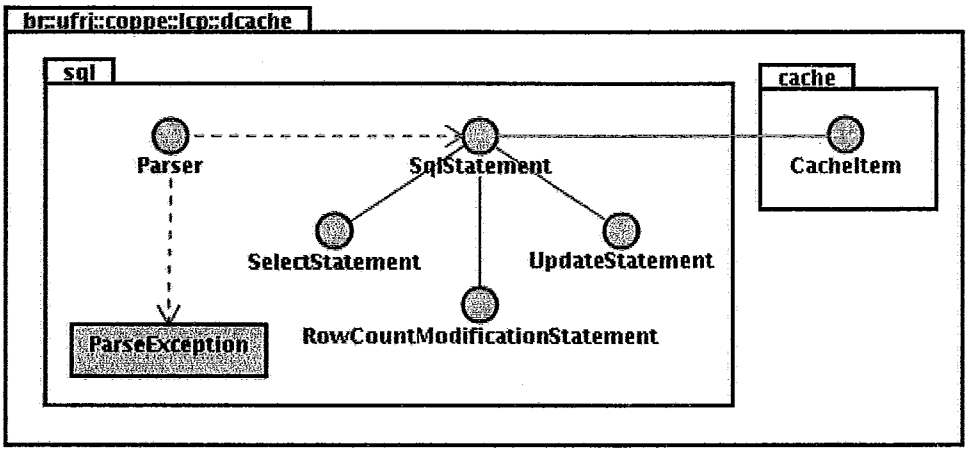


Figura 2.10: Interfaces utilizadas pelo driver dCache para comunicação com o parser de comandos SQL e a associação com a interface `CacheItem`.

A interface `Parser` possui um único método, que recebe um texto SQL e devolve uma implementação da classe `SqlStatement`, que representa um comando SQL compilado qualquer. Caso o texto enviado não possa ser compilado pelo parser, uma `ParserException` será lançada. As especializações da interface `SqlStatement`, `SelectStatement`, `UpdateStatement` e `RowCountModificationStatement`, representam, respectivamente, uma consulta SQL, um comando do tipo UPDATE e comandos DELETE ou INSERT. Cada uma dessas especializações define métodos usados pelo núcleo de processamento de comandos SQL para obter informações necessárias para o tratamento apropriado de cada um desses tipos de comandos. A interface `SelectStatement`, por exemplo, define métodos para obtenção das colunas e tabelas críticas da consulta SQL.

Para que os comandos SQL compilados pudessem ser armazenados na cache de comandos SQL, fizemos com que a interface `SqlStatement` estendesse a interface `CacheItem`. A implementação do método `getKey()` pelos comandos SQL compilados retorna a versão original (em forma de texto SQL) do comando compilado. Assim, antes

de solicitar ao parser a compilação de comando SQL submetido ao driver em forma de texto, o núcleo de processamento acessa a cache de comandos SQL usando como chave o próprio comando SQL submetido.

Para representar uma relação qualquer, foi a definida a interface `Relation`. Essa interface recebeu duas implementações fundamentais: a `TuplesRelation` e a `KeysRelation`. A primeira, representa uma relação de tuplas e a segunda, uma relação de ponteiros para tuplas (esses dois tipos de relação foram definidos na seção 2.1.1). Na primeira implementação, é encapsulada uma lista de objetos da classe `Tuple`, que representam uma tupla. Na segunda, é encapsulada uma lista de objetos da classe `TuplesCacheKey`, que representa um ponteiro para a cache de tuplas.

A definição de uma interface para a representação do comportamento de uma relação qualquer foi muito importante para a simplificação da implementação do restante do driver, principalmente dos *result sets*. A implementação da interface `java.sql.ResultSet` (da API JDBC) pelo driver dCache faz referência apenas à interface `Relation`, não se importando se a relação é de tuplas ou de ponteiros para tuplas. A figura 2.11 mostra a interface `Relation`, suas implementações e a sua associação com a interface `java.sql.ResultSet`. A figura também mostra a implementação `CacheableRelation` que será explicada em seguida.

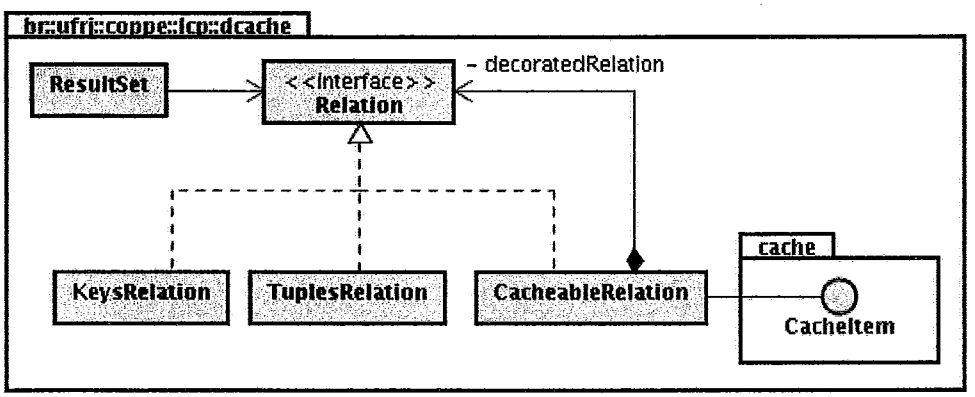


Figura 2.11: Interfaces `Relation` e suas implementações.

De forma geral, queremos que as relações e tuplas possam ser armazenadas em instâncias da `LocalCache`. Por outro lado, vimos na seção 2.1.5 que o mecanismo de suspensão temporária do uso da cache pode decidir que em determinadas circunstâncias, relações e suas tuplas não sejam armazenadas em cache. Por isso, diferentemente da estratégia adotada no caso dos comandos SQL, não fizemos a interface `Relation` estender a interface `CacheItem` e nem fizemos a classe `Tuple` implementar diretamente

a interface `CacheItem`. Preferimos utilizar o *design pattern Decorator* [53], que permite, em tempo de execução, acrescentar novas características a uma instância. Podemos por exemplo, em tempo de execução transformar uma instância de `TuplesRelation` ou `KeysRelation`, em uma relação pronta para ser armazenada em cache.

Para representar relações que podem ser armazenadas na cache de relações criamos então a classe `CacheableRelation`. Esta classe também implementa a interface `Relation` e adicionalmente, implementa a interface `CacheItem`. A classe `CacheableRelation` encapsula uma outra relação acrescentando-lhe a capacidade de armazenamento em cache. A implementação do método `getKey()` por esta classe retorna uma instância da classe `RelationsCacheKey`, responsável por encapsular uma consulta SQL parametrizada e os seus argumentos. Ou seja, cada instância da classe `RelationsCacheKey` é uma chave de acesso à cache de relações (conforme explicado na seção 2.1.1).

Analogamente, criamos a classe `CacheableTuple` (já apresentada na figura 2.9), que encapsula uma tupla acrescentando-lhe a capacidade de armazenamento na cache de tuplas. A implementação do método `getKey()`, por sua vez, retorna agora uma instância da classe `TuplesCacheKey`, que representa a chave de acesso à cache de tuplas.

2.2.2 Relógio global e sincronização

Na seção 2.1.3, vimos que o teste de validade utilizado pelo mecanismo de invalidações se baseia na comparação de rótulos de tempo associados, pelo núcleo de processamento, às tabelas e colunas modificadas (durante o processo de *commit* de uma transação) e às relações (no momento em que são armazenadas na cache de relações). A geração de rótulos de tempo se dá através de uma entidade que chamaremos de *gerador de rótulos de tempo* ou *relógio global* (o termo *global* faz alusão ao fato de que os rótulos de tempo gerados são únicos para todas as transações em curso no driver).

Em [40], a implementação do gerador de rótulos de tempo consiste apenas na chamada ao método `System.currentTimeMillis()` da API *Java Standard Edition* (JSE). Este método retorna a hora atual em milissegundos e sua precisão depende da granulosidade utilizada pelo sistema operacional no qual a Java Virtual Machine (JVM) está em execução [52]. Esta implementação de relógio global foi rejeitada pois a precisão fornecida não estava sendo capaz de garantir a geração de rótulos de tempo sempre distintos para atender à demanda gerada durante a execução dos testes unitários implementados

para avaliar a corretude do driver[8].

A despeito da invocação ao método `currentTimeMillis` retornar a hora atual do sistema, a única exigência feita pela arquitetura do driver dCache é que os rótulos de tempo gerados pelo relógio global obedeçam a alguma relação de ordem total. Assim, na nova implementação, estes rótulos de tempo são gerados por um relógio lógico[9], implementado por um contador global (uma variável *public static* na classe `Driver`), desta forma, único para todas as transações em curso no driver.

Na seção 2.1.6 vimos que a execução do *commit* das transações em regime de exclusão mútua é fundamental para manter a consistência do driver e garantir a atomicidade dos *commits*. Entretanto, a implementação fornecida por [40] observou que, apesar da tabela de modificações globais ser um recurso compartilhado entre transações, o risco de inconsistência entre transações só existe quando duas transações realizam escritas na mesma coluna ou tabela de um banco de dados. Então, para aumentar o grau de paralelismo entre transações no driver, tanto a implementação fornecida por [40] quanto a implementação fornecida pelo presente trabalho, realizam o *lock* (travamento) de colunas e tabelas individuais.

Isso significa que transações que estão realizando, por exemplo, *commit* de duas operações de inserção em duas tabelas `A` e `B` distintas podem executar em paralelo. Para tanto, a implementação do driver dCache associa a cada tabela e coluna existente em um *schema* de banco de dados, uma instância da classe `java.util.concurrent.locks.ReentrantLock` [52]. Antes de iniciar o processo de *commit*, a transação tentar adquirir o *lock* das colunas e tabelas do seu interesse. Ao final do processo, a transação libera os *locks* que estavam em sua posse.

2.2.3 Limitações da implementação

Vimos, nas seções 2.1.7 e 2.1.8, que a arquitetura centralizada do driver dCache descrita dá suporte apenas aos níveis de isolamento 0 e 1 do padrão ANSI, e que o suporte ao nível 1, fica condicionado à garantia de exclusão mútua entre o processo de *commit* e a leitura da tabela de modificações globais. Como essa exclusão mútua não foi implementada pelo presente trabalho, podemos citar como a primeira limitação da implementação, a restrição

[8]Para garantir a corretude do driver, além dos experimentos utilizando o benchmark TPC-W (que serão detalhados no capítulo 3) também foram implementados 70 testes unitários através do *framework* JUnit [54].

[9]Um relógio lógico é um mecanismo utilizado para a gerenciar relacionamentos cronológicos e causais em um sistema distribuído [55].

do nível de isolamento suportado pelo driver dCache a apenas o nível 0 do padrão ANSI (*Read Uncommitted*).

A segunda limitação do driver dCache se refere à sintaxe SQL entendida pelo driver. Como foi dito, o driver dCache é composto por um parser de comandos SQL que transforma o texto recebido através da API JDBC em objetos manipulados pelo restante do driver. Esse processo de tradução utiliza a gramática apresentada no apêndice A de [40], que admite apenas um subconjunto de comandos SQL previstos pelo padrão SQL-92 [50].

Dentre as restrições, estão a impossibilidade de uso de subconsultas (comandos SELECT aninhados, ou encadeados), a impossibilidade de uso do operador NOT em expressões booleanas, a impossibilidade de uso de funções de agregação dentro da cláusula ORDER BY e a restrição do formato dos comandos UPDATE ao padrão `UPDATE TABELA_A SET` *`COLUNAS_DE_A`* `WHERE` *`CACHE_PRIMARIA_DE_A`*.

No formato, *`COLUNAS_DE_A`* representa uma lista de atribuições do tipo `COLUNA_1 = ?, COLUNA_2 = ?, ..., COLUNA_N = ?` onde `COLUNA_x` é uma coluna da tabela `TABELA_A` e *`CACHE_PRIMARIA_DE_A`* representa uma lista de igualdades do tipo `CHAVE_1 = ?, CHAVE_2 = ?, ..., CHAVE_N = ?` onde `CACHE_x` é uma das colunas pertencentes a conjunto de N colunas componentes da chave primária da `TABELA_A`.

A terceira e última limitação se refere à implementação da API JDBC. A implementação atual do driver dCache implementa apenas um subconjunto dos métodos definidos pela API JDBC 1.0. Ou seja, o driver dCache não pode ser utilizado por aplicações que precisam de recursos disponibilizados por versões mais recentes da API JDBC e nem por aplicações que só dependem da API JDBC 1.0 porém, utilizam métodos desta API que ainda não foram implementados pelo driver dCache.

2.3 Arquitetura distribuída

Quando descrevemos a segunda limitação arquitetural do driver dCache (seção 2.1.8), dissemos que "todas as alterações no banco de dados precisam ser realizadas através da mesma instância do driver dCache". Nesta seção, veremos como a arquitetura do driver pode ser modificada para permitir que várias instâncias do driver se mantenham consistentes. Essa sofisticação permitirá, por exemplo, que mais de uma aplicação acesse o mesmo SGBD, usando o driver dCache, sem risco de inconsistências. Como a arquitetura modificada permitirá que várias instâncias do driver dCache executem em nós de processamento

com memórias independentes, chamaremos-na de *arquitetura distribuída*.

A figura 2.2 mostrou as camadas de um sistema que se beneficia da arquitetura centralizada do driver dCache, descrita até o presente momento. Na figura 2.12 vemos um sistema que se beneficia da arquitetura distribuída. No primeiro nível, existem agora N aplicações (com códigos executáveis distintos ou apenas instâncias diferentes de um mesmo executável), que acessam N instâncias do driver dCache no segundo nível. As instâncias do driver dCache se comunicam para manterem seus estados consistentes, e acessam instâncias do driver JDBC escravo no terceiro nível. No quarto nível, inalterado, permanece o SGBD, sempre acessado apenas pelo driver escravo.

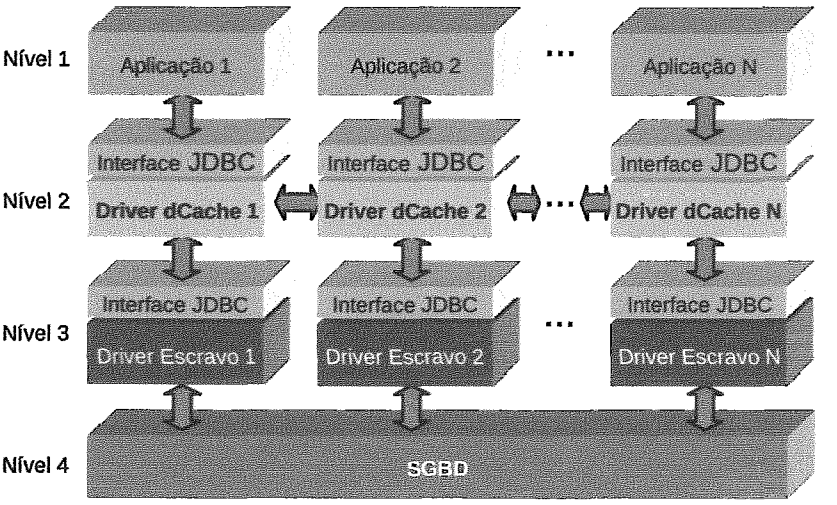


Figura 2.12: Camadas de um sistema usando a versão distribuída do driver dCache.

Os três componentes do driver dCache permanecem inalterados na arquitetura distribuída. Cada instância do driver dCache possuirá seu próprio catálogo de metadados de bancos de dados, seu próprio parser de comandos SQL e seu próprio núcleo de processamento. Vimos que o núcleo de processamento conta com cinco estruturas de dados: a cache de comandos SQL, a cache de relações, a cache de tuplas, a tabela de modificações globais e a tabela de modificações locais. Vimos que as quatro primeiras estruturas são compartilhadas por todas as transações em curso no driver, enquanto que a tabela de modificações locais contém informações úteis para uma única transação. Uma vez que transações executando em diferentes instâncias do driver dCache não terão acesso a mesma memória, precisaremos analisar que tratamento deverá ser dispensado a cada uma das quatro estruturas compartilhadas para que as instâncias não fiquem inconsistentes.

2.3.1 Caches distribuídas

Na arquitetura centralizada do driver dCache, todas as transações em curso no driver lêem e escrevem na mesma cache de comandos SQL, na mesma cache de relações e na mesma cache de tuplas. O compartilhamento das entradas dessas três caches entre as diversas transações em curso no driver permite que cada transação se aproveite do resultado do processamento efetuado em benefício das outras transações, melhorando o desempenho do driver (conforme mostrado em [40]). Na arquitetura distribuída, gostaríamos que todas as transações também tivessem acesso ao resultado do processamento efetuado em benefício das demais.

Uma abordagem possível para o tratamento das caches na arquitetura distribuída, seria manter cada instância com suas próprias caches locais (comandos SQL, relações e tuplas) e replicar o conteúdo de cada uma delas nas demais instâncias do driver. Ou seja, quando uma instância realiza uma escrita em cache, essa escrita é replicada por todas as instâncias do driver, que ficarão sempre com o mesmo conteúdo. Assim, em qualquer instância, a imagem das caches seria a mesma a todo instante.

Neste ponto, vale observar que a distribuição do driver dCache por vários nós de processamento, cada um com sua própria unidade de memória, pode ser vista como uma estratégia de aumento da quantidade total de memória disponível para o driver. Sabemos que o aumento da capacidade de uma cache pode melhorar o desempenho do sistema em decorrência do aumento na taxa de acertos propiciada pelas entradas disponíveis adicionais [56]. A abordagem apresentada, entretanto, além de gastar tempo de processamento replicando as escritas em todas as instâncias, não aproveita a memória adicional devido ao armazenamento de entradas repetidas em todas as instâncias do driver.

Optamos então por uma outra abordagem, que visa aproveitar melhor a memória disponibilizada pelos nós de processamento e evita a réplica das operações de escrita. Segue o algoritmo adotado: Toda vez que uma cache de uma instância receba a solicitação de um dado por parte de um cliente, procede da seguinte forma:

- Caso a instância possua o dado solicitado na sua cache local, atende ao pedido do cliente.
- Caso contrário:
  - Verifica se alguma outra instância possui o dado solicitado e em caso positivo:
    - atualiza a cache local com o item obtido remotamente;
    - atende o cliente com o item obtido remotamente.

- Caso contrário:
    * atualiza a cache local;
    * atende o cliente.

Outros algoritmos poderiam ter sido eleitos para realizar a comunicação entre as caches e poderão ser explorados em trabalhos futuros. O algoritmo apresentado possui as seguintes características: requer uma implementação simples, realiza pouca comunicação entre as caches e minimiza a interferência de uma cache no conteúdo das demais. Dizemos que o algoritmo realiza pouca comunicação pois uma cache remota é acessada somente quando um dado solicitado não pode ser encontrado na cache local. Dizemos que o algoritmo minimiza a interferência de uma cache no conteúdo nas demais pois o algoritmo só utiliza caches remotas para consulta, nunca atualizando-as. O algoritmo apresentado pode ser utilizado em todas as três caches existentes no driver dCache, sem risco de inconsistência entre as caches, mesmo sem a propagação das operações de escrita pelas diversas instâncias.

No caso da cache de comandos SQL, não existe risco de inconsistência pois o processo de tradução realizado pelo parser é determinístico. Ou seja, dado um comando SQL em forma de texto, a representação gerada pelo parser será sempre a mesma, qualquer que seja a instância que efetue a tradução e qualquer que seja o momento no qual a tradução é realizada. Assim, quando uma instância não encontra um comando SQL em cache, pode adicioná-lo apenas localmente, sem risco de inconsistência [10]. No caso das caches de relações e de tuplas, é importante observar que a validade dos dados nelas armazenados é controlada pelos mecanismos de atualizações e invalidações, os quais se baseiam na tabela de modificações globais. Ou seja, se a tabela de modificações globais é consistente para todas as instâncias, a validade dos dados das caches de relações e tuplas está garantida mesmo que estes dados sejam diferentes na cache de cada instância.

2.3.2 Tabela de modificações globais distribuída

Obviamente, qualquer instância do driver pode executar comandos de escrita sobre qualquer *schema*; e após o *commit* das transações responsáveis por essas escritas, as modificações realizadas precisam ser visíveis por qualquer outra transação, independentemente da outra transação estar ou não executando na mesma instância do driver. Vimos que a

[10] Veremos na seção 2.4.3 que, devido à expectativa de que processo de tradução realizado pelo parser seja barato computacionalmente, na implementação do driver foi preferida a utilização da versão não-distribuída da cache de comandos SQL.

tabela de modificações globais do driver dCache armazena os valores mais atualizados para as células modificadas pelas transações, armazena os rótulos de tempo que indicam quando foram realizadas essas atualizações, e armazena os rótulos de tempo que indicam o momento em que foram inseridas ou removidas tuplas das tabelas. Essas informações são fundamentais para os mecanismos de atualização e invalidação que, por sua vez, impedem que o núcleo de processamento de comandos SQL utilize valores desatualizados, oriundos da leitura das caches de relações e de tuplas.

Gostaríamos que, na arquitetura distribuída, os mecanismos de atualização e invalidação continuassem desempenhando suas funções corretamente, apesar do fato de cada instância do driver dCache possuir seu próprio núcleo de processamento. Ou seja, ao contrário da cache de comandos SQL, da cache de relações e da cache de tuplas, a imagem da tabela de modificações globais precisa ser única para todas as instâncias do driver na arquitetura distribuída. Com isso, independentemente de qual a instância responsável pela execução de um comando de escrita, o mecanismo de atualização conhecerá os valores mais atualizados para as colunas do banco de dados e o mecanismo de invalidações conhecerá os rótulos de tempo mais atualizadas para as colunas e tabelas alteradas.

2.3.3 Sincronização distribuída

O compartilhamento das caches e da tabela de modificações globais do driver dCache pelas diversas instâncias do driver traz à tona o problema da sincronização distribuída. Em outras palavras, como tratar, na arquitetura distribuída do driver, as regiões críticas tratadas na arquitetura centralizada? A arquitetura distribuída continua admitindo mais de uma transação em execução concorrente em uma mesma instância do driver, mais também admite, adicionalmente, transações em execução paralela entre mais de uma instância do driver. No caso das caches, vimos que o acesso concorrente não traz problemas de sincronização do ponto de vista arquitetural. Na arquitetura distribuída, as caches também não precisarão cuidados especiais. Isso porque, do ponto de vista de uma mesma instância, não existe diferença, entre as duas arquiteturas, na forma de acesso às caches; e entre instâncias, o acesso realizado é de leitura somente.

Já o acesso a tabela de modificações globais precisa ser analisado com mais cuidado. Na arquitetura centralizada, vimos que o processo de *commit* precisa ser realizado em exclusão mútua pois a escrita concorrente na tabela de modificações globais pode deixar o driver em um estado inconsistente. Na arquitetura distribuída, podem existir transações em instâncias diferentes realizando o processo de *commit* em paralelo. Como

a imagem da tabela de modificações globais é única para todas as instâncias, o acesso a esta região crítica também precisa ser realizado em exclusão mútua entre instâncias. A solução adotada para este problema foi utilizar um mecanismo de travamento (*lock*) distribuído.

2.3.4 Relógio global

Na arquitetura centralizada, a geração de rótulos de tempo pode ser realizada por um relógio lógico, implementado, por exemplo, através de um contador global, ou seja, único para todas as transações em curso na instância única do driver. Vimos que, na arquitetura distribuída, a tabela de modificações globais (responsável por armazenar os rótulos de tempo) possui imagem única para todas as instâncias do driver. Também foi dito que as relações (as quais contém seus respectivos rótulos de tempo) presentes na cache de relações podem ser compartilhadas por todas as instâncias do driver.

Se cada instância do driver tivesse seu próprio relógio lógico, as estruturas de dados mencionadas conteriam informações com rótulos de tempo baseados em referenciais distintos. A falta de um referencial de tempo único daria margem à obtenção de resultados equivocados nos testes de validade efetuados pelas instâncias do driver dCache. A obtenção de resultados equivocados nos testes de validade implica na indesejável utilização de dados desatualizados vindos das caches de relações e de tuplas. Concluímos então que, para evitar esse problema, a implementação da arquitetura distribuída do driver dCache precisará contar com um relógio global, capaz de gerar rótulos de tempo baseados em um referencial único para todas as instâncias do driver.

2.3.5 Catálogo de metadados distribuído

O compartilhamento dos dados presentes nas caches do driver dCache pelas diversas instâncias não é um comportamento obrigatório para o correto funcionamento da arquitetura distribuída. O compartilhamento desses dados é uma opção de projeto que visa aumentar a taxa de acertos na consulta a essas caches. Já o uso de uma imagem única para a tabela de modificações globais, a sincronização distribuída e o uso de um relógio global são fundamentais para a corretude de uma implementação distribuída do driver dCache. Dissemos que, assim como o parser de comandos SQL e o núcleo de processamento de comandos SQL, o catálogo de metadados de bancos de dados é único para cada instância do driver, mas ainda não havíamos feito uma análise mais detalhada deste componente.

Como cada instância tem seu catálogo de metadados particular, mesmo que uma instância já tenha realizado a leitura dos metadados de um determinado banco, uma outra instância que venha a acessar o mesmo banco precisará reler essas informações. Poderíamos pensar em também compartilhar, entre instâncias do driver, as informações do catálogo de metadados. Assim, uma vez que uma instância já tivesse realizado a leitura dos metadados de um *schema*, as demais instâncias não precisariam realizar um novo acesso ao driver escravo para obtê-las. Poderiam, na verdade, lê-las de um catálogo de metadados distribuído, com imagem única para todas as instâncias. Note que este comportamento não é obrigatório para o correto funcionamento do driver. A leitura dos metadados a partir do driver escravo, que é determinística[11], e o seu posterior armazenamento em catálogos particulares não gera inconsistência no driver dCache. Como a leitura desses metadados é realizada apenas uma vez por cada instância (para cada *schema* acessado), achamos que o compartilhamento do catálogo de metadados não traria melhoria de desempenho e, por isso, não foi realizado.

2.4 Implementação distribuída

Por simplicidade de projeto e implementação, optamos por agregar, na arquitetura distribuída do driver dCache, um servidor, que chamaremos de *servidor dCache*. Assim, quando falarmos da arquitetura distribuída do driver, estaremos assumindo a existência, não só das instâncias do driver, da aplicação (ou aplicações) cliente e do SGBD, mas também do servidor dCache. O servidor dCache acumula as seguintes responsabilidades:

- Centralizar as informações de configuração das instâncias do driver tornando-se um ponto de referência para qualquer nova instância inicializada e facilitando o gerenciamento, pelos usuários, das configurações do driver. Quando uma instância do driver é inicializada, ela conhece apenas o endereço do servidor dCache e a porta *Transmission Control Protocol* (TCP) à qual deve se conectar para obter informações de configuração que a farão apta a começar o atendimento à aplicação cliente.
- Solicitar às instâncias, relatórios sobre o seu funcionamento e arquivá-los, disponibilizando-os assim, para análise posterior pelos desenvolvedores ou usuários do driver.
- Fornecer rótulos de tempo para as instâncias do driver.

[11] Vale lembrar que na seção 2.1.8 assumimos que a arquitetura do driver dCache não admite modificações nos *schemas* dos bancos de dados

- Gerenciar *locks* distribuídos.

As configurações do driver são explicadas em maiores detalhes no apêndice B. Os relatórios do driver, bem como os processos de geração e coleta destes relatórios são explicados no apêndice A e no capítulo 3, seção 3.2. No restante desta seção, falaremos sobre a questão da geração de rótulos de tempo, dos *locks* distribuídos e sobre outras questões sobre a implementação da arquitetura distribuída do driver.

2.4.1 Relógio global

Como, na arquitetura centralizada do driver dCache, todas as transações em curso residem na mesma instância do driver, a implementação desta arquitetura pode utilizar como relógio global uma variável global inteira incrementada toda vez que um novo rótulo de tempo se faz necessário. Já na arquitetura distribuída, não podemos utilizar uma variável global pois as transações residem em nós de processamento com memórias independentes.

Na implementação da arquitetura distribuída do driver, transferimos o papel de relógio global para o servidor dCache. O servidor dCache mantém um *socket* TCP aberto, aguardando conexões e requisições de rótulos de tempo oriundas das instâncias do driver. Quando qualquer instância do driver é iniciada, obtém informações de configuração a partir do servidor dCache, descobrindo à qual porta TCP deve se conectar no servidor dCache para obter rótulos de tempo. Através desta conexão, a instância envia solicitações e recebe rótulos de tempo. Cada solicitação recebida pelo servidor dCache faz com que seja incrementado um contador implementado em uma variável inteira no servidor. O novo valor do contador é retornado pela conexão como resposta à solicitação da instância.

Inicialmente, utilizamos *sockets* TCP para realizar a implementação, exatamente como descrito acima. Escolhemos o protocolo TCP para nos beneficiarmos das características de ordenação e confiabilidade importantes para garantir a relação de ordem esperada pela arquitetura do driver na geração de rótulos de tempo. Nesta ocasião, o desempenho apresentado pelo driver foi tão baixo que era mais vantajoso não utilizar a cache no servidor de comércio eletrônico, contrariando os resultados apresentados por [40]. Realizamos então uma nova implementação, desta vez, utilizando *sockets User Datagram Protocol* (UDP).

Usando TCP, o tempo médio para que uma instância do driver dCache conseguisse um rótulo de tempo a partir do servidor dCache era de 40 milissegundos, com variações desde alguns poucos milissegundos a até mais de 200 milissegundos. Com

	tpcw-10K-25	tpcw-10K-100
BM	4.67%	3.27%
SM	4.78%	8.09%
OM	28.13%	44.01%

Tabela 2.1: Ganhos de desempenho (em WIPM) obtidos após a utilização do protocolo UDP na implementação do relógio global.

UDP, o tempo médio caiu para cerca de 1 milissegundo, atingindo no máximo, a casa dos 100 milissegundos. Graças a este aprimoramento, o desempenho do driver melhorou em até 44% com relação a implementação usando TCP. Este ganho de desempenho foi suficiente para que o uso da cache voltasse a ser vantajoso com relação a não-utilização do driver.

A tabela 2.1 mostra o ganho percentual obtido pela implementação com *sockets* UDP em cima da implementação com *sockets* TCP, utilizando-se o benchmark TPC-W. Os códigos *tpcw-10K-25*, *tpcw-10K-100*, *BM*, *SM* e *OM* referem-se a parâmetros de configuração do benchmark e serão explicados em detalhes no capítulo 3, quando falarmos sobre a avaliação experimental do dCache.

Apesar do melhor desempenho alcançado com o uso do protocolo UDP, não poderíamos abrir mão das características de comunicação confiável e ordenação fornecidas pelo protocolo TCP, caso contrário, estaríamos pondo em risco a consistência do driver. Para contornar o problema, transferimos então, a responsabilidade de garantir essas características imprescindíveis para a camada da aplicação. Para a solicitação de rótulos de tempo ao servidor dCache as instâncias do driver utilizam um *UDPChannel*. O *UDPChannel* é uma implementação de uma abstração de canal de comunicação confiável realizada em cima de UDP. O *UDPChannel* é fornecido pelo pacote *JNetwork*, desenvolvido especialmente para a implementação da versão distribuída do driver dCache[12].

2.4.2 Sincronização distribuída

Vimos que, na implementação da arquitetura centralizada do driver dCache (seção 2.2.2), instâncias da classe `ReentrantLock` (da API JDK) são utilizadas para travar as tabelas e colunas que serão modificadas durante o *commit* de uma transação. A classe `ReentrantLock`, entretanto, só é capaz de bloquear a execução de *threads* pertencentes

[12]Para maiores detalhes sobre o pacote JNetwork, consulte o apêndice D.

à mesma JVM. No caso da arquitetura distribuída do driver, precisaremos interromper a execução de *threads* de diferentes instâncias do driver, ou seja, *threads* que executam em máquinas virtuais distintas (sobretudo, em nós de processamento com memórias independentes). Logo, no caso da arquitetura distribuída, precisaremos utilizar algum mecanismo de sincronização distribuída.

Além disso, enquanto na implementação da arquitetura centralizada optamos por travar colunas e tabelas individualmente para aumentar o grau de paralelismo entre as transações que realizam *commit*, na arquitetura distribuída, optamos por utilizar apenas um *lock* para garantir exclusão mútua no processo de *commit*. Isso porque, utilizando-se um mecanismo de sincronização distribuída, o custo de aquisição de um *lock* se torna grande o suficiente para que o aumento de paralelismo proporcionado pelo travamento de colunas e tabelas individuais acabe não sendo compensador.

Travando colunas e tabelas individuais, são gastos, em média, 1,2 milissegundos para entrar na região crítica do processo de *commit*, enquanto utilizando-se um único *lock* para proteger a região crítica, este tempo cai para 0,45 milissegundos. Realizando experimentos com o benchmark TPC-W (descrito em detalhes no capítulo 3), verificamos que esta diminuição no tempo de travamento proporciona, aproximadamente, 8% mais interações Web por minuto no servidor de comércio eletrônico avaliado[13].

O *toolkit JGroups* [57] fornece, dentre os seus *building blocks*, um gerenciador de locks distribuídos[14]. Tentamos utilizar este gerenciador de *locks* na implementação da arquitetura distribuída do driver, mas a quantidade de memória disponível nas máquinas utilizadas para a avaliação do driver dCache foi insuficiente para esta solução[15], mesmo utilizando-se um único *lock* para garantir a exclusão mútua no processo de *commit*.

Tentamos também utilizar como solução para o problema da sincronização distribuída, os *locks* distribuídos fornecidos pelo pacote *Java Inter-Process Communication (JIPC)* [58]. O JIPC é uma biblioteca que fornece primitivas para comunicação distribuída em Java e foi desenvolvido por um indivíduo, sem fins lucrativos. Essa solução entretanto

[13]Vale notar que, para realizar estas medições, foram utilizadas as versões 2.10.2.re (travamento individual) e 2.11.0.re (travamento único) do driver dCache. Nenhuma destas versões corresponde a versão final (2.16.0), utilizada no capítulo 3 para avaliação do dCache. Por isso, o tempo de travamento reportado naquele capítulo é diferente dos números informados aqui.

[14]Para maiores detalhes sobre o JGroups, consulte o apêndice C.

[15]O erro `java.lang.OutOfMemoryError` foi reportado pelo servidor Tomcat após, aproximadamente, 1 hora de execução do benchmark TPC-W utilizando as configurações tpcw-10K-25 e tpcw-10K-100, para diferentes perfis de navegação e com *heap* máximo das JVMs dos Tomcats definido em 512 MB. Maiores detalhes sobre o servidor Tomcat, sobre benchmark utilizado, sobre as configurações das máquinas e as configurações do benchmark podem ser encontrados no capítulo 3.

foi descartada pois possuía erros de programação e não possuía documentação suficiente. A biblioteca foi obtida a através de *download* a partir do site do desenvolvedor, em meados de abril de 2007 e, nesta ocasião, não possuía versionamento.

Partimos então, para o desenvolvimento de uma solução própria para sincronização distribuída, incluindo no pacote JNetwork um gerenciador de *locks* distribuídos (ver apêndice D). O gerenciador de *locks* distribuídos desenvolvido baseia-se em um servidor de *locks*, responsável por receber pedidos de clientes (no caso, instâncias do driver dCache) interessados em adquirir e liberar *locks*. O servidor de *locks* conhece o estado de todos os *locks* existentes, decidindo então, qual dos clientes receberá a posse do *lock*. Assim como no caso do relógio global, a implementação do gerenciador de *locks* faz uso de *UDPChannels*, fornecidos pelo próprio pacote JNetwork.

2.4.3 Caches distribuídas

As caches distribuídas utilizadas na versão distribuída do driver dCache foram implementadas através da classe `DistributedCache`. A classe `DistributedCache` também implementa a interface `Cache` (já apresentada na seção 2.2), porém, ao contrário da implementação `LocalCache`, não se limita a procurar os itens solicitados apenas em um campo `LinkedHashMap` local. Esta nova implementação também faz uma busca em um campo deste tipo, porém, quando o item solicitado não é encontrado, parte para a sua busca em caches localizadas em outras instâncias do driver (caches remotas).

A comunicação com as caches remotas é implementada através de *sockets* UDP. Quando uma `DistributedCache` é instanciada, recebe como argumento, através do seu construtor, uma lista de endereços das demais instâncias do driver. Quando um item não pode ser encontrado localmente, é iniciada a sua busca nas instâncias cujos endereços foram informados. A lista de endereços é percorrida seqüencialmente, até que o item seja encontrado em uma cache remota ou até que a lista se esgote.

Para cada endereço, a cache envia uma mensagem de solicitação. A cache remota tem um intervalo de 40 milissegundos para responder a solicitação. Caso a cache remota responda que também não possui o item solicitado (ou não responda dentro do intervalo fixado) parte-se para o próximo endereço. No caso de uma resposta positiva de uma cache remota, o item (que vem encapsulado dentro da própria mensagem positiva) é armazenado na cache local e retornado para a aplicação, a qual não tem como saber se este é proveniente da cache local ou de uma cache remota.

É interessante notar que, uma vez que uma mesma instância do driver dCache é

capaz de fornecer à aplicação cliente diversas conexões concorrentes com o banco de dados, a implementação das caches precisa ser *thread-safe* (precisa permanecer consistente com várias *threads* executando concorrentemente a implementação de seus métodos). Dentre os cuidados tomados pela a implementação da `DistributedCache` está a utilização de uma lista de *threads* em espera.

Sempre que uma *thread* solicita à cache um item que não pode ser encontrado localmente, esta *thread* se registra na lista de espera e começa a realizar as solicitações seqüenciais às caches remotas, conforme explicado anteriormente. Após cada solicitação, a *thread* se põem em suspensão (*sleep*), aguardando a resposta da cache remota. A implementação da `DistributedCache` por sua vez, conta com uma *thread* auxiliar, responsável exclusivamente por receber as respostas oriundas das caches remotas. Cada resposta recebida pela *thread* auxiliar é guardada em um *buffer* de respostas recebidas e todas as *threads* na lista de espera são acordadas.

Quando a *thread* é acordada, verifica se o *buffer* de respostas contém uma resposta para a sua última solicitação (o casamento entre pedidos e respostas é feito por rótulos inteiros associados às mensagens). Caso a resposta não seja encontrada, a *thread* volta a dormir, até sua resposta chegar ao *buffer* ou o intervalo de 40 milissegundos terminar. Quando um dos dois eventos ocorre, a *thread* se retira da lista de espera e continua sua execução deixando o código da cache.

Uma mensagem de resposta que chega ao *buffer* depois que a *thread* responsável "desistiu" de aguardá-la, ficaria eternamente abandonada no *buffer*. Para evitar o acúmulo de mensagens abandonadas, a implementação da `DistributedCache` ainda conta com mais outra *thread* auxiliar, responsável por periodicamente eliminar as mensagens abandonadas.

A `DistributedCache` poderia ainda contar com outras *threads* auxiliares, responsáveis por buscar "em paralelo", nas caches remotas, um item não encontrado localmente. Essa abordagem substituiria a busca seqüencial realiza atualmente com base na lista de endereços das caches remotas. Na ocasião da implementação da `DistributedCache`, entretanto, achamos que, além da abordagem seqüencial possuir implementação mais simples, a abordagem paralela poderia gerar uma sobrecarga muito grande no sistema. Isso porque, toda vez que que uma consulta é feita a uma cache remota, esta cache remota precisa tratar a solicitação recebida, gastando tempo de processamento que poderia estar sendo útil para atender pedidos das suas *threads* locais.

Não podemos deixar de registrar, no entanto, que todos os experimentos realizados no capítulo 3 com a arquitetura distribuída do driver, registraram uma baixa taxa de

utilização da CPU nas máquinas que executam o dCache. Assim, poderia ser uma futura linha de investigação, a implementação da cache distribuída contando com várias *threads* auxiliares para consulta às caches remotas concorrentemente.

A `DistributedCache` é mostrada na figura 2.13. Na mesma figura, também podemos observar a classe `CacheMessage`, que implementa as mensagens trocadas pelas caches para realizar o compartilhamento de itens. O campo `cacheKey` de `CacheMessage` contém a chave de busca do item que está sendo procurado remotamente. Se uma cache remota contém o item solicitado, retorna uma mensagem onde o campo `cacheItem` contém o item solicitado. Note que, para trafegar na rede, a classe `CacheMessage` implementa a interface `Serializable`, e os seus campos também precisam implementá-la.

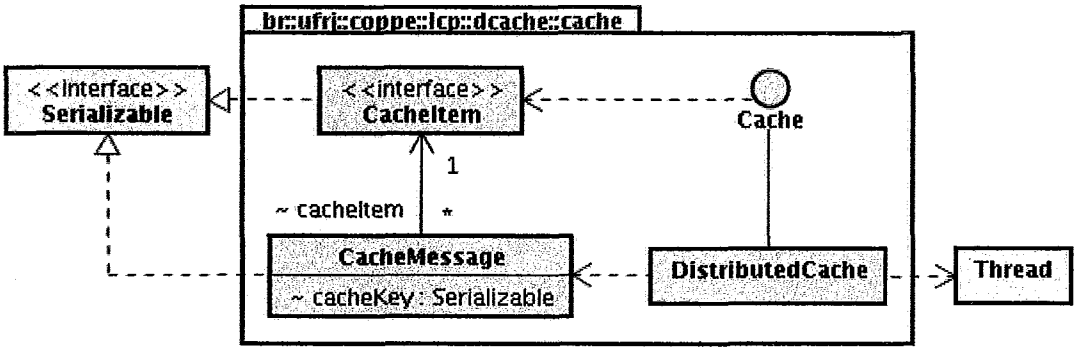


Figura 2.13: Classes importantes para a implementação da cache distribuída.

Essa exigência faz com que, na arquitetura distribuída do driver, todos os itens que podem ser armazenados em cache (ou seja, classes que implementam `CacheItem`: consultas SQL, relações e tuplas - ver figura 2.9), precisem implementar `Serializable`, bem como todo o grafo de objetos que podem ser alcançados a partir da navegação pelos campos das classes que implementam esses itens. Alguns cuidados precisaram ser tomados para evitar que todas as classes do driver acabassem, teoricamente, precisando ser serializáveis, o que seria inviável. Por exemplo, na implementação da classe `KeysRelation`, a referência para cache de tuplas precisou ser ignorada pelo processo de serialização e, através da implementação dos métodos `readExternal` e `writeExternal` (da interface `java.io.Externalizable`), reconstituímos essa referência após a de-serialização [59].

Na implementação da arquitetura distribuída do driver dCache, utilizamos a classe `DistributedCache` como cache para o armazenamento de relações e para o ar-

mazenamento de tuplas. A cache de comandos SQL, entretanto, continuou sendo implementada pela classe `LocalCache`, assim como na arquitetura centralizada (seção 2.2). A razão para esta escolha é a expectativa de que a cache local seja capaz de proporcionar uma taxa de acertos bastante elevada, tornando raras as oportunidades em que o acesso remoto seria realizado[16]. Além disso, também esperamos que o processo de *parsing* de um comando SQL seja muito rápido. Assim, não seria compensador trocá-lo por um acessos à caches remotas, que envolvem uso da rede e provavelmente seriam mais lentos do que o processo de *parsing*[17].

2.4.4 Limitações da implementação distribuída

A implementação da arquitetura distribuída contém as mesmas limitações da implementação da arquitetura centralizada (seção 2.2.3). Adicionalmente, ainda faz a exigência de que o usuário determine a priori o tamanho dos *buffers* de comunicação que serão utilizados pelas caches distribuídas. Justificaremos essa exigência nesta seção.

Vimos, na seção 2.4.3, que a implementação das caches distribuídas utilizadas pela arquitetura distribuída realiza troca de mensagens através de *sockets* UDP. Vimos também, que as mensagens trocadas pelas caches carregam, entre outros dados, o item obtido graças ao acesso à uma cache remota. Este item pode ser uma relação (no caso da cache de relações) ou uma tupla (no caso da cache de tuplas). O tamanho (em *bytes*) de uma relação varia, dentre outros fatores, com a quantidade de tuplas que ela contém. O tamanho de uma tupla, por sua vez, varia com um número de colunas que ela contém. O tamanho de uma coluna varia de acordo com o tipo de dado que ela armazena. E sobretudo, as relações e tuplas geradas dependem das consultas que são submetidas ao driver. Assim, uma mensagem trocada pelas caches distribuídas certamente possuiria tamanho variável.

Entretanto, a troca de mensagens através de *sockets* UDP exige que programador aloque explicitamente um *buffer* de tamanho pré-definido para o recebimento e envio dos datagramas. O tamanho do *buffer* utilizado para o envio de mensagens poderia até ser calculado dinamicamente de acordo com o tamanho da próxima mensagem a ser enviada.

[16] O experimentos realizados (capítulo 3) mostram que essa premissa é verdadeira para o benchmark TPC-W, onde a taxa de acertos à cache local de comandos SQL é sempre maior que 90%.

[17] O experimentos realizados (capítulo 3) mostram que essa premissa é falsa para o benchmark TPC-W. Nos experimentos realizados, o tempo médio de *parsing* de um comando SQL é de 7 ms, enquanto o tempo médio máximo de acesso à cache distribuída é de 6 ms (tempo de acesso à cache de relações, utilizando-se 3 servidores de aplicação no experimento).

Porém, o destinatário não sabe o tamanho da próxima mensagem que receberá e, por isso, precisa alocar um *buffer* grande o suficiente para que a mensagem recebida não seja truncada.

Atualmente, o tamanho dos *buffers* utilizados pelas caches do driver precisa ser definido pelo usuário através do arquivo de configuração do driver dCache (ver apêndice B). E para determinar o tamanho a ser especificado no arquivo de configuração, a única alternativa é a realização de testes. Em outras palavras, antes de utilizar o driver em ambiente de produção, o usuário precisa executar sua aplicação em ambiente de teste e coletar relatórios gerados pelo driver (ver apêndice A). Nesses relatórios, o driver informa o tamanho máximo de mensagem gerada por cada cache durante a execução. Com base neste número, o usuário então pode estimar um tamanho suficientemente grande de *buffer* a ser utilizado nas caches.

Capítulo 3

Avaliação Experimental

A avaliação experimental do driver dCache foi realizada em um cluster de comércio eletrônico dotado de, basicamente, um servidor Web, alguns servidores de aplicação e um servidor de banco de dados. Os servidores de aplicação executam uma aplicação de comércio eletrônico escrita em Java, que acessa o servidor de banco de dados utilizando o driver dCache por intermédio da API JDBC. A aplicação de comércio eletrônico utilizada foi um benchmark especificado e implementado especificamente para a avaliação de aplicações de processamento de transações *on-line*.

Na próxima seção, descreveremos em maiores detalhes o benchmark utilizado. Nas seções subseqüentes, descreveremos o ambiente experimental utilizado para a execução do benchmark, definiremos os experimentos que foram realizados no ambiente experimental descrito e, por fim, analisaremos os resultados obtidos.

3.1 Benchmark utilizado

Para a avaliação experimental do driver dCache foi utilizado o benchmark *TPC-W* [41] especificado pelo grupo *Transaction Processing Performance Council* (TPC) [60]. O TPC é uma organização sem fins lucrativos criada para, além de especificar benchmarks aplicados a sistemas de bancos de dados e processamento de transações, disponibilizar para a indústria resultados quantitativos sobre o desempenho desses sistemas. No escopo do TPC, o termo *transação* refere-se, de forma abrangente, a quaisquer operações realizadas por clientes de sistemas de computação, conectados a sistemas de banco de dados. Seriam exemplos de transações tratadas pelo TPC: acessos a um sistema de controle de estoque, a um sistema de reserva de passagens aéreas ou a um sistema bancário.

O TPC-W é um benchmark de transações Web. De forma pragmática, podemos dizer que ele especifica um servidor de comércio eletrônico (mais precisamente uma livraria virtual) acessada por clientes e seus administradores, os primeiros, interessados em pesquisar e comprar livros via Internet, e os demais, responsáveis por modificar dados sobre o estoque disponível. O TPC-W especifica um banco de dados onde são armazenadas informações sobre os livros disponíveis, sobre os clientes e seus pedidos. Este banco é acessado pelos clientes da livraria através de requisições HTTP, feitas ao servidor de comércio eletrônico por um emulador de navegadores Web (*browsers*).

São definidos pelo benchmark diferentes tipos de interações Web, que representam as "ações" que um cliente pode realizar durante sua visita à livraria virtual. São exemplos de interações Web: acessar a página de entrada da livraria, pesquisar por livros usando alguma chave de busca, solicitar informações detalhadas sobre um determinado exemplar, cadastrar-se como cliente da livraria, adicionar livro no carrinho de compras ou efetuar pagamento.

O benchmark TPC-W permite a configuração de alguns parâmetros de funcionamento, dentre eles, o número de navegadores emulados, o tamanho da base de dados acessada e o perfil de navegação. O número de navegadores emulados determina o carga de trabalho à qual estará submetido o servidor de comércio eletrônico em teste. O tamanho da base de dados acessada e o número de navegadores emulados variam obedecendo um fator de escala especificado pelo benchmark com o objetivo de manter constante a relação entre essas duas grandezas. Os perfis de navegação definem o comportamento dos clientes emulados. A escolha do perfil a ser empregado na execução do benchmark determina a proporção entre os tipos de interações Web que serão realizadas e, indiretamente, influi na quantidade de operações de leitura e escrita efetuadas no banco de dados.

São definidos três perfis de navegação: *Browsing Mix* (BM), *Shopping Mix* (SM) e *Ordering Mix* (OM). O perfil BM representa o comportamento de navegação daqueles clientes que preferem acessar os *links* presentes nas páginas HTML ao invés de realizar pesquisas através da página de busca disponibilizada pela livraria virtual, e efetuam poucas compras. O perfil SM representa o comportamento dos clientes que costumam utilizar a página de busca para encontrar os livros que desejam adquirir e efetuam mais compras que o perfil anterior. O perfil OM representa aqueles clientes que acessam a livraria para encontrar um livro específico e comprá-lo.

A especificação particiona o tempo total de execução do benchmark TPC-W em três fases: a fase de aquecimento (*ramp-up*), o intervalo de medição e a finalização (*ramp-down*). A fase de aquecimento precisa durar, no mínimo 10 minutos, e deve se estender até

que o sistema entre em estado de regime. A fase de medição começa após o aquecimento e deve se estender por, no mínimo 30 minutos. Após o intervalo de medição o sistema deve ser ainda mantido em funcionamento por mais, pelo menos 5 minutos, período que chamamos de "finalização".

O dado de saída do benchmark é a vazão do servidor de comércio eletrônico, medida em *Web Interactions Per Minute* (WIPM), onde uma interação Web consiste em um ciclo completo (pedido e resposta) de comunicação entre o navegador Web emulado e o servidor de comércio eletrônico. Se durante o intervalo de medição o servidor de comércio eletrônico realiza N interações Web e Ri é o tempo de resposta para a interação i, então,

$$WIPM = \frac{N}{\sum_{i=1}^{N} Ri}$$

3.1.1 Implementação do benchmark

O projeto PHARM [61] da Universidade Wisconsin-Madison desenvolveu, em 1999, uma implementação não oficial da versão 1.0 da especificação TPC-W. Essa implementação foi escrita completamente em Java (mais precisamente, utilizando Java Servlets [62]) e preparada para o SGBD IBM DB2. Posteriormente, essa implementação foi adaptada para o banco MySQL e distribuída pelo consórcio ObjectWeb [63], que promove o desenvolvimento e teste de *middleware* de código aberto. Para a avaliação do driver dCache realizada por [40], foi utilizada uma implementação derivada desta última, onde foram realizadas as seguintes modificações:

- Implementação de uma solução baseada somente em "dados de seção" para o carrinho de compras da livraria. A implementação original armazenava as informações do carrinho de compras na base de dados, acarretando uma ocupação desnecessária da base de dados, quando na verdade, a especificação TPC-W [64] não faz essa exigência.
- Substituição da função de comparação fonética *soundex* pelo operador "*match against*" na implementação da busca por palavras chave no banco de dados. A função *soundex* não produz resultados exatos e o operador *LIKE*, normalmente utilizado com esta finalidade em SQL, acarretou problemas de desempenho no MySQL quando a tabela buscada possuía muitos registros.
- Correção na implementação da cláusula 2.14.3.1 da especificação TPC-W, responsável pela interação onde são solicitadas informações mais detalhadas sobre

determinado exemplar da livraria: a implementação original trazia, a partir da leitura do banco de dados, mais informações do que as realmente necessárias.

A avaliação do dCache realizada pelo presente trabalho utiliza a implementação dotada das modificações listadas a cima, acrescida das seguintes modificações:

- Em [40] foi descoberto que, apesar de não ser indicado na documentação, a implementação original do benchmark precisava ser manualmente ajustada para realização de medições cujo fator de escala não fosse o de 10.000 itens na base de dados. Adicionamos então o número de itens disponíveis na base de dados ao arquivo de configuração do benchmark. Assim, o benchmark pode ser corretamente parametrizado sem necessidade de edição do seu código-fonte.
- Adaptação dos comandos SQL utilizados pelo benchmark para uso do SGBD Postgres. Foram necessárias três adaptações: substituição do operador "*match against*" pelo operador *LIKE*, utilização de caracteres ' em torno dos argumentos passados para a função *INTERVAL* e substituição do tipo *double*, usado na criação das tabelas, pelo tipo *double precision*.
- Utilização de *sequences* no lugar de blocos Java *synchronized*.
- Utilização de consulta montada dinamicamente substituindo sintaxe não entendida pelo driver.

As duas últimas modificações exigem uma explicação mais detalhada. Vamos falar primeiramente do uso das *sequences*: Como, antes do presente trabalho, o driver dCache podia apenas ser executado por um único servidor de aplicação (pois só existia a versão centralizada da arquitetura), a implementação do benchmark realizava a inserção de novos registros na base de dados dentro de blocos de código Java envolvidos pela cláusula *synchronized*. Dentro destes blocos, o benchmark executava um comando SELECT para obter a cardinalidade da tabela que receberia o novo registro e, em seguida, executava um comando INSERT para inserir o novo registro, cuja chave primária era computada incrementando-se em uma unidade a cardinalidade obtida.

Dentro de uma única JVM, a cláusula *synchronized* é capaz de evitar que múltiplas *threads* realizem inserções concorrentes na mesma tabela e, assim, não existe o risco de tentativas de violações de chave primária[1]. Como é utilizado mais de um servidor

[1]Uma tentativa de violação de chave primária poderia ocorrer se, na ausência da cláusula *synchronized*, uma *thread* $t1$ obtém a cardinalidade de uma tabela e a incrementa em uma unidade obtendo o valor x. Uma *thread* concorrente $t2$, interessada em realizar uma inserção na mesma tabela, obtém a mesma cardinalidade e após incrementá-la em uma unidade, também obtém o valor x. Neste momento, $t1$ volta a executar e realiza, com sucesso, a inserção de um novo registro cuja chave primária é x. Quando $t2$ voltar a

de aplicação na avaliação da versão distribuída do dCache, precisamos evitar as tentativas de violação também entre máquinas virtuais distintas. Optamos pelo emprego do recurso de *sequences* fornecido pelo SGBD Postgres. Usando *sequences* para fornecer as chaves primárias das tabelas, é possível incrementar um contador global e obter o valor incrementado atomicamente, evitando as tentativas de violação de integridade.

Vamos agora à explicação da última modificação: Na seção 2.2, vimos que, dentre as limitações da implementação do driver dCache, encontra-se a restrição dos comandos SQL entendidos pelo driver a um subconjunto da sintaxe definida pelo padrão SQL-92. Vimos também que, dentre as restrições, estão a impossibilidade de uso de consultas encadeadas e impossibilidade do uso do operador NOT. A interação responsável pela atualização de informações cadastrais de um determinado exemplar da livraria faz uso de uma consulta que utiliza essas duas sintaxes não entendidas pelo driver dCache.

Na implementação do benchmark utilizada pelo presente trabalho, desmembramos as consultas encadeadas em consultas simples (montadas em tempo de execução) e eliminamos o uso do operador NOT filtrando os resultados obtidos com código Java adicional. Como conseqüência, a implementação obtida demandará um pouco mais de processamento por parte do servidor de aplicação (responsável por montar as consultas em tempo de execução) e causará uma ocupação diferente nas caches do driver (devido ao uso das consultas montadas dinamicamente). Entretanto, é importante salientar que a interação que utiliza essa consulta representa apenas 0.09% das interações dos perfis BM e SM e apenas 0.11% das interações do perfil OM. Ou seja, é esperado que o impacto provocado por esta mudança tenha efeito limitado quando se observa o baixo percentual de uso da consulta modificada perante as demais.

A implementação utilizada pelo presente trabalho ainda se desvia de uma implementação oficial do TPC-W, pelo menos, nos seguintes aspectos:

- Não existe comunicação com um emulador de *gateway* de pagamento, que serviria para simular a operação de solicitação de confirmação de pagamento com a administradora de cartão de crédito, no momento da confirmação da compra.
- Não são feitas conexões via *Secure Sockets Layer* (SSL) ou *Transport Layer Security* (TLS) para as transações seguras.
- O programa que faz a criação e população da base de dados não gera os nomes de autores e títulos dos livros seguindo a especificação. Uma ferramenta chamada "*wgen*" fornecida pelo próprio TPC foi utilizada para esse propósito.

executar, tentará inserir outro registro também com chave primária x, caracterizando a tentativa de violação de integridade.

Esses desvios da especificação oficial podem ter os seguintes impactos nos resultados obtidos:

- Em conexões com criptografia para as transações seguras as transferências ocupam mais a CPU do servidor Web, mas não do servidor de aplicação ou do servidor de banco de dados, onde a cache tem influência direta. Haveria um peso maior para a carga do servidor Web, talvez diminuindo um pouco a importância da cache na vazão total. Entretanto, essas conexões deveriam ser feitas somente para algumas interações específicas, interações as quais totalizam apenas 1,99% das interações do perfil BM, 5,21% das interações do perfil SM e 23,38% das interações do perfil OM.
- O acesso ao emulador de *gateway* de pagamento é feito pelo servidor de aplicação e também envolve criptografia. A criptografia é assimétrica e a chave privada usada deve ter no mínimo 1.024 bits. Haveria uma carga maior de processamento justamente onde a cache é manipulada - no servidor de aplicação, potencialmente prejudicando o uso da cache. Entretanto, o emulador de *gateway* de pagamento é acessado apenas durante interações que correspondem a 0,69% das interações do perfil BM, 1,20% das interações do perfil SM e 10,18% das interações no perfil OM.

3.2 Instrumentação

Dissemos, durante a descrição da arquitetura do driver dCache (capítulo 2), que as aplicações cliente se comunicam com o driver através da API JDBC. Não é diferente com o benchmark TPC-W. Por este motivo, focamos a instrumentação do driver na invocação aos métodos desta API. Complementarmente, a instrumentação também deu foco especial ao comportamento das caches, por se tratarem de componentes fundamentais na determinação do desempenho do driver.

De todos os métodos definidos pela API JDBC, apenas 25 são utilizados pelo benchmark TPC-W. Destes 25, apenas 12 são relevantes para a avaliação de desempenho do driver dCache[2]. Listamos abaixo, agrupados por suas respectivas classes, os 25 métodos da API JDBC utilizados pelo benchmark TPC-W e ressaltamos em negrito os 12 relevantes para avaliação de desempenho do driver dCache:

[2]Dizemos que apenas estes 12 métodos são relevantes para a avaliação de desempenho pois, após algumas medições, verificamos que o tempo total de execução de cada um dos demais 13 métodos é insignificante comparado ao tempo de execução dos métodos escolhidos.

- Classe `Connection`:
  - `Connection#isClosed()`
  - `Connection#close()`
  - `Connection#setAutoCommit(boolean)`
  - **`Connection#commit()`**
  - `Connection#rollback()`
  - **`Connection#prepareStatement(String)`**
- Classe `PreparedStatement`:
  - **`PreparedStatement#executeQuery()`**
  - **`PreparedStatement#executeUpdate()`**
  - `PreparedStatement#setInt(int,int)`
  - `PreparedStatement#setString(int, String)`
  - `PreparedStatement#setDouble(int, double)`
  - `PreparedStatement#setDate(int, Date)`
  - `PreparedStatement#setTimestamp(int, Timestamp)`
  - `PreparedStatement#setLong(int, long)`
  - `PreparedStatement#close()`
- Classe `ResultSet`:
  - **`ResultSet#next()`**
  - `ResultSet#close()`
  - **`ResultSet#getInt(int)`**
  - **`ResultSet#getInt(String)`**
  - **`ResultSet#getString(int)`**
  - **`ResultSet#getString(String)`**
  - **`ResultSet#getDouble(int)`**
  - **`ResultSet#getDouble(String)`**
  - **`ResultSet#getDate(String)`**
  - **`ResultSet#getTimestamp(String)`**

Na implementação do driver dCache, a execução dos *getters* da classe `ResultSet` é sempre delegada para um método privado chamado `getValue(int)`. Este é caso dos últimos 8 métodos em negrito da classe `ResultSet`. Por isso, apesar destes 8 métodos serem de fato invocados pelo benchmark TPC-W, deste ponto em diante não vamos nos referir diretamente ao tempo de execução de cada um destes métodos, e sim, ao tempo de execução do método `getValue(int)`, responsável pela execução de qualquer um dos 8.

A instrumentação do driver armazena em um relatório, para cada um destes métodos, o seu número total de execuções, o tempo total de execução e os tempos mínimo, máximo e médio de execução. Os tempos de execução de atividades relevantes desempenhadas internamente por cada um destes métodos também são armazenados no relatório, assim como, os dados sobre o funcionamento das caches e qualquer informação complementar importante para o entendimento do comportamento do driver.

Cada instância do driver mantém o seu próprio relatório em memória e adiciona dados dinamicamente, durante a execução da aplicação cliente (no caso da avaliação do driver, o benchmark TPC-W). Periodicamente, o servidor dCache, envia uma requisição a cada instância do driver, solicitando o seu relatório. O servidor dCache recebe os relatórios das instâncias e armazena-os no seu sistema de arquivo local, disponibilizando-os para a nossa análise futura[3].

3.3 Ambiente experimental

Os experimentos utilizados para a avaliação do driver dCache foram realizados no cluster do Laboratório de Computação Paralela da COPPE-UFRJ (LCP) no período de outubro de 2007 até junho de 2008. Durante o processo de implementação e testes das duas arquiteturas (centralizada e distribuída) apresentadas na capítulo 2, foram geradas 51 versões diferentes do driver dCache. Para os testes de corretude e avaliação de desempenho dessas versões foram utilizadas aproximadamente 1250 horas de processamento no cluster do LCP, divididas em, aproximadamente, 680 execuções do benchmark TPC-W, que geraram mais de 6 *gigabytes* de dados.

Apresentaremos nas seções subseqüentes os resultados obtidos por um subconjunto dos experimentos realizados, correspondente às medições relacionadas às duas versões finais do driver. A versão final correspondente à arquitetura centralizada chama-se 1.12.0.local. A versão final correspondente à arquitetura distribuída chama-se 2.16.0. Para cada uma destas duas versões, foram gerados dois *builds* (duas "compilações") distintos, um com código adicional de instrumentação e outro sem código de instrumentação.

A compilação com código de instrumentação seria utilizada exclusivamente para

[3]Para uma discussão mais detalhada sobre a implementação da instrumentação no driver, em especial, sobre a implementação dos relatórios e do mecanismo de solicitação remota de relatórios, sugerimos a leitura do apêndice A. O apêndice B ensina como configurar a geração de relatórios no driver dCache e como acompanhar o funcionamento do driver em tempo de execução, através da solicitação de relatórios às instâncias via ferramenta de linha de comando.

obter dados sobre o funcionamento interno do driver ao longo dos experimentos, enquanto a compilação sem instrumentação seria utilizada quando desejássemos obter dados de desempenho do driver sem nos preocuparmos com o seu funcionamento interno (estilo "caixa-preta"). A existência de duas compilações distintas permitiria que os dados obtidos pelas medições de desempenho estilo "caixa-preta" não fossem contaminados com o *overhead* adicional de instrumentação. Entretanto, após algumas dezenas de medições, notamos que este *overhead* é desprezível e, então, passamos a utilizar a compilação com instrumentação indiscriminadamente. Os dados de desempenho apresentados no restante do texto terão sido obtidos, ora pela compilação com instrumentação, ora pela compilação sem instrumentação, mas não faremos distinção entre os dois casos.

Os experimentos realizados para avaliação do driver utilizaram de quatro a oito máquinas alocadas da seguinte forma:

- Uma máquina é reservada para executar exclusivamente o *remote browser emulator* (RBE) (emulador de navegadores remoto). O RBE emula navegadores de clientes hipotéticos acessando a livraria virtual do benchmark TPC-W.
- Outra máquina foi reservada para exercer o papel de servidor de banco de dados. Nesta máquina foi instalado o SGBD *PostgreSQL* [65], versão 8.2.5. Este SGBD gerencia um único banco de dados, onde são armazenados os dados da livraria virtual. Esta máquina possui dois discos rígidos, cada um ligado em uma porta IDE dedicada. Um dos discos foi reservado para armazenar, exclusivamente, o banco de dados usado nos experimentos. Este disco trabalha em 7200 RPM, possui 1023 cilindros, 256 cabeças, 63 setores e um tempo de posicionamento médio de 8.5 ms (dados do fabricante).
- De uma a três máquinas são utilizadas exclusivamente para executar instâncias do servidor Tomcat (os experimentos utilizaram a versão 6.0.16 deste servidor). Cada instância do servidor Tomcat hospeda uma instância da livraria virtual do benchmark TPC-W. Deste ponto em diante, usaremos as expressões "servidor de aplicação" ou "servidor Tomcat" para nos referirmos ao mesmo software.
- Outra máquina é dedicada a executar um servidor HTTP Apache [10], versão 2.2.6. O servidor Apache é utilizado para intermediar a comunicação entre os navegadores dos clientes hipotéticos da livraria virtual e os servidores Tomcat. Para permitir a comunicação entre o servidor Apache e os servidores Tomcat foi utilizado o módulo de comunicação *mod_jk*[4], versão 1.2.25. O protocolo utili-

[4]Antes de optarmos pelo módulo *mod_jk* para intermediar a comunicação entre Apache e Tomcats, tentamos utilizar o módulo *mod_proxy* [66] fornecido pela própria equipe de desenvolvimento do servidor

zado por este módulo é o *Apache JServ Protocol version 1.3 (AJP13)* [67]. O servidor Apache e o módulo *mod_jk* foram configurados para realizar balanceamento de carga, sempre direcionando novos clientes para os servidores Tomcat menos sobrecarregados.

- Outra máquina é dedicada a executar o monitor de desempenho *Sysusager*[5].

O servidor Apache é hospedado em uma máquina com processador Intel Pentium 4 Hyper-Threading, 2.4 GHz, 512 KB de cache, rodando Linux kernel 2.6.9, com *Symmetric Multiprocessing* (SMP) desabilitado. O servidor Postgres é hospedado em uma máquina com mesmo hardware, porém, rodando Linux kernel 2.6.23, também com SMP desabilitado. As demais aplicações rodam em máquinas com processador Intel Pentium 4 Hyperthreading, 2.8 GHz, 1 MB de cache rodando Linux kernel 2.6.9 com SMP habilitado. Todas as máquinas utilizadas possuem 1 GB de memória RAM e são conectadas através de 2 switches gigabit ethernet conforme indicado pela figura 3.1.

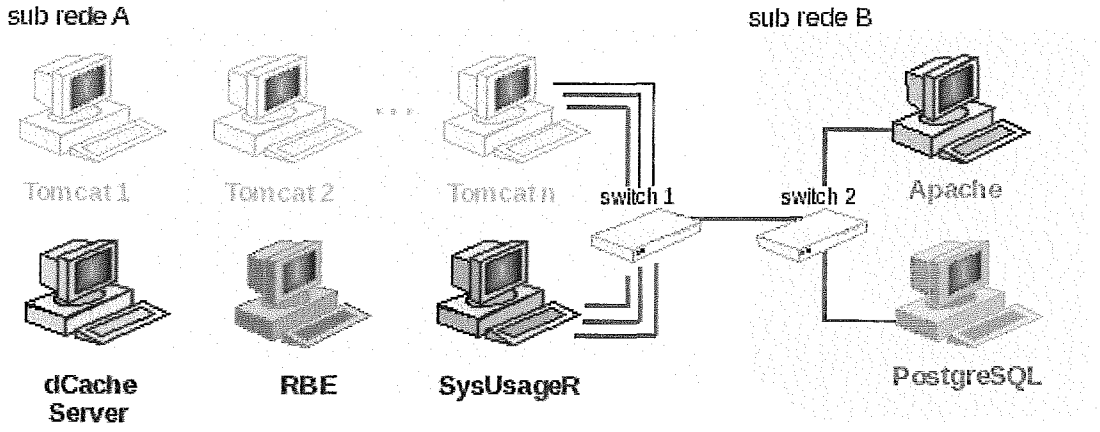


Figura 3.1: Alocação das máquinas do cluster para os experimentos com o dCache.

Os navegadores emulados pelo RBE enviam requisições HTTP para o servidor Apache solicitando páginas da livraria virtual. O servidor Apache seleciona qual servidor Tomcat será o responsável por processar as requisições de cada cliente e repassa as requisições destes clientes para o seus respectivos servidores Tomcat. Os servidores Tomcat, por sua vez, se comunicam com o SGBD através de um driver JDBC. Dependendo

Apache. No entanto, este módulo não foi capaz de atender a experimentos utilizando 50 ou mais navegadores emulados. Nestes experimentos, após 5 minutos de execução, os servidores Apache e Tomcat lançavam exceções reportando falha de comunicação.

[5]O monitor de desempenho Sysusager é uma aplicação desenvolvida neste trabalho para monitorar o uso de recursos nas diversas máquinas utilizadas pelos experimentos. Para maiores detalhes sobre este monitor, consulte o apêndice E.

do objetivo do experimento que estiver sendo realizado, este driver JDBC será ou o driver dCache, ou o driver *PostgreSQL* [68] (versão 8.2-507.jdbc3). Quando o driver dCache for utilizado, uma máquina adicional será dedicada a executar o servidor do driver dCache. A figura 3.2 mostra a conexão lógica entre as máquinas (o servidor do driver dCache e o servidor Sysusager não são mostrados nesta figura por simplificação).

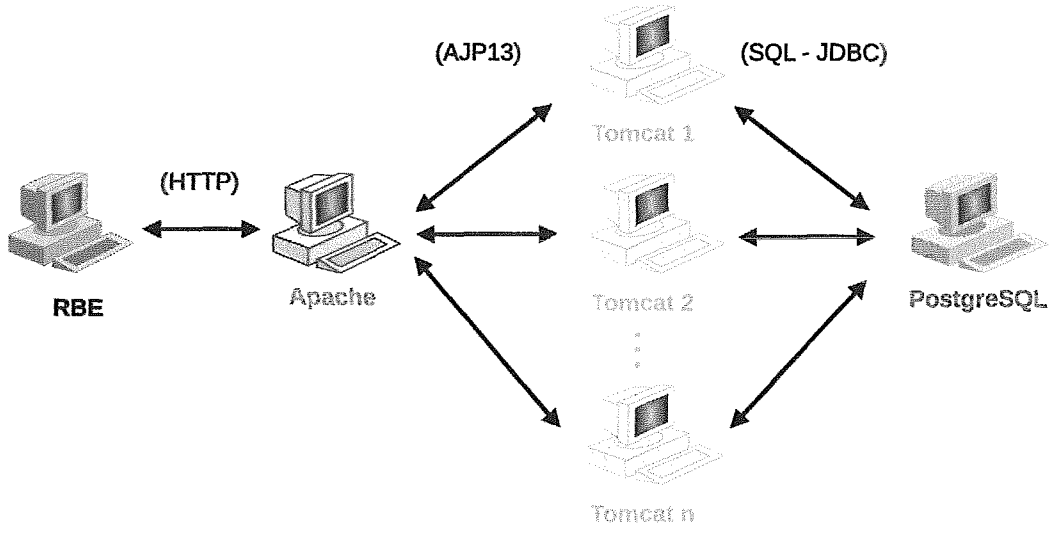


Figura 3.2: Alocação das máquinas do cluster para os experimentos com o dCache

Para executar os servidores Tomcat, o RBE, o servidor dCache e o monitor de desempenho, foi utilizado o *Java Runtime Environment* (JRE) [69] versão 1.5.0_15. Para esta versão do JRE, o tamanho máximo padrão da *heap* das JVMs equivale a um quarto da memória física da máquina. Dado que as máquinas utilizadas nos experimentos do driver dCache possuem 1 GB de RAM, essa fração corresponde a 256 MB. Em experimentos com grande quantidade de navegadores emulados, os servidores Tomcat excederam o tamanho máximo default das *heaps* das suas JVMs. Assim, optamos por fixar o tamanho máximo da *heap* utilizada por estes servidores em 512 MB. Esse limite superior de memória atendeu a todos os experimentos realizados. Fora o caso dos servidores Tomcat, foi utilizado como tamanho máximo da *heap* da JVM a configuração default.

As máquinas utilizadas como servidor de banco de dados e com o servidor Apache são fixas, ou seja, são sempre as mesmas em todos os experimentos. As demais, embora possuam sempre as configurações descritas, são selecionadas a critério do sistema de gerência do cluster do LCP e sofrem variações a cada nova execução do benchmark. Para preparar o ambiente experimental e disparar uma nova execução do benchmark, é executado, a partir do *frontend* do cluster, um script de linha de comando (*shell script*)

para o interpretador de comandos do Linux. Após ler, a partir de uma série de arquivos de configuração, os parâmetros de execução do experimento (tamanho das caches, número de servidores Tomcats, duração do intervalo de medição ...) o *script* executa as seguintes atividades:

1. Alocar uma quantidade suficiente de máquinas no cluster e definir qual o papel (servidor de aplicação, servidor dCache...) de cada uma delas.
2. Instalar os sistemas apropriados (Tomcat, servidor dCache...) nas máquinas alocadas de acordo com o papel definido.
3. Gerar arquivos de configuração para os sistemas utilizados e instalá-los nas máquinas que hospedam os respectivos sistemas.
4. Preparar a base de dados de acordo a configuração do benchmark que será utilizada.
5. Iniciar os sistemas.
6. Monitorar a execução (para abortar a execução do benchmark assim que seja detectada uma eventual falha, evitando assim perda de tempo de utilização do cluster com execuções do benchmark cujos resultados já estão comprometidos).
7. Coletar os dados gerados (arquivos de *log* e arquivos de saída de instrumentação) pelos sistemas em execução nas diversas máquinas e arquivá-los para análise posterior.
8. Remover os sistemas que foram instalados no cluster mantendo o ambiente computacional "limpo" para receber outros experimentos dCache ou quaisquer outros *jobs* de outros usuários do cluster.

Veremos que, durante os experimentos realizados para avaliar o dCache, quatro tamanhos diferentes de base de dados são utilizados. *Backups* destas quatro bases são armazenadas no sistema de arquivo do servidor de banco de dados. Antes de realizar uma nova execução do benchmark, o script elimina do SGBD o banco de dados do TPC-W, cria um novo banco e o popula restaurando o *backup* relativo à configuração selecionada para o experimento a ser realizado. Mesmo que fosse sempre utilizada a mesma configuração, ainda assim a base de dados precisaria ser reiniciada pois após algumas execuções consecutivas do benchmark, os resultados obtidos por experimentos equivalentes começam a gerar resultados discrepantes devido às modificações realizadas pelas operações de escrita na base de dados.

É importante também registrar alguns cuidados de otimização que foram tomados antes de realização das medições:

- No arquivo de configuração do servidor Apache (*httpd.conf*), foram utilizadas

as diretivas `ServerLimit` e `MaxClients`. Nas duas, foi utilizado como argumento, 1,5 vezes a quantidade de navegadores emulados que será utilizada no experimento[6].

- Nos arquivos de configuração dos Tomcats (*server.xml*) foram utilizados os atributos `maxThreads`, `minSpareThreads` e `maxSpareThreads`. O primeiro, recebe como valor 1,5 vezes a quantidade máxima de navegadores emulados que será usada no benchmark[6]. Os últimos recebem como valor a quantidade de navegadores emulados que será usada no experimento divida pelo número de Tomcats que serão utilizados no experimento.
- Assim como em [40] e [61], a base de dados conta com vários índices criados para otimizar o acesso a algumas tabelas [61].
- Assim como em [40] e [61], as conexões para acesso ao banco são criadas sob demanda pelo benchmark e armazenadas em um *pool* de conexões para reutilização futura.
- O emulador de browsers remoto foi configurado para não realizar as requisições das imagens das páginas Web acessadas (opção *-GETIM false*).

3.4 Experimentos realizados

Os experimentos que foram realizados através do benchmark TPC-W no ambiente experimental descrito procuraram responder a quatro questões principais:

1. Qual o ganho proporcionado pela versão centralizada do driver dCache no desempenho do sistema em teste quando é utilizado apenas um servidor de aplicação?
2. Qual a sobrecarga imposta pelo uso da versão distribuída do driver dCache ao sistema em teste quando é utilizado apenas um servidor de aplicação?
3. Qual o efeito do uso de mais de um servidor de aplicação no desempenho do sistema em teste quando o driver dCache não é utilizado?
4. Qual o efeito do uso da versão distribuída do driver dCache no desempenho do sistema em teste quando são utilizados mais de um servidor de aplicação?

A primeira pergunta já havia sido respondida por [40]. No entanto, como a arquitetura centralizada apresentada no presente trabalho modificou a arquitetura proposta em [40], consideramos interessante repetir os experimentos realizados por [40] para verificar

[6]Os 50% de *threads* adicionais são usados para evitar erros que experimentamos durante as execuções do benchmark.

o impacto das modificações realizadas nos resultados obtidos. Além disso, independentemente da avaliação do impacto dessas modificações, usaremos os novos resultados como parâmetro para comparar o desempenho do sistema em teste usando as versões centralizada e distribuída do driver dCache. Os experimentos realizados para tentar responder a esta primeira pergunta e os resultados obtidos são descritos na seção 3.4.1.

Embora não faça sentido utilizar a versão distribuída do driver em um sistema onde é empregado apenas um servidor de aplicação, responderemos à segunda questão pois a sua resposta será um indicativo de qual a sobrecarga que a versão distribuída do driver dCache impõe ao sistema em teste. Em outras palavras, estamos esperando que o acréscimo de código de sincronização distribuída e o uso de estruturas de dados distribuídas na versão distribuída do driver tenha adicionado um *overhead* ao funcionamento do driver. Gostaríamos então, de ter uma noção quantitativa deste *overhead.* Tentaremos obtê-la respondendo a segunda pergunta na seção 3.4.2.

Em [40] apenas um servidor de aplicação foi utilizado. De fato, não esperávamos que mais de um servidor de aplicação fosse utilizado pois a arquitetura do driver avaliada por [40] foi preparada para suportar apenas um servidor de aplicação. Ainda não sabemos qual o comportamento do sistema em teste quando mais de um servidor de aplicação é utilizado, nem mesmo, sem empregar o driver dCache. Responderemos a terceira pergunta para saber como o sistema em teste se comporta com o aumento do número de servidores de aplicação. A resposta a essa pergunta (mostrada na seção 3.4.3) mostrará que a replicação de servidores de aplicação impõe uma sobrecarga ao sistema em teste, independentemente, da utilização ou não do driver dCache.

Se por um lado esperamos que a versão distribuída do driver dCache imponha *overhead* ao sistema em teste graças ao acréscimo de código de sincronização distribuída e o uso de estruturas de dados/algoritmos distribuídos, por outro lado, esperamos que o compartilhamento dos dados em cache por diversas instâncias do driver dCache e a distribuição da carga imposta pelos navegadores por diversos Tomcats se reverta em melhoria de desempenho. Para tentar avaliar essa relação custo/benefício, responderemos então à quarta e última pergunta na seção 3.4.4.

Nas seções seguintes, usaremos os seguintes códigos para simplificar a descrição das configurações utilizadas nos experimentos: *tpcw-100K-300*, *tpcw-1M-50*, *tpcw-10K-25* e *tpcw-10K-100*. Em cada código, o primeiro número indica a quantidade de itens (livros) no banco de dados no início do experimento. O último número indica a quantidade de navegadores emulados que serão utilizados no experimento. O primeiro código, por exemplo, indica que será utilizada uma base de dados de 100 mil itens, acessada por 300

navegadores. Utilizando a especificação do TPC-W [64], é possível determinar qual a cardinalidade das demais tabelas existentes na base de dados.

3.4.1 Comparando as versões centralizadas

Em [40] foram realizadas duas séries de experimentos para a avaliação da implementação da versão centralizada da arquitetura do driver dCache. Na primeira série, foi utilizada a configuração tpcw-100K-300 e na segunda, a configuração tpcw-1M-50. Em cada uma das séries, foram realizadas medições com e sem o uso do driver dCache. Quando o driver dCache não é utilizado, as medições duram 30 minutos e começam após um período de aquecimento do sistema com duração de, também, 30 minutos. Quando o driver dCache é usado na primeira série, as medições duram 30 minutos e são precedidas por um período de aquecimento de uma hora. Quando o driver dCache é usado na segunda série, as medições também duram 30 minutos, mas o período de aquecimento é de 3 horas. Em ambas as séries, foram permitidas, no máximo, 1000 entradas na cache de comandos SQL e um milhão de entradas nas caches de relações e tuplas.

O número de interações Web por minuto obtidos por [40], na primeira série, com e sem uso do driver dCache, são mostrados na figura 3.3 *a*. A partir das quantidade de interações Web por minuto mostradas no gráfico, percebemos que, com o uso do driver dCache, ocorre um ganho de desempenho de 55% para o perfil BM e de 17% para o perfil SM. Para o perfil OM, há uma redução de 14%.

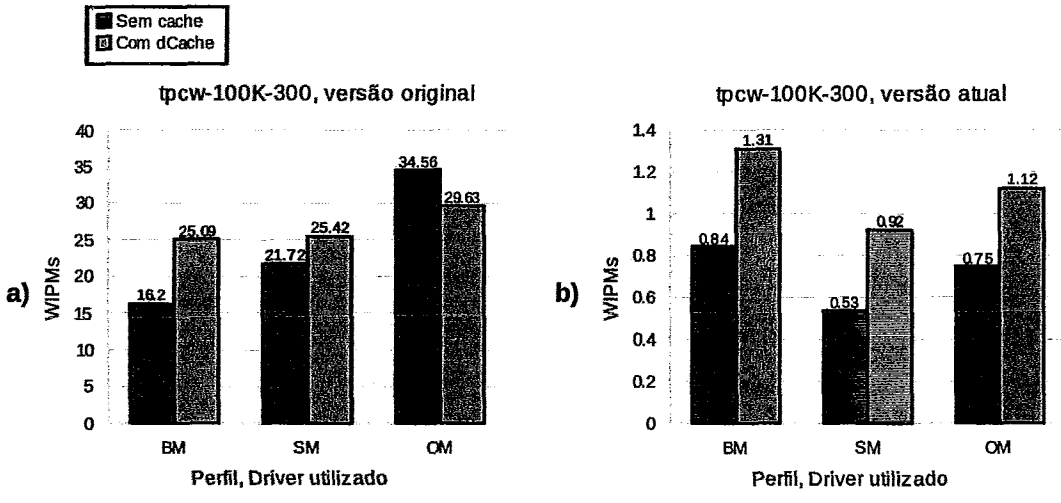


Figura 3.3: Gráfico de interações Web por minuto para a configuração tpcw-100K-300, com e sem o uso do driver dCache (apenas 1 Tomcat).

Repetimos as medições com a nova implementação da versão centralizada da arquitetura do driver e obtivemos os resultados mostrados na figura 3.3 *b*. Para uma comparação mais justa, repetimos não só as medições realizadas utilizando-se o driver dCache, como também as que não utilizam o driver dCache. Para chegarmos aos valores no topo de cada uma das barras do gráfico apresentado na figura 3.3 *b*, realizamos 5 medições, descartamos as que apresentaram o maior e o menor valor e realizamos a média aritmética simples das 3 medições restantes. Podemos observar que a quantidade absoluta de interações Web por minuto diminui em todos os perfis de navegação, independentemente do emprego ou não do driver dCache. Entretanto, o ganho de desempenho proporcionado pelo uso do driver é o mesmo para o perfil BM (55%), aumenta para 72% no perfil SM e para 50% no perfil OM.

É interessante notar que ocorrem aumentos nos ganhos de desempenho proporcionados pelo uso do driver dCache exatamente nos perfis onde ocorrem mais operações de escrita e, conseqüentemente, mais invalidações. A melhoria no desempenho nestes perfis pode indicar que a utilização, na arquitetura centralizada implementada por este trabalho, de uma cache de tuplas única para cada *schema* (ver seção 2.1.9) pode ter sido um aprimoramento eficiente no sentido de minimizar as perdas proporcionadas pelo mecanismo de invalidação utilizado em [40], onde todas as tuplas relacionadas à uma consulta compilada invalidada eram eliminadas da cache.

A figura 3.4 mostra a variação na ocupação das caches de consultas SQL, na cache de relações e de tuplas no decorrer do experimento. O gráfico *a* refere-se à ocupação da cache de consultas SQL, o *b* à cache de relações e o *c* à cache de tuplas. Em cada um dos gráficos são mostradas as variações para os três perfis de navegação utilizados. Nos três gráficos, as linhas pontilhadas verticais indicam o início e término do período de medição (ou seja, o gráfico engloba o período de aquecimento e 10 minutos adicionais após o final do período de medição). As taxas de acertos para as três caches, nos três perfis é mostrada pelo gráfico *a* apresentado na figura 3.5.

Comparando-se a variação, em função do perfil de carga empregado no benchmark, da taxa de acerto nas caches, da taxa de ocupação das caches e do ganho de desempenho obtido, parece não ser possível estabelecer uma relação direta entre nenhum destes índices. É interessante observar, por exemplo, que o perfil SM, no qual é obtido o maior ganho de desempenho (72%), é o que apresenta menor taxa de acertos na cache de relações e uma taxa de acertos mediana na cache de tuplas.

É difícil comparar os índices obtidos pelas novas medições com os obtidos por [40] por dois motivos. Primeiramente, a estrutura das caches foi alterada (conforme já

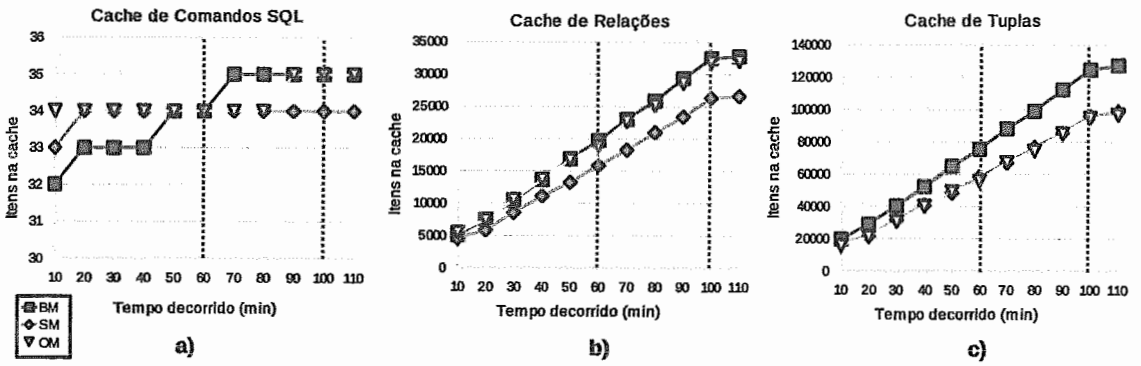


Figura 3.4: Variação na ocupação das caches com o tempo de execução do benchmark para a configuração tpcw-100K-300 e para os três perfis de navegação (apenas 1 Tomcat).

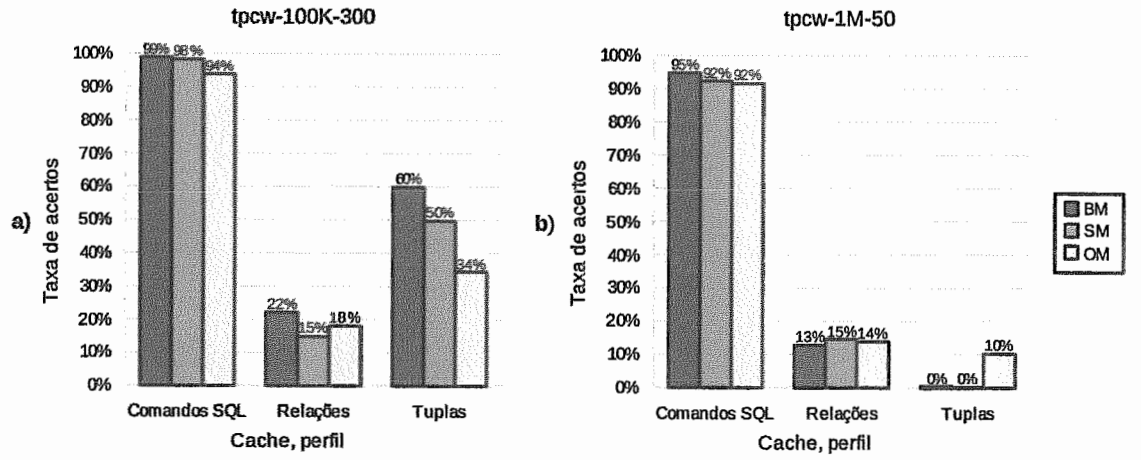


Figura 3.5: Variação nas taxas de acerto das caches de acordo com o perfil e para as configurações tpcw-100K-300 e tpcw-1M-50 (apenas 1 Tomcat).

explicado em 2.1.9) e os índices utilizados nas análises baseiam-se exatamente no comportamento das caches. Depois, porque a diferença no número absoluto de interações Web por minuto obtidas pelos dois trabalhos é muito grande, o que implica em variações bruscas em alguns índices entre os dois trabalhos, como a taxa de ocupação das caches. A diferença entre a quantidades absolutas de interações obtidas pelos dois trabalhos pode ter sido causada pelas seguintes hipóteses:

- O SGBD utilizado por [40] era o MySQL versão 4.1.4, enquanto nos novos experimentos utilizamos o Postgres.
- Em [40] foi utilizada a *engine* de armazenamento MyISAM [43] que não dá suporte a transações, enquanto nos novos experimentos utilizamos o nível de isolamento de transações padrão do Postgres (*read committed*).
- Os processadores usados por [40] para o servidor de banco de dados e para o servidor Apache possuíam 2.8 GHz de clock e 1 MB de cache, enquanto os usados nos novos experimentos possuíam apenas 2.4 GHz de clock e 512 MB de cache.
- O kernel utilizado por [40] para o servidor de banco de dados e para o servidor Apache possuía SMP habilitado, enquanto nos novos experimentos este modo estava desabilitado.
- A topologia da rede utilizada por [40] é diferente da topologia empregada nos novos experimentos. Em [40], foi utilizado um único switch para interconexão de todas as máquinas envolvidas, enquanto neste trabalho utilizamos dois switchs para conectar duas sub redes. É importante notar que o RBE fica na primeira sub rede, o servidor Apache na segunda sub rede, os servidores Tomcat na primeira e o SGBD na segunda. Ou seja, uma requisição realizada por um navegador emulado trafega por dois switchs até alcançar o servidor Apache, retorna pelos mesmos dois switchs para alcançar um servidor Tomcat, passa novamente pelos dois switches para acessar o banco de dados e depois que consegue os dados desejados do banco, precisa voltar pelo mesmo caminho até levar os dados de resposta ao navegador.
- Algumas modificações efetuadas na implementação do benchmark TPC-W para executar os novos experimentos não foram realizadas por [40] (as novas modificações já foram listadas na seção 3.1.1).
- As versões dos servidores Apache e Tomcat, a versão do JRE e a versão de kernel Linux usada por [40] são diferentes das versões utilizadas pelos novos experimentos[7].

[7]Em [40] é usado o Apache versão 2.0.46, o Tomcat versão 5.0.28 e Linux kernel versão 2.6.7. O JRE

Analisemos agora os resultados da segunda série de experimentos (tpcw-1M-50). O número de interações Web por minuto obtidos na segunda série, com e sem uso do driver dCache, por [40] são mostrados na figura 3.6 *a*. A partir das quantidade de interações Web por minuto mostradas no gráfico, percebemos que, graças ao uso do driver dCache, ocorre um ganho de desempenho de 129% para o perfil BM, de 8% para o perfil SM e de 7% para o perfil OM.

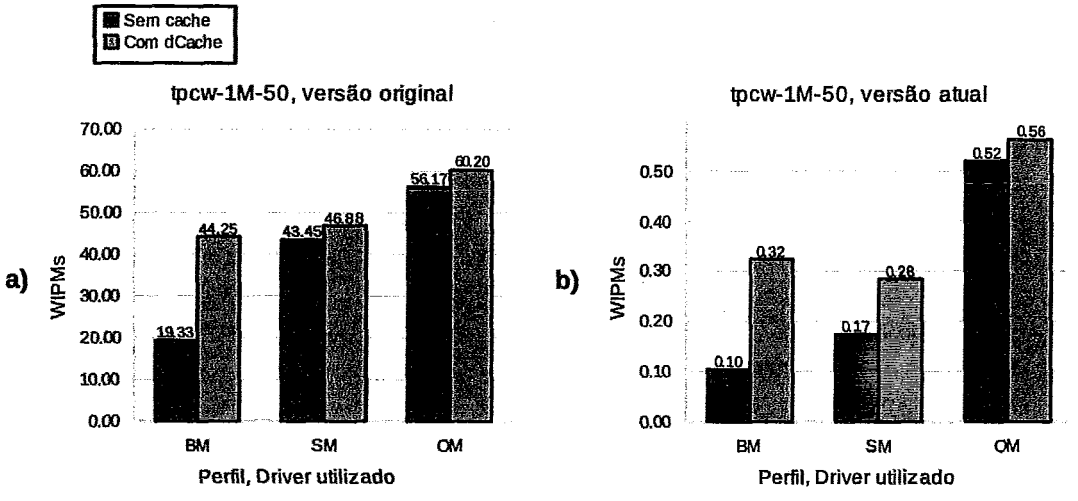


Figura 3.6: Gráfico de interações Web por minuto para a configuração tpcw-1M-50, com e sem o driver dCache (apenas 1 Tomcat).

Quando repetimos as medições da segunda série utilizando a nova versão da arquitetura centralizada, obtivemos os resultados mostrados na figura 3.6 *b*. Assim como na configuração tpcw-100K-300, para chegarmos aos valores no topo de cada uma das barras do gráfico, realizamos 5 medições, descartamos as que apresentaram o maior e o menor valor e realizamos a média aritmética simples das 3 medições restantes. Nesta série, o ganho de desempenho alcançado no perfil BM sobe para 212%, sobe para 63% para o perfil SM e sobe para 8% no perfil OM. Novamente, as quantidades absolutas de interações Web por minuto caem bruscamente em todos os perfis, independentemente do uso ou não do driver dCache.

Embora a nova implementação da arquitetura centralizada tenha também aumentado os ganhos de desempenho para a configuração tpcw-1M-50, é importante observar que as melhorias mais significativas ocorrem nos perfis BM e SM, contrariando assim, a hipótese de que teria sido o novo modelo de invalidações – atuando sobre as configurações

é versão 1.5.0 porém, enquanto usamos o *build* 1.5.0_15, nos experimentos realizados por [40] foi usado *build* 1.5.0_03.

com mais operações de escrita – o responsável pelo aumento do desempenho observado entre [40] e o presente trabalho. A relação entre operações de escrita e o novo mecanismo de invalidação pode sim impactar positivamente nos resultados obtidos com o uso do driver dCache, porém, outros fatores, como o peso de consultas complexas e/ou envolvendo muitos registros[8] também precisam ser levados em consideração para explicar as variações obtidas nos resultados.

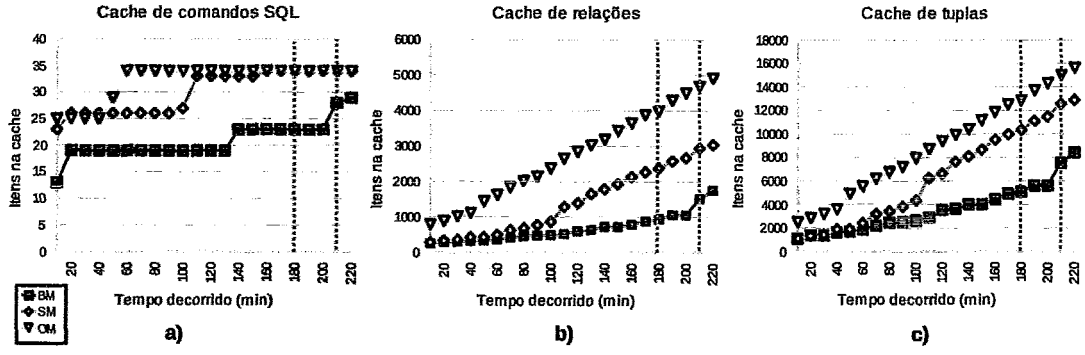


Figura 3.7: Variação na ocupação das caches com o tempo de execução do benchmark para a configuração tpcw-1M-50 e para os perfis de navegação (apenas 1 Tomcat).

A figura 3.7 mostra a variação na ocupação das caches com o tempo de execução do benchmark usando a configuração tpcw-1M-50, para os três perfis de navegação e para as três caches do sistema. Comparando-se os gráficos da figura 3.7 com os da figura 3.4, é interessante observar que, mesmo com mais tempo de decorrido de execução, o tamanho máximo atingido pelas caches é muito maior na primeira série do que na segunda série de experimentos. Essa diferença certamente se deve ao fato da segunda série utilizar apenas 50 navegadores emulados, contra 300 da primeira série. O número menor de navegadores acaba impondo menor carga de trabalho ao driver resultado em menor ocupação das caches.

A exceção é para a cache de comandos SQL que apresenta taxas muito semelhantes entre as duas séries. Isso ocorre pois os comandos SQL utilizados pelo benchmark são sempre os mesmos, independentemente da configuração utilizada (tpcw-100K-300 ou tpcw-1M-50). As taxas de acertos nas caches para a segunda série de experimentos é mostrada no gráfico *b* da figura 3.5. Comparando este gráfico com o gráfico *a* da mesma figura, notamos que as taxas de acertos da segunda série de experimentos são menores do

[8]Este é exatamente o caso da configuração tpcw-1M-50, que utiliza uma base de dados muito grande, fazendo com que as consultas precisem operar sobre grande quantidades de tuplas, sobrecarregando o SGBD, como foi mostrado em [40].

que na primeira série, com exceção da cache de comandos SQL. A justificativa para esta exceção é a mesma: os comandos SQL usados pelas execuções do benchmark são sempre os mesmos.

É interessante notar que a taxa de acertos na cache de comandos SQL é maior do que 90% em todas as execuções do benchmark. A alta taxa de acertos ocorre pois o benchmark utiliza um conjunto de comandos SQL pequeno, mesmo com a geração de consultas dinâmica implementada por este trabalho (conforme explicado na seção 3.1.1). Mostramos assim, que, graças à baixa participação das consultas geradas dinamicamente no conjunto total de consultas utilizado pelo benchmark, a taxa de acertos à esta cache permanece alta, conforme havíamos previsto na seção 3.1.1.

Finalizando a comparação entre as duas implementação, registremos mais duas observações interessantes. Primeiramente, através dos gráficos *b* e *c* da figura 3.4, podemos observar que, para a configuração tpcw-100K-300, o perfil BM é aquele que apresenta maior taxa de ocupação das caches de relações e tuplas durante todo o tempo de execução do benchmark. Já no perfil tpcw-1M-50, o perfil OM toma o lugar do BM, ocupando mais as caches do que os demais perfis, conforme observamos nos gráficos *c* da figura 3.7.

A última observação se refere a estabilidade dos resultados obtidos. Dissemos que os valores no topo das barras apresentadas nos gráficos 3.3 *b* e 3.6 *b* são obtidos a partir de 5 medições. A tabela 3.1 apresenta os desvios padrão das 5 medições realizadas para cada perfil e configuração, com e sem o uso do driver dCache. Observando os valores apresentados por esta tabela, verificamos que os dados do perfil tpcw-1M-50 apresentam variações muito mais expressivas do que os do perfil tpcw-100K-300, seja usado ou não o driver dCache.

	Desvios padrão			
	tpcw-100K-300		**tpcw-1M-50**	
	com dCache	sem dCache	com dCache	sem dCache
BM	5,31%	1,62%	36,29%	42,42%
SM	4,97%	6,86%	14,43%	40,14%
OM	5,28%	2,90%	16,63%	16,76%

Tabela 3.1: Desvios padrão nas medições realizadas com o driver dCache e com o driver Postgres.

3.4.2 Medindo a sobrecarga da versão distribuída

Para responder a segunda questão realizamos medições utilizando as configurações tpcw-10K-25 e tpcw-10K-100, sempre empregando apenas um servidor de aplicação. Primeiramente, realizamos medições utilizando a versão centralizada do driver e depois, repetimos as medições utilizando a versão distribuída do driver. Cada medição durou 1 hora, precedida por um período de aquecimento de 30 minutos. Para cada uma das duas versões (centralizada e distribuída), para cada uma das duas configurações (tpcw-10K-25 e tpcw-10K-100) e para cada um dos três perfis de navegação (BM, SM e OM), realizamos 5 medições (ou seja, foram realizadas, no total, 60 medições). De cada grupo de 5 medições, descartamos o maior e o menor valor de WIPM obtido.

A tabela 3.2 mostra os resultados obtidos para as duas configurações. Os gráficos da figura 3.8 apresentam os dados da mesma tabela. Como poderíamos esperar (devido ao *overhead* adicional imposto pela versão distribuída), quando é utilizado um único servidor de aplicação, a versão centralizada do dCache proporciona maior quantidade de interações Web por minuto, qualquer que seja a configuração e o perfil de navegação utilizado no benchmark.

	tpcw-10K-25		tpcw-10K-100	
	Centralizada	Distribuída	Centralizada	Distribuída
BM	1868.57	1850.55	2476.23	2474.92
SM	1515.78	1480.99	2175.75	2105.69
OM	1514.36	1473.8	1684.88	1574.22

Tabela 3.2: Interações Web por minutos proporcionadas pelas versões centralizada e distribuída do driver dCache para os três perfis (usando apenas um 1 Tomcat).

3.4.3 Medindo o driver Postgres com vários Tomcats

Para responder a terceira pergunta realizamos medições utilizando as configurações tpcw-10K-25 e tpcw-10K-100, usando de 1 até 3 servidores Tomcat. Cada medição durou 1 hora, precedida por um período de aquecimento de 30 minutos. Para cada uma das duas configurações (tpcw-10K-25 e tpcw-10K-100), para cada um dos três perfis de navegação (BM, SM e OM) e para cada número de Tomcats (1, 2 e 3), realizamos 5 medições (ou seja foram realizadas, no total, 90 medições). De cada grupo de 5 medições, descartamos o maior e o menor valor obtido. As quantidades de WIPMs apresentadas nas tabelas 3.3 *a* e *b* correspondem a média das três medições remanescentes de cada grupo de 5 medições.

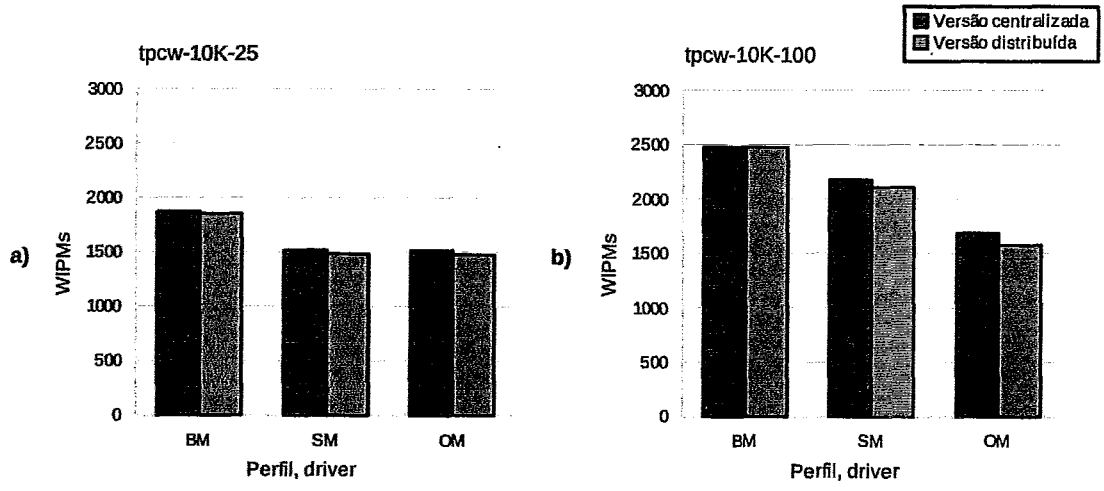


Figura 3.8: Variação nas interações Web por minutos proporcionadas pelas versões centralizada e distribuída do driver dCache para os três perfis (usando apenas um 1 Tomcat).

Nas tabelas 3.4 *a* e *b* são mostrados os desvios padrão, porém considerando as cinco medições e não só as 3 usadas para o cálculo da média.

a)

10K itens, 25 navegadores

	1 Tomcat	2 Tomcats	3 Tomcats
BM	1567.75	1463.65	1505.28
SM	1370.9	1304.41	1279.23
OM	1410.49	1378.58	1336.65

b)

10K itens, 100 navegadores

	1 Tomcat	2 Tomcats	3 Tomcats
BM	1933.83	1749.73	1695.44
SM	1652.96	1501.01	1477.85
OM	1387.69	1316.45	1316.7

Tabela 3.3: Interações Web por minuto para as configurações tpcw-10K-25 e tpcw-10K-100, usando o driver Postgres, com 1, 2 e 3 Tomcats.

O gráfico apresentado na figura 3.9 representa os dados da tabela 3.3. Observando este gráfico, podemos notar que o uso de apenas um Tomcat é sempre mais vantajoso do que o uso de 2 ou 3 Tomcats. O uso de 2 Tomcats é mais vantajoso do que o o uso de 3 Tomcats nos perfis SM e OM da configuração tpcw-10K-25 e nos perfis BM e SM da configuração tpcw-10K-100. O uso de 3 Tomcats é mais vantajoso do que 2 Tomcats apenas no perfil BM da configuração tpcw-10K-25 e no perfil OM da configuração tpcw-10K-100, casos em que ele ganha com muito pouca margem. Ou seja, a partir das medições realizadas, concluímos que não faz sentido aumentar o número de Tomcats para

a)

10K itens, 25 navegadores

	1 Tomcat	2 Tomcats	3 Tomcats
BM	3.68%	3.66%	9.84%
SM	2.58%	3.52%	6.71%
OM	3.76%	4.71%	2.62%

b)

10K itens, 100 navegadores

	1 Tomcat	2 Tomcats	3 Tomcats
BM	2.03%	4.31%	3.93%
SM	7.25%	7.93%	10.37%
OM	6.29%	15.23%	4.83%

Tabela 3.4: Desvios padrão nas interações Web por minuto para as configurações tpcw-10K-25 e tpcw-10K-100, usando o driver Postgres, com 1, 2 e 3 Tomcats.

melhorar o desempenho do sistema em teste, quando o driver Postgres é utilizado.

Os resultados apresentados mostram que a replicação dos servidores de aplicação, por si só, não se reverte em melhor desempenho do sistema em teste. Esses resultados correspondem a nossa expectativa, pois, como foi mostrado em [40], o "gargalo" do sistema em estudo encontra-se no servidor de banco de dados, e não nos servidores Tomcat. Logo, técnicas que se valham da replicação dos servidores de aplicação para melhorar o desempenho do sistema em teste precisam estar voltadas para a redução da necessidade de acesso ao SGBD.

Outra observação interessante é que, além de não melhorar o desempenho do sistema em teste, o uso de mais servidores de aplicação piora o seu desempenho em todos os casos. Note que o driver Postgres não possui nenhuma estratégia de comunicação entre as suas instâncias. Ou seja, durante os experimentos, cada instância do driver Postgres se comporta como se estivesse sozinha no sistema. Isso mostra que, para obter ganhos de desempenho, a versão distribuída do driver dCache precisará, minimamente, proporcionar uma quantidade adicional de interações Web por minuto suficiente para sobrepujar o perda de desempenho causada pela sobrecarga inerente ao uso de mais servidores de aplicação.

Trabalhos futuros poderão ser realizados para investigar a causa desta sobrecarga. Uma hipótese é o mecanismo de balanceamento de carga utilizado no servidor Web Apache, que precisa intermediar a comunicação entre os navegadores emulados com um número maior de servidores de aplicação. Outra hipótese, é a sobrecarga adicional que pode estar sendo causada no SGBD que, apesar de lidar com a mesma quantidade de conexões (pois não variamos o número de navegadores emulados), precisa lidar com um número possivelmente maior de conexões concorrentes, geradas pelos servidores de

aplicação em paralelo.

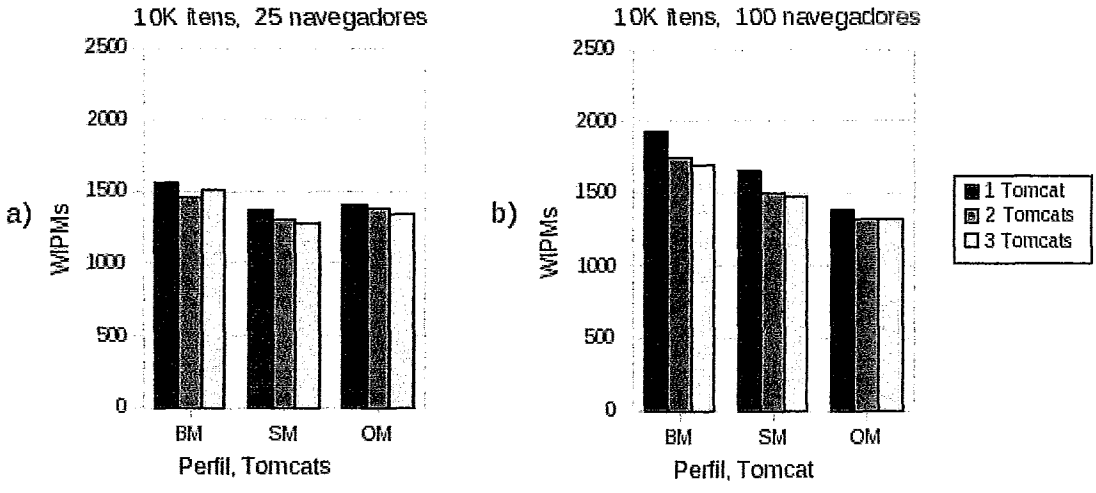


Figura 3.9: Gráfico de interações Web por minuto para as configurações tpcw-10K-25 e tpcw-10K-100, usando o driver Postgres, com 1, 2 e 3 Tomcats.

3.4.4 Medindo o driver dCache com vários Tomcats

Para responder a quarta e última questão, realizamos medições utilizando as configurações tpcw-10K-25, tpcw-10K-100, tpcw-100K-300 e tpcw-1M-50, utilizando 2 e 3 Tomcats. Primeiramente, vamos apresentar e analisar os resultados obtidos com as configurações tpcw-10K-25 e tpcw-10K-100. Veremos que, para estas duas configurações, o emprego de apenas um servidor de aplicação proporciona melhor desempenho do que o uso de mais servidores, não justificando a utilização da arquitetura distribuída do driver.

Em seguida, vamos apresentar os resultados obtidos com as configurações tpcw-100K-300 e tpcw-1M-50, que utilizam maior número de navegadores e maior tamanho de base de dados do que as duas primeiras. Com essas duas últimas configurações, veremos que o uso de mais servidores de aplicação começa a proporcionar maior desempenho, justificando a utilização da arquitetura distribuída e apontando como direção de investigação futura, a avaliação do driver dCache através de configurações com ainda mais servidores de aplicação e bases de dados maiores.

Configurações tpcw-10K-25 e tpcw-10K-100

Utilizando as configurações tpcw-10K-25 e tpcw-10K-100, com 2 e 3 servidores de aplicação, cada medição durou 1 hora, precedida por um período de aquecimento de 30 minutos. Em ambas as configurações, utilizamos uma cache de relações com capacidade máxima de 35 mil itens e uma cache de tuplas com capacidade máxima de 80 mil itens. Escolhemos estas capacidades de forma que as caches fossem capazes de armazenar, com uma pequena folga, todos os itens inseridos durante o experimento, sem necessidade de substituição. Em outras palavras, o tamanho máximo atingido pelas caches não alcançou suas capacidades máximas em nenhuma medição.

Para cada uma das configurações, para cada um dos três perfis de navegação e para cada número de Tomcats, realizamos 5 medições. As tabelas 3.5 *a* e *b* mostram, respectivamente, a quantidade média de iterações Web por minuto e o desvio padrão das medições com a configuração tpcw-10K-25. Para o cálculo da média, foram utilizadas apenas as 3 medições restantes, depois de eliminadas aquelas que reportaram a maior e a menor quantidade de interações Web. No cálculo do desvio padrão, foram consideradas as 5 medições realizadas.

a)

WIPM			
	1 Tomcat	2 Tomcats	3 Tomcats
BM	1850,55	1495,84	1239,11
SM	1480,99	1225,84	1061,83
OM	1473,80	1112,95	905,88

b)

Desvios padrão			
	1 Tomcat	2 Tomcats	3 Tomcats
BM	5,23%	3,37%	4,52%
SM	8,93%	4,59%	1,64%
OM	6,02%	3,62%	2,52%

Tabela 3.5: Médias e desvios padrão nas interações Web por minuto conseguidas utilizando a versão distribuída do dCache para a configuração tpcw-10K-25.

As tabelas 3.6 *a* e *b* mostram, respectivamente, a quantidade média de iterações Web por minuto e o desvio padrão das medições com a configuração tpcw-10K-100. Assim como na configuração tpcw-10K-25, a média foi calculada apenas com 3 medições e o desvio padrão com 5 medições. Para comparação, tanto nas tabelas da configuração tpcw-10K-25, quanto nas tabelas da configuração tpcw-10K-100, também são apresentados os dados obtidos quando foi utilizado apenas um Tomcat, mesmo estes dados já tendo sido apresentados na seção 3.4.2.

Observando os gráficos apresentados na figura 3.10 (que apresenta os dados das tabelas 3.5 *a* e 3.6 *a*), notamos que o aumento do número de Tomcats não melhora o desempenho do sistema em teste quando utilizamos a versão distribuída do driver dCache. Assim como nas medições realizadas com o uso do driver Postgres, o melhor desempenho

a)

	WIPM		
	1 Tomcat	2 Tomcats	3 Tomcats
BM	2474.92	2143.45	1806.99
SM	2105.69	1592.2	1357.34
OM	1574.22	516.21	390.19

b)

	Desvios padrão		
	1 Tomcat	2 Tomcats	3 Tomcats
BM	3.76%	1.64%	5.41%
SM	1.47%	15.27%	7.72%
OM	8.53%	14.37%	10.11%

Tabela 3.6: Médias e desvios padrão nas interações Web por minuto conseguidas utilizando a versão distribuída do dCache para a configuração tpcw-10K-100.

ocorre sempre quando é utilizado apenas um Tomcat. E diferentemente das medições com o driver Postgres, o uso de apenas 2 Tomcats tem sempre melhor desempenho quando comparado com o uso de 3 Tomcats.

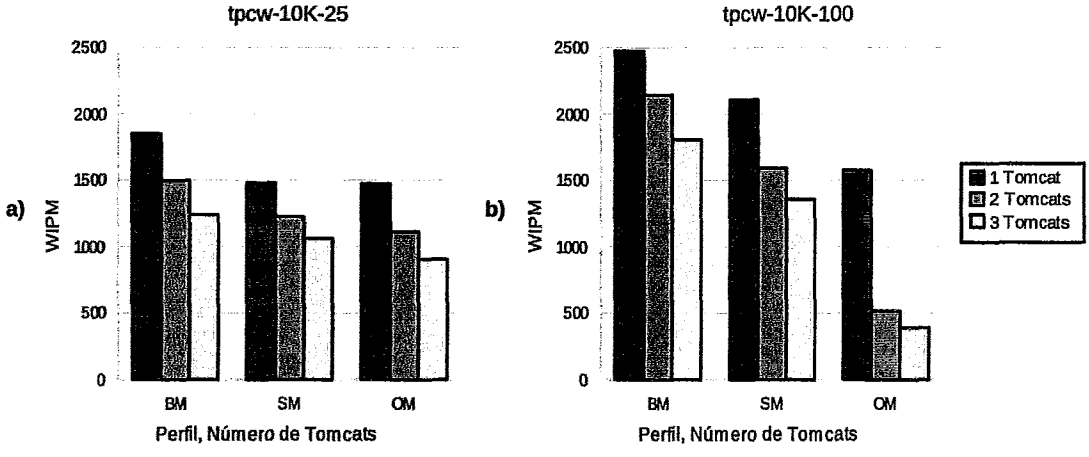


Figura 3.10: Variação, com o número de Tomcats e perfis, das WIPM conseguidas com uso do driver dCache distribuído com as configurações tpcw-10K-25 e tpcw-10K-100.

Todas as máquinas utilizadas no sistema em teste, inclusive os servidores de aplicação – onde o driver dCache executa, apresentaram taxas de utilização da CPU muito baixas, permanecendo quase 100% do tempo ociosas, independentemente da configuração e quantidade de Tomcats utilizada nos experimentos. Uma exceção é o servidor de banco de dados, cuja CPU, apesar de também permanecer ociosa por quase 100% do tempo com a configuração tpcw-10K-25, apresenta 5% de utilização quando a configuração utilizada é a tpcw-10K-100.

Para entendermos o porquê da queda de desempenho do sistema com o aumento do número de servidores de aplicação, nos cabe agora, analisar em maiores detalhes o comportamento do driver no decorrer da execução do benchmark. Vamos focar nos resultados obtidos com configuração tpcw-10K-25, para o perfil BM. Os dados apresentados na tabela 3.7 mostram a variação da quantidade de execuções dos métodos mais

significativos da implementação do driver dCache com o aumento do número de servidores de aplicação. Na coluna *"Execuções Por Instância"*, são mostradas as quantidades de execuções dos métodos do ponto de vista de uma única instância do driver dCache[9]. Na coluna *"Total de Execuções"*, são consideradas as execuções realizadas por todas as instâncias envolvidas no experimento.

Nome do método	Execuções Por Instância			Total de Execuções		
	1 Tomcat	2 Tomcats	3 Tomcats	1 Tomcat	2 Tomcats	3 Tomcats
Connection#prepareStatement(String)	32461,0	16191,0	10746,0	32461	32382	32238
Connection#commit()	28520,0	14194,5	9414,7	28520	28389	28244
PreparedStatement#executeQuery()	31320,0	15607,0	10358,0	31320	31214	31074
PreparedStatement#executeUpdate()	764,0	388,5	262,0	764	777	786
ResultSet#next()	281684,0	141129,5	93469,7	281684	282259	280409
ResultSet#getValue(int)	1803748,0	905474,5	603033,0	1803748	1810949	1809099

Tabela 3.7: Quantidade de execuções dos principais métodos da API JDBC implementados pelo driver dCache, para a configuração tpcw-10K-25, com o perfil BM.

Observando os dados da tabela 3.7, podemos notar que, mesmo com a utilização de mais servidores de aplicação, a quantidade total de execuções dos métodos não aumenta significativamente. Pelo contrário, em alguns casos, ela até decai. Essa observação está de acordo com a constatação de que o uso de mais instâncias do driver dCache não melhora o desempenho do sistema em teste. Já do ponto de vista de uma única instância, a queda no número de execuções é bem clara. Isso indica que, com o aumento do número de servidores de aplicação, as execuções passam a ser distribuídas pelas instâncias que, individualmente, realizam menos execuções, enquanto a soma das execuções realizadas por todas as instâncias sofre pouca variação.

Vejamos agora, como o tempo total gasto com a execução de cada um destes métodos varia em função do número de Tomcats empregados (tabela 3.8). Do ponto de vista de uma instância do driver, com exceção do método `PreparedStatement#executeQuery()`, o tempo total gasto com a execução de todos os métodos diminui. Porém, do ponto de vista global (somando-se o tempo de todas as instâncias), o tempo total gasto com a execução de cada um dos métodos aumenta. A explicação para este fato está contida na tabela 3.9, onde podemos observar que o tempo médio de execução de todos os métodos aumenta, conforme são empregados mais servidores de aplicação nos experimentos.

[9]Para chegarmos a estes valores, para cada método, somamos as execuções realizadas por cada uma das instâncias e dividimos pelo número de instâncias. Por exemplo: Se são utilizados dois Tomcats `T1` e `T2`, para chegar a quantidade de execuções de um método `M`, somamos a quantidade de execuções de `M` realizadas por `T1` com quantidade de execuções de `M` realizadas por `T2` e dividimos por 2.

Nome do método	Tempo Total Por Instância (ms)			Tempo Total Global (ms)		
	1 Tomcat	2 Tomcats	3 Tomcats	1 Tomcat	2 Tomcats	3 Tomcats
Connection#prepareStatement(String)	2617,0	2163,5	1572,3	2617	4327	4717
Connection#commit()	5034,0	3292,0	2455,3	5034	6584	7366
PreparedStatement#executeQuery()	1267,0	50752,5	63209,7	1267	101505	189629
PreparedStatement#executeUpdate()	697,0	380,5	265,7	697	761	797
ResultSet#next()	157717,0	127041,5	112834,3	157717	254083	338503
ResultSet#getValue(int)	4748,0	2899,5	2012,3	4748	5799	6037

Tabela 3.8: Tempo total gasto em execuções dos principais métodos da API JDBC implementados pelo driver dCache, para a configuração tpcw-10K-25, com o perfil BM.

Nome do método	Tempo Médio (ms)		
	1 Tomcat	2 Tomcats	3 Tomcats
Connection#prepareStatement(String)	0,0806	0,1336	0,1463
Connection#commit()	0,1765	0,2319	0,2608
PreparedStatement#executeQuery()	0,0405	3,2519	6,1025
PreparedStatement#executeUpdate()	0,9123	0,9794	1,0140
ResultSet#next()	0,5599	0,9002	1,2072
ResultSet#getValue(int)	0,0026	0,0032	0,0033

Tabela 3.9: Tempo médio gasto em cada execução dos principais métodos da API JDBC implementados pelo driver dCache, para a configuração tpcw-10K-25, com o perfil BM.

Os gráficos mostrados na figura 3.11 apresentam os dados da tabela 3.9. Os gráficos *a* e *b* apresentam os mesmos valores, porém, para melhor visualização dos tempos de execução muito discrepantes entre métodos, o gráfico *b* utiliza escala logarítmica no eixo de tempo de execução.

Vamos investigar em maiores detalhes o método `PreparedStatement#executeQuery()`, que sofre os aumentos mais abruptos no seus tempos de execução. Do ponto de vista de uma única instância do driver, o tempo total de execução do método `executeQuery()` aumenta em mais de 40 vezes quando o número de Tomcats sobe para 2 e em, aproximadamente, 1,2 vezes quando o número de Tomcats sobe para 3. O tempo médio de execução deste método aumenta em mais de 80 vezes na passagem de 1 para 2 Tomcats e em, aproximadamente, 1,9 vezes quando o número de Tomcats sobe para 3. Note que as variações mais abruptas ocorrem exatamente na passagem de 1 para 2 Tomcats, quando também se nota uma queda abrupta na quantidade de interações Web por minuto reportadas pelo benchmark.

O tempo de execução do método `executeQuery()` se divide nas seguintes sub atividades: realização de teste de validade, obtenção de rótulo de tempo a partir do servidor dCache, acesso à cache de relações e à tabela de modificações globais (ver seções 2.1.3 e 2.4.1). Para cada quantidade de Tomcats utilizada, a tabela 3.10 mostra o tempo total gasto com a execução de cada atividade e o tempo médio de cada execução. Os

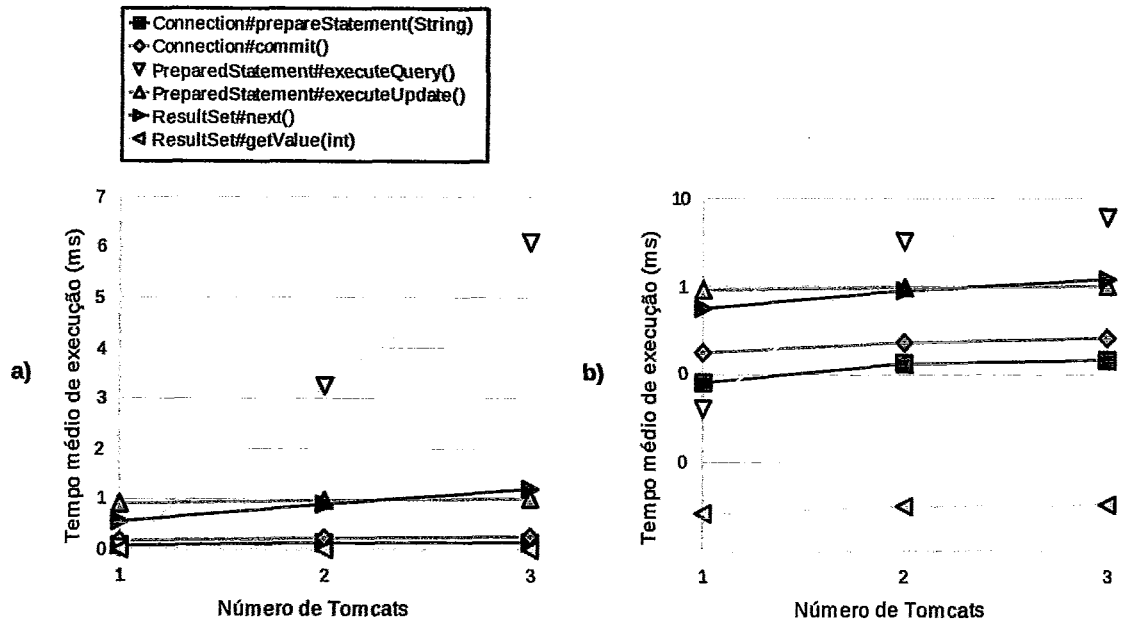


Figura 3.11: Variação, com o número de Tomcats, dos tempos médios de execução dos principais métodos da API JDBC implementados pelo driver dCache, para o perfil BM, configuração tpcw-10K-25.

valores apresentados nesta tabela são do ponto de vista de uma única instância do driver.

Nome das atividades do método *executeQuery()*	Tempo total de execução (ms)			Tempo médio de execução (ms)		
	1 Tomcat	2 Tomcats	3 Tomcats	1 Tomcat	2 Tomcats	3 Tomcats
Teste de validade	544,0	316,5	211,7	0,8690	0,8242	0,5605
Obtenção de rótulo de tempo	5,0	8,0	6,0	0,0080	0,0208	0,0159
Acesso cache de relações	217,0	50000,5	62677,7	0,0071	3,2855	6,2814
Acesso tabela de modificações global	277,0	154,0	107,0	0,0014	0,0016	0,0017

Tabela 3.10: Tempo total e médio gasto em cada execução das atividades do método `executeQuery()`, para a configuração tpcw-10K-25, com o perfil BM.

Os gráficos apresentados na figura 3.12 mostram a variação dos tempos médios de execução das atividades realizadas pelo método `executeQuery()`, com base na tabela 3.10. Os gráficos *a* e *b* apresentam os mesmos valores, porém, o gráfico *b* utiliza escala logarítmica devido a grande discrepância nos tempos de execução das atividades. Observando os gráficos apresentados na figura 3.12, percebemos que a grande responsável pelo aumento no tempo médio de execução do método `executeQuery()` é a atividade "acesso a cache de relações". Para o tempo médio de execução desta atividade, ocorre um aumento de mais de 462 vezes quando o número de Tomcats utilizados é aumentado para 2 e de, aproximadamente, 1,9 vezes quando o número de Tomcats utilizados sobe para 3.

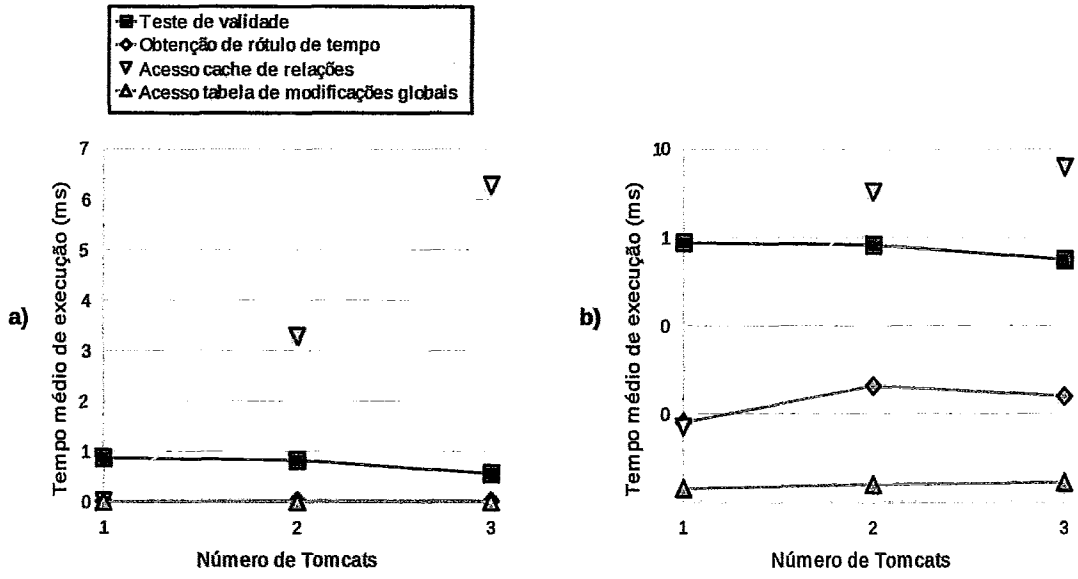


Figura 3.12: Variação dos tempos médios de execução das atividades realizadas pelo método `executeQuery()` com o número de Tomcats, para o perfil BM, configuração tpcw-10K-25.

Neste ponto, será interessante analisarmos os dados gerados pela instrumentação das caches. A tabela 3.11 mostra os dados coletados através da instrumentação das caches de relações e de tuplas. Os dados apresentados correspondem a média das medições coletadas nos servidores de aplicação usados em cada experimento, em outras palavras, são dados do ponto de vista de uma instância do driver. A primeira observação que fazemos a partir da tabela 3.11 é que o tempo médio de acesso às caches é diretamente proporcional ao tempo médio de acesso remoto. Estamos considerando como *tempo médio de acesso remoto*, o tempo gasto por uma instância do driver dCache para solicitar, à outra instância do driver (e aguardar retorno), um item que não pode ser obtido a partir da sua cache local.

	Cache de Relações			Cache de Tuplas		
	1 Tomcat	2 Tomcats	3 Tomcats	1 Tomcat	2 Tomcats	3 Tomcats
Número de acessos realizados	30691,00	15218,50	9978,33	247413,00	123493,50	82419,67
Tempo médio de acesso (ms)	0,01	3,29	6,28	0,00	0,06	0,15
Taxa de acertos global	55,93%	54,04%	53,39%	80,16%	80,08%	80,26%
Taxa de acertos local	55,93%	42,25%	35,31%	80,16%	72,25%	68,38%
Taxa de acertos remoto	0,00%	20,42%	21,51%	0,00%	28,21%	33,81%
Tempo médio de acesso remoto (ms)	0	7	13	0	1	2
Tempo total gasto com acessos remotos (ms)	33,00	49865,00	62579,33	70,00	28509,50	39004,33
Tamanho máximo atingido	13951,00	9025,50	6678,33	49090,00	34260,50	26058,00

Tabela 3.11: Dados de instrumentação das caches de relações e tuplas, para a configuração tpcw-10K-25, perfil BM.

Obviamente, o aumento do tempo médio de acesso remoto, faz com que o tempo total gasto com acessos remotos também aumente. Voltando a tabela 3.10, podemos observar que o tempo total de execução da atividade "acesso à cache de relações" é aproximadamente o mesmo que aparece na tabela 3.11, na linha de tempo total gasto com acesso remoto para a cache de relações. Ou seja, o aumento do tempo gasto acessando a cache de relações está diretamente relacionado com o tempo gasto realizando acessos à caches remotas.

Apesar de ser evidente o aumento no tempo médio de acesso às caches, devido ao aumento no tempo de acesso às caches remotas, ele não seria, por si só, justificativa para a queda de desempenho no sistema em teste. Nossa expectativa, na realidade, era que o desempenho do sistema em teste melhorasse devido ao aumento na taxa de acertos em cache proporcionada pela possibilidade de acesso à caches de outras instâncias do driver dCache. Um curioso fenômeno é então observado quando voltamos à tabela 3.11 para analisar as taxas de acerto à cache experimentadas pelo benchmark.

Quando utilizamos caches distribuídas podemos dividir os acertos em cache em duas categorias: *acertos locais* e *acertos remotos*. Os acertos locais são obtidos quando os dados requisitados são encontrados na cache da própria instância do driver dCache onde foi realizada a sua requisição. Os acertos remotos são obtidos quando, após uma falta no acesso local, o dado é então encontrado através do acesso à uma cache localizada em outra instância do driver dCache.

A tabela 3.11 apresenta a taxa de acertos sob três óticas distintas. A *taxa global de acertos* é a taxa de acertos observando a cache como uma "caixa preta". Ou seja, é a taxa de acertos proporcionada pela cache independentemente da instância onde o dado requisitado foi encontrado. Ela é dada pela relação "número de acertos locais mais o número de acertos remotos, divididos pelo número de acessos. A *taxa de acertos locais* é dada pela relação "número de acertos locais sobre o número de acessos". Por fim, a *taxa de acertos remotos*, é dada pela relação "número de acertos remotos sobre o número de faltas locais". Ou seja, indica quantas vezes o dado é encontrado remotamente, uma vez que ele não pode ser encontrado localmente.

A taxa de acertos remoto para os experimentos utilizando apenas um servidor Tomcat é 0%, uma vez que não existem outras instâncias do driver dCache a serem consultadas quando ocorre uma falta no acesso à uma cache local. Utilizando dois Tomcats, a taxa de acertos remotos é de 20,42% para a cache de relações e 28,21% para a a cache de tuplas. Utilizando-se três Tomcats, as taxas de acertos sobem para 21,51% e 33,81%. Isso significa, por exemplo, que de cada 100 faltas no acesso à cache de relações local,

cerca de 20 destas faltas podem ser "revertidas" em acertos, utilizando-se para isso duas instâncias do driver dCache. Utilizando-se 1 instância a mais, conseguimos reverter cerca de 21. Intuitivamente, poderíamos esperar que a taxa global de acertos à essas caches aumentaria, no entanto, não é isso o que é mostrado pela tabela 3.11.

Para a cache de relações, a taxa global de acertos cai aproximadamente 1% com cada servidor de aplicação adicional. Para a cache de tuplas, a taxa global de acertos se mantém estável, em aproximadamente 80%, independentemente do número de servidores Tomcat utilizados. A explicação matemática para este comportamento não intuitivo, é que, enquanto a taxa de acertos remota aumenta com o número de Tomcats, a taxa de acertos locais decai, gerando um equilíbrio entre duas. Resta entender o porquê da queda na taxa de acertos locais com o aumento no número de servidores de aplicação utilizados.

As taxas de acertos apresentadas até o presente momento eram obtidas a partir do número de acessos e do número de acertos observados no final do período de medição, ou seja, após 1 uma hora e meia de execução do benchmark. Vamos agora analisar a variação das taxas de acertos na cache de relações ao longo da execução do benchmark (figura 3.13). Os gráficos *a*, *b* e *c* e *d* trazem, respectivamente, a variação da ocupação da cache, a variação das taxas de acerto global, local e remota. Uma linha vertical pontilhada ressalta o início das medições da quantidade interações Web por minuto. Isso porque, os dados apresentados pelo gráficos cobrem também o período de 30 minutos de aquecimento.

Observando as diferentes inclinações apresentadas pelas curvas dos diferentes números de Tomcats, notamos que a "velocidade" de ocupação da cache relações e a "velocidade" de crescimento das taxas de acertos diminui à medida que aumenta-se o número de servidores de aplicação. Podemos interpretar esse fato da seguinte forma: o aumento do tempo de acesso à cache de relações acaba "freando" a execução do benchmark, que é obrigado a permanecer aguardando dados da cache por mais tempo. Com isso, mesmo para intervalos de medição iguais, as taxas de ocupação e acertos apresentadas são menores quando se utiliza mais servidores de aplicação. Porém, se este intervalo de medição tendesse ao infinito, as taxas de ocupação e acertos iriam se estabilizar em um valor constante e neste momento, a taxa de global de acertos nos experimentos com maior número de Tomcats seria maior do que nos experimentos com menos Tomcats.

Para validar este hipótese, em trabalhos futuros precisaríamos repetir os experimentos utilizando intervalos de medição maiores. É importante notar, entretanto, que mesmo com taxas de acerto globais maiores, a estratégia de utilização de mais servidores de aplicação para melhorar o desempenho do sistema em teste ainda poderia ser ineficaz se o tempo gasto com a execução do driver continuasse sendo dilatado pelo tempo de

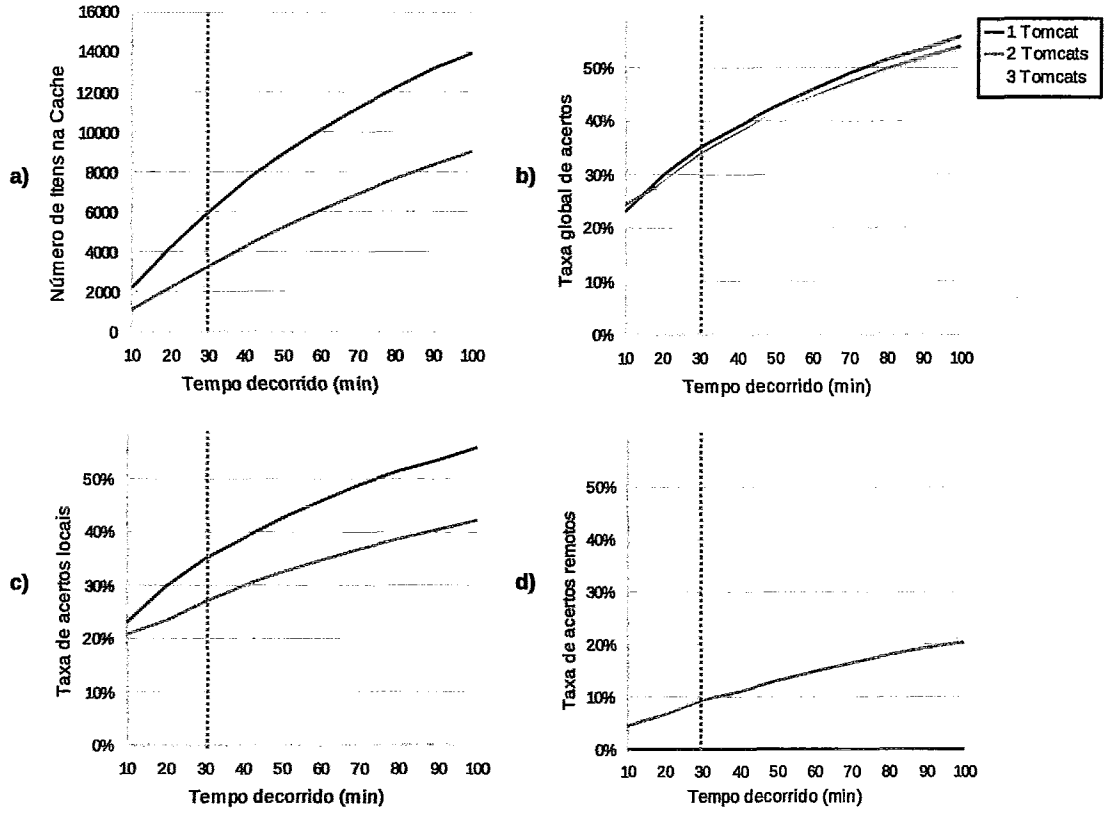


Figura 3.13: Taxas de acerto e de ocupação da cache de relações ao longo da execução do benchmark utilizando a configuração tpcw-10K-25 e perfil BM.

acesso às caches remotas.

Até agora, mantivemos nossa atenção na cache de relações, por termos concluído que ela seria a responsável pelo aumento no tempo de execução do código do driver dCache. Voltando aos dados apresentados na tabela 3.11, percebemos que a cache de tuplas possui comportamento muito semelhante ao da cache de relações. Por exemplo, assim como na cache de relações, o tempo total gasto com acesso remotos pela cache de tuplas também aumenta significativamente conforme aumenta-se o número de servidores de aplicação. Ficamos então interessados em entender como esse aumento interfere no tempo de execução do driver.

Enquanto que o acesso a cache de relações ocorre exclusivamente como resultado de invocações ao método `PreparedStatement#executeQuery()`, o acesso a cache de tuplas ocorre exclusivamente como resultado de invocações ao método `ResultSet#next()`. O método `next()`, por sua vez, realiza as seguintes atividades: obtenção de um *result set* a partir do driver escravo (chamaremos de atividade *getSlaveResultSet()*), invocação do método `next()` do *result set* obtido a partir do driver escravo, obtenção de

uma tupla a partir da relação encapsulada pelo *result set* (atividade *getTuple()*) e solicitação de crescimento[10] da relação encapsulada (atividade *grow()*).

A atividade *getSlaveResultSet()* é realizada para possibilitar a leitura, a partir do driver escravo, de valores que não puderam ser obtidos exclusivamente a partir do driver dCache. A invocação ao método `next()` do *result set* obtido a partir do driver escravo serve para posicionar o cursor deste *result set* na tupla de interesse. Essas duas primeiras atividades, obviamente, envolvem acesso ao driver escravo, e são realizadas apenas quando a relação obtida a partir da cache de relações não possui todas as tuplas solicitadas pela aplicação cliente do driver dCache.

O acesso à cache de tuplas pode ocorrer durante as atividades *grow()* e *getTuple()*. Dizemos "pode" ocorrer pois, como foi dito na seção 2.2, os *result sets* do driver dCache não fazem distinção entre relações de tuplas e relações de ponteiros para tuplas, enxergando apenas a interface `Relation`. Desta forma, o acesso à cache de tuplas acontecerá na execução das duas atividades mencionadas apenas quando a relação encapsulada pelo *result set* do driver dCache for uma relação de ponteiros para tuplas. A tabela 3.12 mostra, do ponto de vista de uma instância, a variação do tempo total de execução e do tempo médio por execução de cada uma das atividades realizadas pela implementação do método `next()`, a medida que o número de Tomcats utilizados no benchmark aumenta. Continuamos ainda focando somente no perfil BM, configuração tpcw-10K-25.

Nome das atividades do método *next()*	Tempo total de execução (ms)			Tempo médio de execução (ms)		
	1 Tomcat	2 Tomcats	3 Tomcats	1 Tomcat	2 Tomcats	3 Tomcats
getTuple()	325	7646,5	12011,00	0,0016	0,0764	0,1840
getSlaveResultSet()	154288	96089,5	71876,00	2,3381	2,7620	2,9881
next() driver escravo	114	72,5	122,00	0,0017	0,0021	0,0051
grow()	2431	22909,0	28600,67	0,0436	0,7743	1,3892

Tabela 3.12: Tempo total e médio gasto em cada execução das atividades do método `next()`, para a configuração tpcw-10K-25, com o perfil BM.

A figura 3.14 exibe dois gráficos que mostram a variação do tempo médio de execução das atividades com base na tabela 3.12. Os gráficos *a* e *b* exibem os mesmos dados, porém o segundo utiliza escala logarítmica. Analogamente ao comportamento dos tempos médios de execução das atividades realizadas pelo método `executeQuery()` (figura 3.12), os gráficos da figura 3.14 mostram que as atividades do método *next()* que

[10]Na seção 2.1.4, vimos que as relações encapsuladas pelos *result sets* do driver dCache podem estar completas ou incompletas. A "solicitação de crescimento" consiste na invocação do método `grow()` da interface `Relation` (seção 2.2) fazendo com que a relação encapsulada materialize mais uma tupla para atender uma solicitação da aplicação cliente do driver dCache.

envolvem acesso a cache de tuplas sofrem um abrupto aumento nos seus tempos médios de execução quando são utilizados mais servidores de aplicação. Considerando-se o conjunto de acessos realizados pelas atividades *getTuple()* e *grow()*, concluímos que o tempo de execução total gasto acessando a cache de tuplas quando são usados dois Tomcats é de aproximadamente 30555 ms. Usando 3 Tomcats, este valor sobe para aproximadamente 40612 ms.

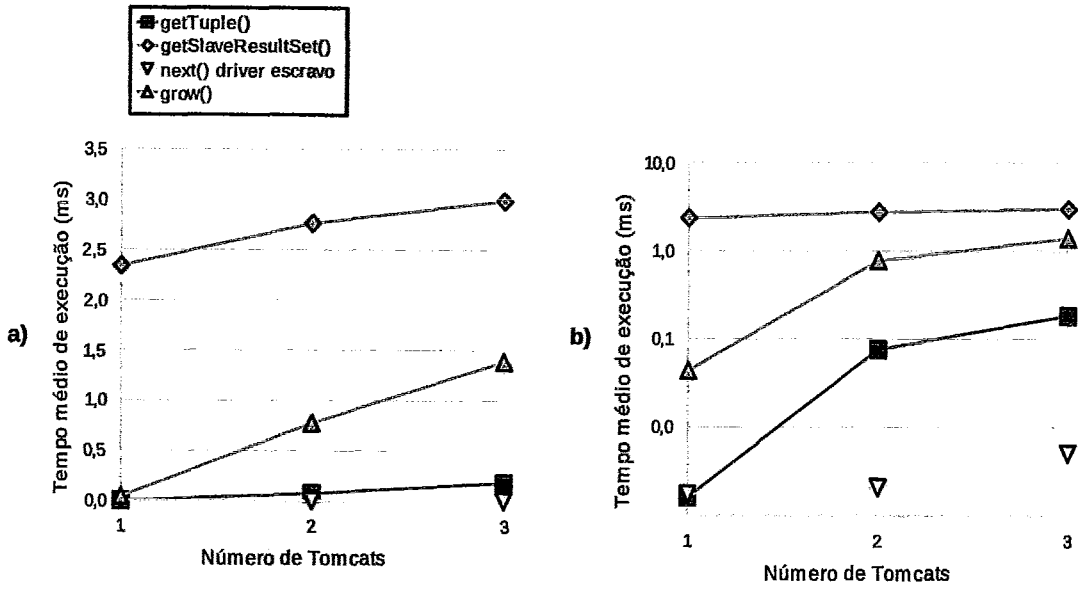


Figura 3.14: Variação, com o número de Tomcats, dos tempos médios de execução do método `ResultSet#next()` para o perfil BM, configuração tpcw-10K-25.

Voltando aos dados da cache de tuplas apresentados pela tabela 3.11, notamos que estes dois últimos valores acompanham o crescimento do tempo total gasto com acesso remotos. Ou seja, a justificativa dada para o aumento no tempo de acesso à cache de tuplas em função do aumento do número de Tomcats utilizados no benchmark é a mesma dada para o caso da cache de relações: reflexo do aumento no tempo de acesso remoto à cache. Vimos que no caso da cache de relações, este aumento resultou em aumento no tempo total gasto executando o método `executeQuery()`. No caso da cache tuplas, o aumento no seu tempo de acesso tem reflexo no tempo de execução do método `next()`. Embora bem menos expressivo do que o aumento sofrido pelo método `executeQuery()`, o aumento do método `next()` é um dos que se destaca dentre os demais, como pode ser observado nos gráficos apresentados na figura 3.11.

Ainda analisando os gráficos da figura 3.14, não podemos deixar passar desapercebido o crescimento do tempo médio de execução da atividade *getSlaveResultSet()*

e da chamada ao método `next()` do driver escravo. O crescimento no tempo médio de execução destas duas atividades mostra que o tempo de acesso ao driver escravo aumentou com o número de servidores de aplicação. Mas o acesso ao driver escravo não se restringe a estas duas sub atividades do método `next()`. Na verdade, com exceção do método `executeQuery()`, todos os demais acessam o driver escravo em algum momento. Será interessante entendermos como o tempo médio de acesso ao driver escravo se comporta em função da variação do número de Tomcats empregados e qual a relação entre o tempo gasto acessando o driver escravo com as demais atividades realizadas pelo driver, em especial, o acesso às caches distribuídas.

A tabela 3.13 mostra o tempo total de execução do driver dCache particionado em três categorias: tempo gasto com acesso ao driver escravo, tempo gasto com acesso às caches distribuídas e tempo gasto com outras atividades quaisquer. O tempo total de execução do driver corresponde ao somatório dos tempo gastos em todos o métodos apresentados na tabela 3.8. O tempo gasto com acesso ao driver escravo, corresponde não só aos acessos realizados pelo método `next()` e sim, a todos os acessos realizados contando com o próprio `next()` e outros métodos. Os tempo gasto com acessos à caches distribuídas leva em conta o tempo gasto com acessos a cache de relações e de tuplas. São mostrados o tempo total do ponto de vista de uma única instância e o tempo total global. O tempo total global corresponde ao somatório dos tempos gastos individualmente por cada instância do driver.

Particionamento do tempo dCache	Tempo Total Por Instância (ms)			Tempo Total Global (ms)		
	1 Tomcat	2 Tomcats	3 Tomcats	1 Tomcat	2 Tomcats	3 Tomcats
Acesso ao driver escravo	162735	101468,5	75893,00	162735	202937	227679
Acesso às caches distribuídas	2973	80556,0	103289,33	2973	161112	309868
Outros	6372	4505,0	3167,33	6372	9010	9502
Total	172080	186529,5	182349,67	172080	373059	547049

Tabela 3.13: Particionamento do tempo total gasto pelos métodos do driver dCache, para a configuração tpcw-10K-25, com o perfil BM.

Os gráficos da figura 3.15 (baseados na tabela 3.13) mostram a contribuição de cada partição no tempo total gasto por uma instância do driver dCache (gráfico *a*) e a contribuição de cada partição no tempo total gasto por todas as instâncias (gráfico *b*). Observando ambos os gráficos verificamos que a participação relativa do tempo de acesso às caches distribuídas tanto no tempo total gasto por uma instância, quanto no tempo total gasto pelas instâncias, cresce expressivamente com o aumento do número de servidores de aplicação.

De forma complementar, na tabela 3.14 é apresentada a variação do médio gasto

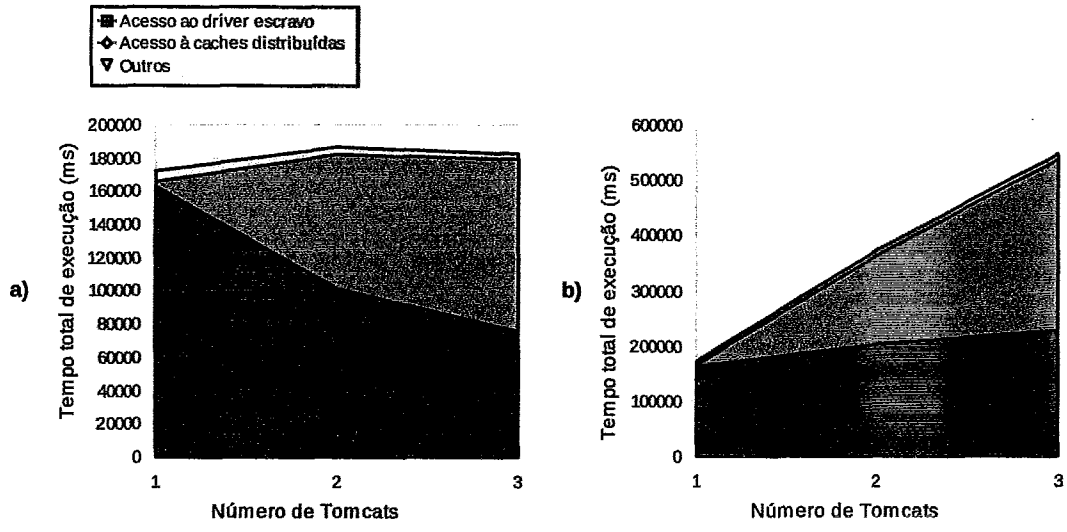


Figura 3.15: Contribuição de cada partição no tempo total de execução do dCache, para o perfil BM, configuração tpcw-10K-25.

para realização de uma atividade de cada partição. No gráfico 3.16, que apresenta os dados da tabela 3.14, podemos observar que o tempo médio de acesso às caches cresce muito mais acentuadamente do que o tempo de acesso ao driver escravo.

Particionamento do tempo dCache	Tempo médio de execução (ms)		
	1 Tomcat	2 Tomcats	3 Tomcats
Acesso ao driver escravo	0,3704	0,4463	0,5033
Acesso às caches distribuídas	0,0103	0,5561	1,0778
Outros	0,0044	0,0063	0,0066

Tabela 3.14: Particionamento dos tempos médios gastos pelo driver dCache, para a configuração tpcw-10K-25, com o perfil BM.

Com base nestas observações e, dado que as caches distribuídas não são necessárias para o funcionamento do driver distribuído, uma interessante linha de investigação seria a implementação e avaliação de uma versão da arquitetura do driver dCache onde não são empregadas caches distribuídas (poderiam ser utilizadas caches locais, como na arquitetura centralizada). Conforme discutido no capítulo 2, dos novos conceitos incorporados ao driver dCache pela arquitetura distribuída proposta por este trabalho, apenas três são necessários para a corretude do driver. São eles: a tabela de modificações globais distribuída, a sincronização distribuída e o relógio global distribuído. Mesmo já tendo sido medido o significativo impacto negativo do acesso remoto às caches distribuídas sobre o desempenho do sistema em teste, registremos agora, como a implementação destes três conceitos impactaram nos resultados obtidos.

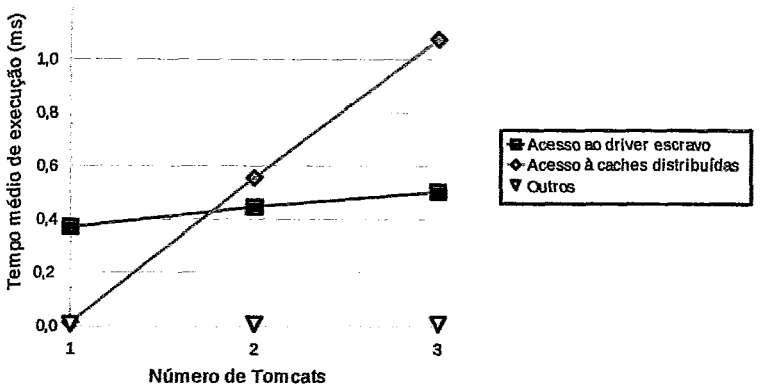


Figura 3.16: Variação dos tempos médios de cada partição para a configuração tpcw-10K-25, com o perfil BM, em função do número de Tomcats.

Quando falamos de sincronização distribuída no contexto da versão distribuída do driver utilizada na presente avaliação experimental, estamos nos referindo ao mecanismo de *locks* distribuídos empregados para garantir a realização do processo de *commit* das transações em regime de exclusão mútua[11]. Comecemos a análise dos três conceitos pelo método *commit()*, que, além de ser o único a fazer uso de um *lock* distribuído na sua implementação, realiza também acessos a tabela de modificações globais e solicita rótulos de tempo ao relógio global.

São dois os fatores que influenciam no tempo gasto por uma instância para obter um *lock* distribuído: a disponibilidade do *lock* solicitado e o tempo de resposta do servidor dCache ao pedido de aquisição do *lock*. O primeiro fator se refere ao tempo que uma instância que precisa adquirir um *lock* é obrigada a aguardar uma vez que alguma outra instância já está de posse do *lock* solicitado. O segundo fator se refere ao tempo de comunicação com o servidor dCache: é o tempo gasto para realizar o pedido e aguardar a sua resposta, mesmo que o *lock* desejado esteja imediatamente disponível. Para liberar um *lock* previamente adquirido, só existe o fator tempo de resposta do servidor, uma vez que não é necessário aguardar a disponibilidade do *lock*.

A disponibilidade de um *lock* está diretamente relacionada com o tempo gasto dentro da região crítica a qual ele protege. Ora, quanto maior o tempo gasto dentro da região crítica, maior o tempo que o *lock* está indisponível. Uma vez que *commit* do dri-

[11] Vimos, na seção 2.1.7, que exclusão mútua também poderia ser imposta entre os códigos de *commit* e a leitura da tabela de modificações globais para proporcionar ao driver dCache o suporte a mais um nível de isolamento entre transações. No entanto, vimos nas seções 2.2.3 e 2.4.4 que as implementações centralizada e distribuída do driver optaram por garantir exclusão mútua apenas entre *commits* realizados concorrentemente e em paralelo.

ver dCache (a região crítica do *lock* utilizado pelo dCache) consiste na obtenção de um rótulo de tempo e no acesso à tabela de modificações globais, a disponibilidade deste *lock* está diretamente relacionada ao tempo gasto com essas duas atividades. Adicionalmente, como no caso do dCache estamos tratando de um *lock distribuído*, podemos acrescentar nesta equação o tempo de liberação do *lock*. Ou seja, como a liberação de um *lock* distribuído pode levar um tempo significativo (devido à comunicação com o servidor), este tempo também deve ser levado em consideração na disponibilidade do *lock*.

O tempo total (do ponto de vista de uma instância do driver) e o tempo médio de execução de cada uma das atividades realizadas pela implementação do método *commit()* são mostrados na tabela 3.15. O tempo que o *lock* utilizado pelo dCache fica indisponível corresponde a soma dos tempos das atividades "Obtenção de rótulo de tempo", "Acesso a tabela de modificações globais" e "Unlocking". Ou seja, 531 milissegundos quando apenas um servidor Tomcat é utilizado, 647 milissegundos quando são usados dois Tomcats e 465 milissegundos quando três Tomcats são empregados.

Nome das atividades do método *commit()*	Tempo total de execução (ms)			Tempo médio de execução (ms)		
	1 Tomcat	2 Tomcats	3 Tomcats	1 Tomcat	2 Tomcats	3 Tomcats
Locking	337,00	148,00	93,00	1,1502	0,9834	0,9058
Obtenção de rótulo de tempo	279,00	89,50	54,67	0,9522	0,5947	0,5325
Acesso à tabela de modificações globais	25,00	422,50	306,00	0,0853	2,8073	2,9805
Unlocking	227,00	135,00	104,33	0,7747	0,8970	1,0162
commit() driver escravo	4095,00	2450,00	1863,00	0,1436	0,1726	0,1979

Tabela 3.15: Tempo total e médio gasto em cada execução das atividades do método `commit()`, para a configuração tpcw-10K-25, com o perfil BM.

Por esta ótica, poderíamos esperar que o tempo médio para que uma instância adquira este *lock* deveria ser maior quando são utilizados dois Tomcats, deveria ser menor quando é empregado apenas um Tomcat e deveria ser menor ainda quando são usados três Tomcats. Porém, observando o tempo médio da atividade *Unlocking* notamos que não é isso que acontece. Na realidade, o tempo médio para obtenção do *lock* diminui à medida que são empregados mais Tomcats na execução do benchmark.

Dissemos que o outro fator que influencia no tempo gasto para obter o *lock* é a disponibilidade do servidor. Nos experimentos em questão, a ocupação da CPU na máquina que roda o servidor dCache é inferior a 1%, independentemente da quantidade de servidores de aplicação empregados, não sendo, portanto, um critério que justifique variações no tempo gasto para adquirir ou liberar um *lock*. Por outro lado, sabemos que, durante a execução do benchmark, o servidor dCache tem duas responsabilidades: fornecer rótulos de tempo e gerenciar o *lock* distribuído. Os pedidos de rótulos de tempo

são realizados pela implementação do método `executeQuery()` (ver tabela 3.10) e pelo método `commit()`. Os pedidos relacionados aos rótulos de tempo são realizados exclusivamente pelo método `commit()`.

A tabela 3.16 mostra a quantidade de vezes que são realizadas solicitações de *lock*, liberação de *lock* e pedidos de rótulos de tempo de acordo com o número de Tomcats utilizados. As quantidades mostradas correspondem ao somatório dos pedidos realizados por todas as instâncias do driver utilizadas na execução do benchmark. Os pedidos de rótulo de tempo são separados em pedidos feitos pelo método `commit()` e pelo método `executeQuery()`. É interessante notar que, com o aumento do número de Tomcats, a quantidade de vezes que pedidos de rótulos de tempo são realizados aumenta, assim como os pedidos de aquisição e liberação de *lock*. Ou seja, com o aumento do número de Tomcats, o servidor dCache precisa tratar um maior número de mensagens, ficando mais sobrecarregado.

Nome das atividades	Total de Execuções		
	1 Tomcat	2 Tomcats	3 Tomcats
Locking	293	301	308
Unlocking	293	301	308
Obtenção de rótulo de tempo (*commit*)	293	301	308
Obtenção de rótulo de tempo (*executeQuery*)	471	678	1042

Tabela 3.16: Variação, com a quantidade de Tomcats, da quantidade de pedidos de *lock*, liberações de *lock* e pedidos de rótulo de tempo para a configuração tpcw-10K-25, com o perfil BM.

Esta observação levaria à expectativa de que os tempos médios para adquirir e liberar o *lock* deveriam aumentar com o número de servidores de aplicação utilizados, bem como o tempo médio para obter um rótulo de tempo. Observando a tabela 3.15, vemos que essa expectativa se confirma para o tempo médio de liberação do *lock* (atividade *Unlocking*), mas não se confirma para aquisição do *lock* nem para a obtenção de rótulos de tempo. Em trabalhos futuros poderão ser realizados estudos adicionais para entender o comportamento não intuitivo da variação do tempo médio para aquisição do *lock* e para obtenção de rótulos de tempo.

Para finalizar a análise dos resultados obtidos com a configuração tpcw-10K-25, perfil BM, vale a pena mencionar a variação no tempo de acesso à tabela de modificações globais à medida que são utilizados mais Tomcats para a execução do experimento. O único ponto no qual ocorre escrita na tabela de modificações globais, é durante a execução do método `commit()`, onde também só ocorre acesso de escrita. A tabela 3.15 mostra que o tempo médio de acesso à esta estrutura aumenta em função do número de servidores

de aplicação utilizados. Observamos que o aumento é mais significativo na passagem de um para dois servidores de aplicação. Talvez, a explicação para este aumento mais significativo seja a utilização de alguma otimização (não documentada em [57]) para o caso onde existe apenas um membro conectado ao canal de comunicação JGroups (ver apêndice C).

Já o acesso "somente leitura" à tabela de modificações globais é realizado na implementação do método `executeQuery()` (ver tabela 3.10) e na implementação do método `getValue()`. Duas observações importantes devem ser feitas sobre o tempo de acesso somente leitura: A primeira é que o tempo de acesso permanece praticamente constante (em torno de 0,0016 milissegundos) independentemente do número de servidores de aplicação usado. A segunda é que o tempo de acesso somente leitura é bem inferior do que, até mesmo, o menor tempo de escrita reportado. As duas observações são explicadas pelo fato de que, na implementação da tabela hash distribuída fornecida pelo JGroups, todos acessos de leitura são realizados em estrutura de dados local, enquanto as escritas são propagadas para todos os membros [70].

Configurações tpcw-100K-300 e tpcw-1M-50

Comparando os dados apresentados pelos gráficos *a* e *b* da figura 3.10, notamos que o uso, na execução do benchmark, de 100 navegadores emulados faz com que o sistema em teste proporcione maior quantidade de interações Web por minuto do que quando são empregados apenas 25. Com exceção do perfil OM com 2 e 3 servidores de aplicação, essa observação é válida qualquer que seja o perfil de navegação utilizado, qualquer que seja o número de servidores de aplicação utilizado.

Por outro lado, comparando os resultados apresentados pelo gráfico *b* da figura 3.10 com os resultados apresentados pelo gráfico *b* da figura 3.3, notamos que o número de interações Web por minuto proporcionadas pelo sistema em teste cai abruptamente quando o número de navegadores emulados sobe de 100 para 300, e a base de dados se torna 10 vezes maior. Queda semelhante ocorre quando são utilizados apenas 50 navegadores emulados, porém, com o uso de uma base de dados 100 vezes maior, conforme pode ser observado pelo gráfico *b* da figura 3.6[12].

[12]É importante observar que o intervalo de medição (60 minutos) utilizado para os experimentos com as configurações tpcw-10K-25 e tpcw-10K-100 é diferente do intervalo utilizado para as configurações tpcw-100K-300 e tpcw-1M-50 (30 minutos). Entretanto, a diferença na quantidade de interações Web conseguidas é tão grande que mesmo dividindo por dois a quantidade de interações obtidas com o intervalo de medição maior, ainda assim, a queda na quantidade de interações Web proporcionada é significativa.

A queda no número de interações Web proporcionadas e o aumento na ocupação da CPU do servidor de banco de dados demonstram que, ao contrário do perfil tpcw-10K-25, os perfis tpcw-100K-300 e tpcw-1M-50 saturam o sistema em teste quando é empregado apenas um servidor de aplicação. Para comprovar esta saturação, é interessante observar que, enquanto nas configurações tpcw-10K-25 e tpcw-10K-100 a CPU do servidor de banco de dados alcança o máximo de, aproximadamente, 5% de taxa de ocupação, com as configurações tpcw-100K-300 e tpcw-1M-50 essa taxa sobe para, aproximadamente, 99% de uso. Uma exceção é o perfil BM na configuração tpcw-1M-50, onde a CPU permanece 95% do tempo esperando por operações de *I/O*.

Ficamos, então, interessados em verificar qual seria o comportamento do sistema em teste quando utilizássemos a arquitetura distribuída do driver dCache para permitir o emprego de mais servidores de aplicação em execuções do benchmark utilizando estas configurações mais demandantes. Para isso, realizamos mais 12 medições através do benchmark TPC-W, 6 com a configuração tpcw-100K-300 e 6 com a configuração tpcw-1M-50. Para cada configuração, testamos os três perfis de navegação e utilizamos 2 e 3 Tomcats. As medições realizadas com a configuração tpcw-100K-300 foram precedidas por um período de aquecimento de 1 hora, enquanto as realizadas com a configuração tpcw-1M-50 foram precedidas por um período de 3 horas. Com ambas as configurações, o intervalo de medição utilizado foi de 30 minutos.

A tabela 3.17 *a* mostra a quantidade de interações Web conseguidas com o perfil tpcw-100K-300. A tabela 3.17 *b* mostra a quantidade de interações Web conseguidas com o perfil tpcw-1M-50. Em ambas as tabelas, a primeira coluna (1 Tomcat) refere-se a quantidade de interações Web obtidas com a versão centralizado do driver. Os valores dessas colunas já haviam sido mostrados nos gráficos *b* da figura 3.3 e *b* da figura 3.6, mas foram repetidos aqui para facilitar a comparação.

a) tpcw-100K-300

	WIPM		
	1 Tomcat	2 Tomcats	3 Tomcats
BM	1,3094	1,3576	1,2601
SM	0,9194	0,8764	0,8651
OM	1,1187	1,2094	1,1723

b) tpcw-1M-50

	WIPM		
	1 Tomcat	2 Tomcats	3 Tomcats
BM	0,4103	0,2235	0,1419
SM	0,3376	0,3173	0,3139
OM	0,4601	0,4818	0,4537

Tabela 3.17: Quantidade de interações Web proporcionadas pelas configurações tpcw-100K-300 e tpcw-1M-50 com o uso do driver dCache.

O gráfico *a* da figura 3.17 apresenta os valores da tabela 3.17 *a* e o gráfico *b* da mesma figura apresenta os valores da tabela 3.17 *b*. Observando os gráfico *a* e *b*, podemos

observar que, tanto para a configuração tpcw-100K-300, quanto para a configuração 1M-50, o emprego de 2 servidores de aplicação sempre proporciona maior quantidade de WIPM do que o emprego de 3 servidores de aplicação. Na configuração tpcw-100K-300, o emprego de 2 servidores de aplicação proporciona melhor desempenho do que o emprego de apenas um servidor de aplicação nos perfis BM e OM. Na configuração tpcw-1M-50, o uso de 2 servidores de aplicação proporciona melhor desempenho do que o emprego de apenas um servidor de aplicação apenas no perfil OM.

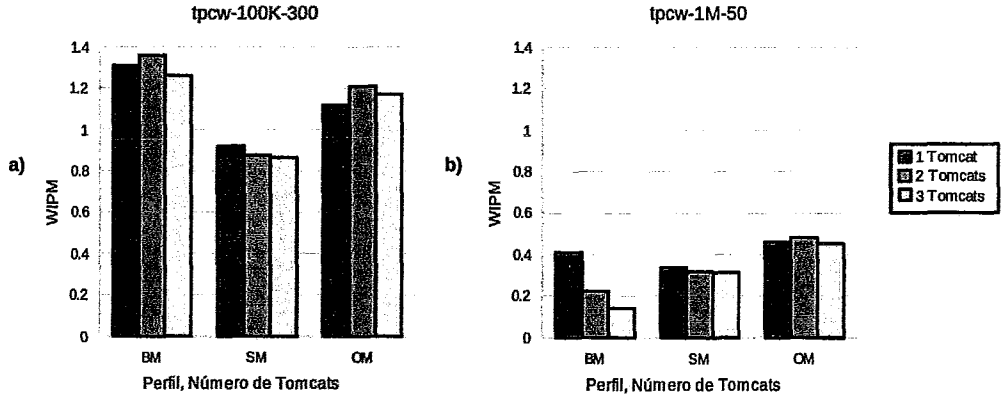


Figura 3.17: Variação, com o número de Tomcats e perfis, das WIPM conseguidas com as configurações tpcw-100K-300 e tpcw-1M-50.

Verificamos então, que, ao contrário das configurações tpcw-10K-25 e tpcw-10K-100, nas configurações tpcw-100K-300 e tpcw-1M-50 o uso da arquitetura distribuída do driver proporciona melhor desempenho em alguns perfis de navegação. Com a configuração tpcw-100K-300, o uso da arquitetura distribuída proporciona 3,68% mais interações Web no perfil BM e 8,1% mais interações no perfil OM. Com a configuração tpcw-1M-50, a arquitetura distribuída proporciona 4,72% mais interação no perfil OM.

Além disso, também é importante observar que, ao contrário das configurações tpcw-10K-25 e tpcw-10K-100, nas configurações tpcw-100K-300 e tpcw-1M-50 a diferença entre a quantidade de interações Web proporcionadas pelo emprego de apenas um Tomcat e quantidade de interações Web proporcionadas pelo emprego de maior número de Tomcats é mais sutil. Essa variação, em conjunto com os ganhos de desempenho já alcançados pela arquitetura distribuída, indica que, apesar do gargalo representado pelo uso de caches distribuídas (discutido na seção 3.4.4), a arquitetura distribuída do driver dCache tem potencial para proporcionar ganhos de performance em sistemas de comércio eletrônico onde o número de acessos e o tamanho da base de dados acessada são grandes o suficiente para saturar o sistema quando é empregado apenas um servidor de aplicação.

Capítulo 4

Trabalhos Futuros

Apresentaremos neste capítulo alguns trabalhos que consideramos relevantes para a continuidade do estudo do cacheamento consistente, transparente e distribuído de banco de dados através do driver dCache. Inicialmente, sugeriremos alguns trabalhos com objetivo de aprimorar as caches distribuídas utilizadas na arquitetura distribuída do driver dCache. A motivação para estes trabalhos é a constatação (seção 3.4.4) de que o tempo gasto por estas caches com acessos remotos constitui um grande gargalo no desempenho da arquitetura distribuída do driver. Esperamos então, que pequenas melhorias neste componente da arquitetura distribuída proporcionarão melhorias de desempenho significativas nos sistemas que utilizam mais de uma instância do driver dCache.

Em seguida, falaremos de outros trabalhos que, apesar de não tratarem diretamente do gargalo representado pelas caches distribuídas, também apresentam propostas que podem melhorar o desempenho da versão distribuída do driver.

4.1 Aprimorando as caches distribuídas

4.1.1 Algoritmos alternativos de comunicação entre as caches

Durante a descrição da arquitetura distribuída do driver dCache (seção 2.4.3), definimos um algoritmo a ser aplicado na comunicação entre as caches distribuídas. Com base nos resultados experimentais obtidos, seria interessante a proposição e avaliação de algoritmos alternativos. Por exemplo: Vimos que as instâncias do driver gastam muito tempo realizando consultas a caches remotas no momento em que ocorre uma falta no acesso a sua cache local. Na implementação atual, o acesso às caches remotas é disparado em

resposta à ocorrência de uma falta no acesso à cache local. Uma estratégia interessante seria não realizar o acesso à cache remota no momento da falta. Ao invés disso, cada cache poderia, pró-ativamente, tomar a iniciativa de publicar para as demais caches os dados trazidos por ela do banco de dados.

4.1.2 Implementação alternativa de acesso às caches remotas

Na implementação das caches distribuídas utilizadas pela arquitetura distribuída (seção 2.4.3), optamos por, dada uma falta em um acesso à cache local, realizar acessos as caches remotas seqüencialmente, percorrendo uma lista de endereços de instâncias do driver dCache. Com os resultados obtidos na avaliação experimental, notamos que esta estratégia faz com que o tempo médio gasto pelas instâncias com acessos às caches aumente linearmente com o aumento do número de instâncias do driver empregadas no experimento. Uma alternativa a esta implementação seria a utilização de *threads* auxiliares para realizar a consulta às caches remotas concorrentemente.

4.1.3 Instâncias hospedeiras de tabelas do banco de dados

Outra altenativa para diminuição no tempo gasto acessando caches remotas seria a criação de instâncias hospedeiras de um conjunto de tabelas do banco de dados. Na arquitetura distribuída apresentada, todas as instâncias do driver podem armazenar em suas caches dados relacionados a consultas sobre quaisquer tabelas do banco de dados.

Com a estratégia das "instâncias hospedeiras", cada instância do driver se responsabilizaria por um subconjunto das tabelas do banco de dados. Quando uma instância do driver executasse uma consulta que resultasse em falta no acesso à cache, essa instância acessaria a cache remota responsável por armazenar os dados das tabelas envolvidas na consulta que ela executara. Assim, apenas uma cache remota seria acessada, ao invés do que é feito pela implementação atual, que acessa todas as caches.

Apesar da estratégia acessar apenas uma cache remota, é esperado que a taxa de acertos a essa cache seja alta, uma vez que ela concentra dados relacionados a apenas um subconjunto das tabelas do banco. A idéia de particionar a cache em regiões com maior taxa de acertos não é nova, tendo sido já utilizada, em [29], por exemplo.

4.1.4 Importação conjunta de relações e tuplas

Vimos que a arquitetura do driver dCache prevê dois tipos de relações: relações de tuplas e relações de chaves para tuplas (seção 2.1.1). Na implementação da arquitetura distribuída apresentada, ao ocorrer uma falta no acesso a cache de relações, a instância do driver dCache consulta as caches de relações remotas, a procura da relação que não pôde ser encontrada localmente. Se a relação é encontrada em alguma cache remota, é importada para a cache local da instância, e passa a ser utilizada pelo driver como se fosse oriunda da cache local, transparentemente (seção 2.4.3).

Se a relação importada fosse uma relação de tuplas, todas as suas tuplas também teriam sido importadas (graças ao processo de serialização da relação) e também passaram a estar disponíveis localmente. Se a relação é uma relação de chaves para tuplas, as chaves para as tuplas também terão sido importadas e passaram a estar disponíveis localmente. Porém, as tuplas às quais tais chaves se referem, não terão sido importadas da instância remota do driver dCache. Se a cache de tuplas local contém, coincidentemente, algumas das tuplas referenciadas, a tentativa de uso destas chaves para tuplas resultará em acerto na cache de tuplas. Por outro lado, se esta concidência não ocorre, a tentativa de uso destas chaves resultará em faltas no acesso a cache de tuplas.

Como as caches de tuplas também são distribuídas, o acesso remoto acabará transformando as faltas em acertos remotos, porém ao custo adicional da comunicação com as caches distribuídas. Para evitar este custo extra, uma alternativa seria, ao importar uma relação de chaves para tuplas, automaticamente importar também todas as tuplas associadas. Um sugestão de implementação seria converter a relação de chaves em uma relação de tuplas antes de realizar a importação.

4.2 Outros trabalhos

4.2.1 Comparação de alternaticas de *lock* para o *commit*

Enquanto na implementação da arquitetura centralizada o núcleo de processamento do driver dCache trava colunas e tabelas individualmente para aumentar o grau de paralelismo entre as transações que realizam *commit*, na arquitetura distribuída, é utilizado apenas um *lock* para garantir exclusão mútua no processo de *commit* (seção 2.4.2).

Alguns poucos experimentos mostraram que a opção por um único *lock* na implementação da arquitetura distribuída é vantajosa, porém estes experimentos utilizaram

uma versão da implementação do driver bastante diferente da versão final, utilizada nos experimentos apresentados no capítulo 3. Achamos importante a realização de mais experimentos, utilizando a versão mais recente do driver, para confirmar a hipótese de que o uso de apenas um *lock* na arquitetura distribuída é mais vantajoso do que o maior paralelimo esperado pelo uso de *locks* individuais para cada tabela e coluna.

4.2.2 Utilização de implementação distribuída de cache para comandos SQL

Durante a implementação da arquitetura distribuída do driver dCache, optamos por utilizar caches distribuídas apenas na implementação da cache de relações e da cache tuplas. Fizemos essa opção com a expectativa de que o tempo de *parsing* de um comando SQL seria menor do que o tempo de acesso às caches remotas, não justificando o seu emprego (seção 2.4.3). Entretanto, os experimentos realizados mostraram que essa expectativa não se concretizou e, portanto, o uso da cache distribuída também para os comandos SQL pode ser vantajosa. A utilização das caches distribuídas também para comandos SQL certamente será irrelevante para o desempenho do driver medido através do benchmark TPC-W, pois este benchmark usa um conjunto pequeno de comandos SQL. Entretanto, essa modificação pode ser relevante para aplicações com uma grande variabilidade de comandos SQL.

4.2.3 Avaliação da sobrecarga de outros níveis de isolamento

Nas seções 2.1.8, 2.2.3 e 2.4.4 relatamos várias limitações do driver dCache. Dentre essas limitações, mostramos que apenas o nível de isolamento *READ UNCOMMITTED* do padrão ANSI é suportado pelo driver dCache. Embora essa limitação facilite a implementação do driver, ela restringe muito a sua aplicabilidade, tornando-o útil apenas para as poucas aplicações que não fazem praticamente nenhuma exigência sobre o isolamento de transações. Consideramos importante o estudo da sobrecarga que seria adicionada ao driver quando níveis de isolamento mais restritivos fossem implementados no driver. Dentre os pontos a serem analisados, a sobrecarga imposta pelo uso da sincronização distribuída na implementação dos novos níveis de isolamento na arquitetura distribuída do driver, seria de especial interesse.

4.2.4 Exploração de caches semânticas

Conforme definido no capítulo 1, o sistema dCache é um sistema de cache de consultas para banco de dados. Sendo um sistema de cache de consultas, a reutilização de relações armazenadas na cache, em detrimento da busca – diretamente no banco de dados – dos dados solicitados pela aplicação cliente, depende da repetição de consultas anteriormente executadas. Mais do que isso, não só a consulta precisa se repetir, mas também todos os argumentos nela utilizados. Assim, a consulta `C1 = SELECT * FROM PESSOA WHERE IDADE < 18` não poderia reutilizar os dados a relação em cache gerada pela consulta `C2 = SELECT * FROM PESSOA WHERE IDADE < 21` pois os argumentos `18` e `21` são diferentes.

Entretando, analisando as consultas `C1` e `C2`, é fácil perceber que os dados de interesse de `C1` são um subconjunto dos dados de interesse de `C2`. Ou seja, teoricamente, poderíamos compor a relação buscada por `C2` a partir do resultado de `C1` acrescido de algumas tuplas adicionais, que poderiam, por exemplo, serem obtidas pela consulta auxiliar `SELECT * FROM PESSOA WHERE IDADE >=18 AND IDADE < 21`. Essa idéia reflete o princípio por traz das *caches semânticas* exploradas em [71], por exemplo.

Consideramos interessante a avaliação dos benefícios que poderiam ser trazidos para o driver dCache por meio da utilização desta técnica. Questões importantes a serem respondidas seriam como manter a generalidade do driver e a sua transparência para aplicação neste contexto.

Capítulo 5

Conclusão

O presente trabalho realizou modificações na arquitetura do driver dCache proposta por [40], gerando uma nova arquitetura centralizada para o driver. As modificações consistiram no aprimoramento da estrutura de caches do driver e no aprimoramento no mecanismo de invalidações, com o objetivo de proporcionar tratamento consistente de comandos SQL do tipo UPDATE.

Repetindo os experimentos realizados por [40] e comparando os resultados obtidos, mostramos que a implementação da arquitetura centralizada do driver, acrescida das modificações descritas por este trabalho, aumentou os ganhos de desempenho proporcionados pelo uso do driver dCache em um servidor de comércio eletrônico dotado de apenas um servidor de aplicação, em todas as configurações e perfis de navegação avaliados. Os resultados obtidos são um forte indício de que a nova estrutura de caches implementada colabora positivamente no desempenho do driver, sobrepujando a perda de desempenho que poderia ter sido proporcionada devido ao tratamento das invalidações decorrentes de UPDATES, que também passou a ser realizada pela nova implementação.

O ambiente experimental utilizado pelo presente trabalho possuiu varias diferenças com relação ao ambiente experimental utilizado em [40]. Muitos índices utilizados em [40] para a avaliação do driver eram baseados no comportamento das caches, no entanto a estrutura das caches foi alterada pelo presente trabalho. Esses dois fatores dificultaram a comparação dos resultados obtidos pelos dois trabalhos e sugerem que novas avaliações sejam realizadas, em um ambiente mais bem controlado, para que as conclusões obtidas sejam mais confiáveis.

Propusemos modificações na nova arquitetura centralizada do driver dCache gerando uma arquitetura distribuída. As principais modificações efetuadas foram a transfor-

mação do gerador de rótulos de tempo em um relógio global distribuído, emprego de um mecanismo de *locks* distribuídos para garantir a consistência entre as instâncias e o emprego de estruturas de dados distribuídas nas caches e na tabela de modificações globais. Implementamos a arquitetura distribuída proposta e avaliamos-na experimentalmente.

Os resultados obtidos pela avaliação da versão distribuída do driver quando submetida a uma carga de trabalho gerada por poucos navegadores acessando uma base de dados pequena mostraram que o uso do driver dCache piora o desempenho do sistema em teste e que esta degradação é agravada conforme é aumentado o número de servidores de aplicação utilizados. A instrumentação do driver mostrou que apesar da utilização de caches distribuídas conseguir reverter parte das faltas no acesso às caches em acertos, o tempo de acesso a essas caches aumenta significativamente o tempo de total gasto com a execução do driver, sendo este o principal motivo da redução de desempenho com o aumento do número de servidores de aplicação utilizados.

Já os resultados obtidos pela avaliação da versão distribuída do driver quando submetida a uma carga de trabalho gerada por um número maior de navegadores acessando uma base de dados maior, mostraram que a utilização de mais servidores de aplicação pode proporcionar melhor desempenho do que o uso de apenas um servidor de aplicação. Os resultados obtidos por esta carga de trabalho mostraram que a versão distribuída do driver, baseada na replicação dos servidores de aplicação utilizados, pode ser uma alternativa viável para a escalabilidade de servidores de comércio eletrônico.

Com base na observação de que as caches distribuídas constituem um gargalo para o desempenho da versão distribuída do driver, trabalhos futuros foram propostos com o objetivo de aprimorar o funcionamento destas caches. Fora estes trabalhos, outros também foram propostos com o objetivo de melhorar o desempenho do driver e conhecer melhor suas limitações. Entre eles, estão a exploração de caches semânticas e exploração de níveis de isolamento ainda não implementados no driver dCache.

Tanto a nova arquitetura centralizada do driver dCache, quando a nova arquitetura distribuída proposta, implementada e avaliada por este trabalho preservam as características desejáveis presentes na arquitetura original apresentada em [40]. Ou seja, as novas arquiteturas mantém o driver transparente para as aplicações cliente e para o banco de dados, são consistentes e genéricas o suficiente para serem empregadas em qualquer aplicação Java que acesse o banco de dados através de um driver JDBC.

Referências Bibliográficas

[1] FILIPPO, D., SZTAJNBERG, A., *Bem-vindo à Internet*. Brasport: Rio de Janeiro, 1996.

[2] JAZAYERI, M., "Some Trends in Web Application Development". In: *FOSE '07: 2007 Future of Software Engineering*, pp. 199–213, IEEE Computer Society: Washington, DC, USA, 2007.

[3] "Netcraft Web site", `http://news.netcraft.com/`; acessado em 4 de maio de 2008.

[4] "Internet Systems Consortium", `http://www.isc.org/`; acessado em 4 de maio de 2008.

[5] MARIUCCI, M., "Web Technologies - Overview", october 2000.

[6] TITCHKOSKY, L., ARLITT, M., WILLIAMSON, C., "A Performance Comparison of Dynamic Web Technologies", *SIGMETRICS Perform. Eval. Rev.*, v. 31, n. 3, pp. 2–11, 2003.

[7] ARLITT, M., KRISHNAMURTHY, D., ROLIA, J., "Characterizing the Scalability of a Large Web-Based Shopping System", *ACM Trans. Interet Technol.*, v. 1, n. 1, pp. 44–69, 2001.

[8] REICHHELD, F. F., SASSER, W. E., "Zero defections: Quality Comes to Services", *Harvard Business Review September-October 1990*, September 1990.

[9] CARDELLINI, V., CASALICCHIO, E., COLAJANNI, M., etal., "The State of the Art in Locally Distributed Web-Server Systems", *ACM Comput. Surv.*, v. 34, n. 2, pp. 263–311, 2002.

[10] "The Apache Web Server", `http://httpd.apache.org/`; acessado em 14 de abril de 2008.

[11] "The Jetty Web server", `http://mortbay.org/`; acessado em 14 de abril de 2008.

[12] "The Resin Web Server", `http://www.caucho.com/resin/`; acessado em 10 de maio de 2008.

[13] "The Tomcat Servlet Container", `http://tomcat.apache.org`; acessado em 14 de abril de 2008.

[14] "The JBoss Application Server", `http://labs.jboss.com/`; acessado em 14 de abril de 2008.

[15] "IBM Inc., The Web Sphere server", `http://www.ibm.com/software/webservers/appserv/was/`; acessado em 10 de maio de 2008.

[16] "The JOnAS Application Server", `http://wiki.jonas.objectweb.org/`; acessado em 10 de maio de 2008.

[17] "BEA WebLogic Platform", `http://www.bea.com/software/weblogic/platform`; acessado em 10 de maio de 2008.

[18] BANGA, G., MOGUL, J. C., "Scalable Kernel Performance for Internet Servers under Realistic Loads". In: *Proceedings of the 1998 USENIX Annual Technical Conference*, New Orleans, LA, 1998.

[19] HU, Y., NANDA, A., YANG, Q., "Measurement, Analysis and Performance Improvement of the Apache Web Server", 1997.

[20] NAHUM, E., BARZILAI, T., KANDLUR, D. D., "Performance Issues in WWW Servers", *IEEE/ACM Trans. Netw.*, v. 10, n. 1, pp. 2–11, 2002.

[21] PAI, V. S., DRUSCHEL, P., ZWAENEPOEL, W., "Flash: An Efficient and Portable Web Server". In: *Proceedings of the USENIX 1999 Annual Technical Conference*, 1999.

[22] DEVLIN, B., GRAY, J., LAING, B., etal., *Scalability Terminology: Farms, Clones, Partitions, and Packs: RACS and RAPS*, Tech. rep., Microsoft, December 1999.

[23] PAI, V. S., ARON, M., BANGA, G., etal., "Locality-Aware Request Distribution in Cluster-based Network Servers". In: *Architectural Support for Programming Languages and Operating Systems*, pp. 205–216, 1998.

[24] "The Java Tutorials - Trail: JDBC Database Access", http://java.sun.com/docs/books/tutorial/jdbc/; acessado em 29 de abril de 2008.

[25] "The Java JDBC Home Page", http://java.sun.com/products/jdbc/overview.html; acessado em 30 de maio de 2008.

[26] ZHANG, Q., RISKA, A., RIEDEL, E., etal., "Bottlenecks and their Performance Implications in E-Commerce Systems". In: *Proceedings of Ninth International Workshop on Web Content Caching and Distribution (WCW2004)*, pp. 273–282, Beijing, China, October 2004.

[27] "MySQL Cluster", http://www.mysql.com/products/database/cluster/; acessado em 8 de junho de 2008.

[28] DOEDERLEIN, O. P., "Persistência Turbinada", *Java Magazine*, pp. 28–41, junho 2005.

[29] DIAS, S., *Técnicas Baseadas em Memória Compartilhada e Passagem de Mensagens para Servidores de Aplicações de Comércio Eletrônico com Qualidade de Serviço*, Master's Thesis, Universidade Federal do Rio de Janeiro, COPPE, 2003.

[30] "Hibernate", http://www.hibernate.org/; acessado em 15 de maio de 2008.

[31] "EHCache", http://ehcache.sourceforge.net/; acessado em 15 de maio de 2008.

[32] "Open Terracotta", http://www.terracotta.org/; acessado em 15 de maio de 2008.

[33] "Open Symphony OSCache", http://www.opensymphony.com/oscache/; acessado em 15 de maio de 2008.

[34] "Swarm Cache", http://swarmcache.sourceforge.net/; acessado em 15 de maio de 2008.

[35] "JBoss Cache", http://www.jboss.org/jbosscache/; acessado em 15 de maio de 2008.

[36] CECCHET, E., MARGUERITE, J., ZWAENEPOLE, W., "C-JDBC: flexible database clustering middleware". In: *ATEC '04: Proceedings of the annual conference on USENIX Annual Technical Conference*, pp. 26–26, USENIX Association: Berkeley, CA, USA, 2004.

[37] LUO, Q., KRISHNAMURTHY, S., MOHAN, C., etal., "Middle-tier Database Caching for e-Business". In: *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pp. 600–611, ACM: New York, NY, USA, 2002.

[38] CUENCA-ACUNA, F. M., NGUYEN, T. D., "Cooperative Caching Middleware for Cluster-Based Servers". In: *Tenth IEEE International Symposium on High Performance Distributed Computing (HPDC-10)*, IEEE Press, Aug. 2001.

[39] "Isocra Livestore Cache", `http://www.isocra.co.uk/livestore/index.php`; acessado em 17 de maio de 2008.

[40] ALEXANDRE, A. M. A., *Cacheamento Transparente de Banco de Dados em Servidores de Comércio Eletrônico*, Master's Thesis, Universidade Federal do Rio de Janeiro, COPPE, agosto 2005.

[41] "The TPC's transaction Web e-commerce benchmark", `http://www.tpc.org/tpcw`; acessado em 23 de abril de 2008.

[42] "MySQL", `http://www.mysql.com`; acessado em 20 de maio de 2008.

[43] "MyISAM Storage Engine", `http://dev.mysql.com/doc/refman/5.0/en/myisam-storage-engine.html`; acessado em 20 de maio de 2008.

[44] DEGENARO, L., IYENGAR, A., LIPKIND, I., etal., "A Middleware System which Intelligently Caches Query Results". In: *Middleware '00: IFIP/ACM International Conference on Distributed systems platforms*, pp. 24–44, Springer-Verlag New York, Inc.: Secaucus, NJ, USA, 2000.

[45] GARCIA-MOLINA, H., ULLMAN, J. D., WIDOM, J., *Database System Implementation*. Prentice Hall, 2000.

[46] GRAY, J., REUTER, A., *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[47] ORFALI, R., HARKEY, D., EDWARDS, J., *Cliente/Servidor - Guia Essencial de Sobrevivência*. IDPI Press, 1996.

[48] WHITE, S., HAPNER, M., *JDBC 2.0 API*, Tech. rep., Sun Microsystems Inc., May 1998.

[49] GOLDSTEIN, R. C., STOREY, V. C., "Materialization", *IEEE Trans. on Knowl. and Data Eng.*, v. 6, n. 5, pp. 835–842, 1994.

[50] "Padrão SQL-92 (ISO/IEC 9075:1992)", `http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt/`;acessado em 30 de maio de 2008.

[51] "The Postgres' Transaction Isolation Levels", `http://www.postgresql.org/docs/7.4/static/transaction-iso.html`;
acessado em 30 de maio de 2008.

[52] "Java 2 Platform Standard Edition 5.0 API Specification", `http://java.sun.com/j2se/1.5.0/docs/api/`; acessado em 18 de abril de 2008.

[53] GAMMA, E., HELM, R., JOHNSON, R., etal., *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[54] "JUnit", `http://www.junit.org/`;acessado em 25 de maio de 2008.

[55] LAMPORT, L., "Time, Clocks, and the Ordering of Events in a Distributed System", *Commun. ACM*, v. 21, n. 7, pp. 558–565, 1978.

[56] HENNESSY, J., PATTERSON, D., *Arquitetura de Computadores - Uma Abordagem Quantitativa*. 3rd ed. Campus, 2003.

[57] "The JGroups Project", `http://www.jgroups.org/`; acessado em 14 de abril de 2008.

[58] "Java Inter-Process Communication",
`http://www.garret.ru/~knizhnik/jipc/jipc.html`;acessado em 1 de junho de 2008.

[59] "Java Object Serialization Specification", `http://java.sun.com/j2se/1.5.0/docs/guide/serialization/`; acessado em 2 de junho de 2008.

[60] "The Transaction Processing Performance Council", `http://www.tpc.org/`;
acessado em 23 de abril de 2008.

[61] "The Java TPC-W Implementation Distribution by UW-Madison Computer Architecture Group", http://www.ece.wisc.edu/~pharm/; acessado em 23 de abril de 2008.

[62] "The Java Servlet Technology", http://java.sun.com/products/servlet/; acessado em 23 de abril de 2008.

[63] "ObjectWeb - Open-Source Middleware", http://consortium.objectweb.org/; acessado em 23 de abril de 2008.

[64] COUNCIL, T. P. P., *TPC Benchmark W Specification*, Tech. rep., Transaction Processing Performance Council, February 2002.

[65] "The Postgres Database Server", http://www.postgresql.org/; acessado em 14 de abril de 2008.

[66] "The mod_proxy Connector", http://httpd.apache.org/docs/2.0/mod/mod_proxy.html; acessado em 14 de abril de 2008.

[67] "The AJPv13 Apache Tomcat Connector", http://tomcat.apache.org/connectors-doc/ajp/ajpv13a.html; acessado em 14 de abril de 2008.

[68] "The Postgres JDBC Driver", http://jdbc.postgresql.org/; acessado em 14 de abril de 2008.

[69] "Java Technologies", http://java.sun.com/javase/technologies/index.jsp; acessado em 14 de abril de 2008.

[70] "The JGroups' Replicated HashMap API", http://www.jgroups.org/javagroupsnew/docs/javadoc/org/jgroups/blocks/ReplicatedHashMap.html; acessado em 30 de abril de 2008.

[71] SOUNDARARAJAN, G., AMZA, C., "Using Semantic Information to Improve Transparent Query Caching for Dynamic Content Web Sites". In: *DEEC '05: Proceedings of the International Workshop on Data Engineering Issues in E-Commerce*, pp. 132–138, IEEE Computer Society: Washington, DC, USA, 2005.

[72] "The Library GNU Public Licence", http://www.opensource.org/licenses/lgpl-license.php; acessado em 14 de abril de 2008.

[73] "The Java 2 Enterprise Edition", `http://java.sun.com/javaee/`; acessado em 14 de abril de 2008.

[74] TANENBAUM, A. S., *Computer Networks*. 4th ed. Prentice Hall, 2003.

[75] SILBERSCHATZ, A., GALVIN, P. B., *Sistemas Operacionais - Conceitos*. 5th ed. Prentice Hall: São Paulo, 2000.

Apêndice A

Servidor de Relatórios

Para fornecer aos usuários e desenvolvedores informações sobre o seu funcionamento interno, foi agregada ao driver dCache a funcionalidade de geração de relatórios. Os relatórios gerados pelo driver fornecem informações como taxa de acertos nas caches, taxa de ocupação das caches, tempo médio de realização de *commits* e número de mensagens perdidas. Essas e outras informações são importantes, antes de mais nada, para permitir o aprimoramento do driver pelos desenvolvedores e, também, para permitir que os usuários possam configurar o driver de forma otimizada para as suas necessidades.

Uma vez que a funcionalidade de relatórios tenha sido habilitada pelo usuário do driver (conforme mostrado no apêndice B), as instâncias do driver começam a gerar relatórios contendo informações sobre o seu funcionamento. Os relatórios gerados pelas instâncias são então colecionados pelo servidor dCache (automaticamente ou sob-demanda, após solicitação do usuário) e ficam armazenados no seu sistema de arquivos, possibilitando consultas posteriores. No restante deste apêndice, descreveremos como a funcionalidade de relatórios foi implementada no driver. Primeiramente, daremos foco às classes e interfaces mais importantes do ponto de vista das instâncias do driver dCache. Depois, daremos focos às classes e interfaces mais importantes do ponto de vista do servidor dCache.

A.1 Os relatórios e as instâncias dCache

Foi criada a interface `Report`, responsável por definir o comportamento de um relatório genérico, gerado por um componente qualquer do driver dCache do qual se desejasse obter informações sobre o seu funcionamento. Essa interface possui um único método,

o `getHtmlString()` que retorna texto em HTML. O conteúdo do texto é definido por cada uma das classes que implementam a interface `Report`. Dentre as classes que implementam essa interface, podemos citar a `ConnectionReport`, a `PreparedStatementReport`, a `ResultSetReport` e a `SchemaReport`.

A classe `ConnectionReport` armazena informações relacionadas a execução dos métodos da classe `Connection`. A invocação ao método `getHtmlString()` implementado por esta classe retornará informações como: o tempo médio para execução de um commit ou tempo médio gasto na execução de uma chamada ao método `Connection#prepareStatement(String)`. Analogamente, as classes `PreparedStatementReport` e `ResultSetReport` armazenam e fornecem informações relacionadas a execução de métodos pertencentes, respectivamente, às classes `PreparedStatement` e `ResultSet`.

A classe `SchemaReport` tem o papel de reunir toda a informação de instrumentação relacionada aos *schemas* de banco de dados acessados pelo driver dCache. Cada instância dessa classe se responsabiliza por um, e somente um, *schema* acessado pelo driver. Essa classe possui, entre outros campos, um do tipo `ConnectionReport`, um do tipo `PreparedStatementReport` e um do tipo `ResultSetReport`. São mantidas nestes campos as informações de invocações à métodos realizadas em benefício de um *schema* específico. Em outras palavras, as classes `ConnectionReport`, `PreparedStatementReport` e `ResultSetReport` funcionam como sub-relatórios da classe `SchemaReport`. Como reflexo deste funcionamento hierárquico, é interessante mencionar que, o resultado da invocação aos métodos `getHtmlString()` de cada um dos três sub-relatórios mencionados, entra na composição do texto HTML retornado pela implementação do método `getHtmlString()` na classe `SchemaReport`.

A figura A.1 mostra a interface `Report` e as implementações citadas. Vale notar que a interface `Report` estende a interface `java.io.Serializable`. Esse comportamento é importante pois, como veremos adiante, os relatórios do driver dCache trafegarão pela rede.

A interface `Reportable` deve ser implementada pelos componentes do driver dos quais se deseja obter informações sobre o seu funcionamento. Esta interface possui apenas um método, chamado `getReport()`, que retorna um relatório contendo as informações do componente instrumentado. A classe `SchemaData`, por exemplo, é uma das classes que implementa esta interface. As instâncias da classe `SchemaData` armazenam informações sobre os *schemas* de banco de dados acessados pelo driver dCache. A implementação do método `getReport()` pela classe `SchemaData`, como era de se

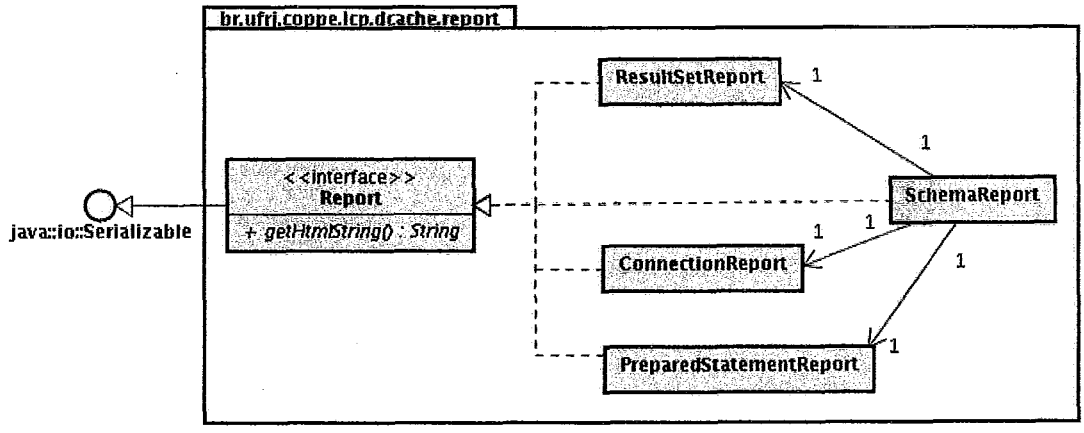


Figura A.1: Interface `Report` e as algumas de suas implementações

esperar, retorna um relatório do tipo `SchemaReport`, explicado anteriormente.

Como a arquitetura distribuída do driver dCache permite a existência de instâncias do driver dCache em diversos nós de processamento, seria conveniente conseguirmos acessar remotamente uma instância específica do driver e dela obter informações sobre o seu funcionamento a qualquer instante. A classe `ReportServer` é a responsável por essa funcionalidade. A classe `ReportServer`, que implementa a interface `java.lang.Runnable`, pode ser executada por uma *thread* independente, a qual permanecerá em *loop* aceitando conexões (através de um *socket* TCP) de clientes interessados em obter informações sobre o funcionamento de um componente instrumentado. O componente instrumentado nada mais é do que uma instância de uma classe qualquer que implemente a interface `Reportable`. Este componente instrumentado fornecerá os relatórios que serão entregues aos clientes que se conectarem no servidor de relatórios.

Cada instância do driver dCache possui um servidor de relatórios para cada *schema* de banco de dados por ela acessado. O componente instrumentado consultado por cada um destes servidores de relatórios será a instância da classe `SchemaReport` que detém as informações relativas ao *schema* de responsabilidade do servidor de relatórios. A figura A.2 mostra a associação da classe `ReportServer` com o seu componente instrumentado, além de exibir também a classe `SchemaData` descrita anteriormente.

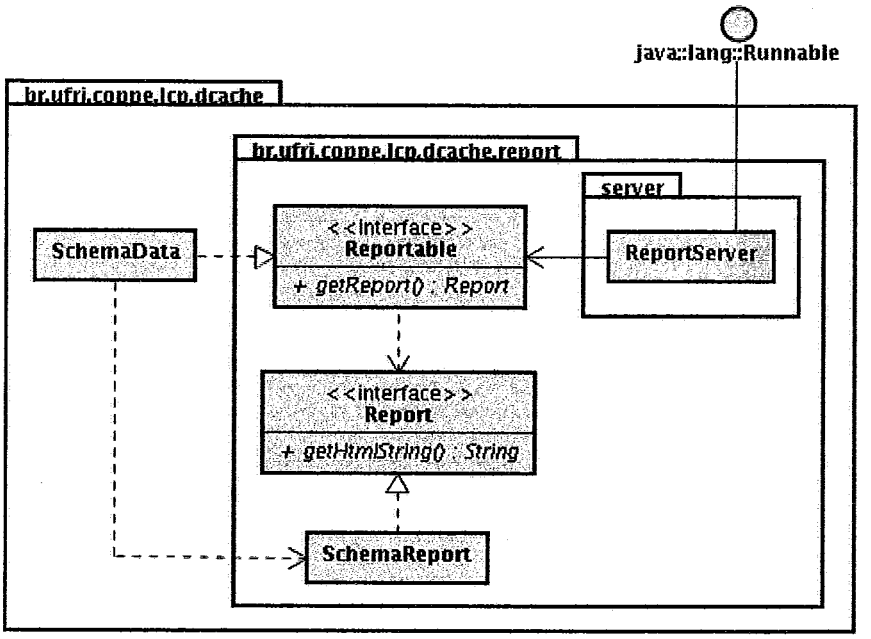


Figura A.2: Classe `ReportServer` e a interface `Reportable`

A.2 Os relatórios e o servidor dCache

Descrevemos até agora a instrumentação do driver dCache pela ótica das instâncias do driver. Vamos agora falar do suporte fornecido pelo servidor do driver dCache ao mecanismo de instrumentação descrito. Para o mecanismo de instrumentação, enquanto as instâncias exercem o papel de servidores de relatórios, o servidor do driver dCache exerce o papel de cliente destes servidores de relatório. Ou seja, o servidor dCache será utilizado para coletar as informações de instrumentação geradas de forma independente por cada instância do driver. A centralização das informações de instrumentação no servidor do driver dCache facilita o gerenciamento e análise da massa de dados gerada pelas instâncias[1].

A figura A.3 mostra as classes mais importantes para o mecanismo de instrumentação distribuído, do ponto de vista do servidor do driver dCache. A classe `Server`, representa o servidor dCache. O servidor dCache pode se associar a diversas instâncias da classe `ReportCollector`, uma para cada *schema* acessado pelo driver dCache. As instâncias da classe `ReportCollector` (que podemos chamar de coletores de relatórios) são as responsáveis por periodicamente acessar cada instância do driver solicitando os

[1]O apêndice B mostra como o usuário pode parametrizar a geração de relatórios de instrumentação usando os arquivos de configuração do servidor do driver dCache.

relatórios de um determinado *schema*. A comunicação com o servidor de relatórios é transparente para os coletores de relatórios. Ela é toda realizada pelas instâncias da classe `ReportServerStub`. Cada instância desta última classe é responsável pela comunicação com apenas um servidor de relatórios.

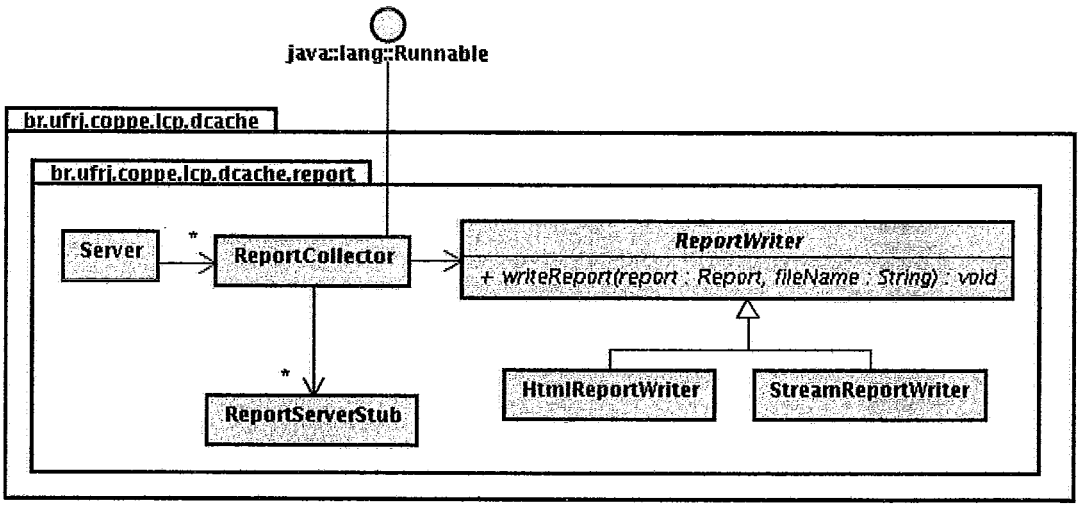


Figura A.3: Classes de instrumentação do lado servidor dCache

Ainda na figura A.3, podemos observar que a classe `ReportCollector` se associa com a classe abstrata `ReportWriter`. A classe `ReportWriter` define o método abstrato `writeReport(Report,String)` cuja implementação é responsável por gravar no disco rígido os relatórios obtidos pelos coletores de relatórios. Foram fornecidas duas implementações para este método: a realizada pela classe `HtmlReportWriter` e a realizada pela classe `StreamReportWriter`. A primeira implementação, grava os relatórios em arquivos HTML. Este formato é útil para inspeção visual dos dados de instrumentação. A segunda implementação, grava arquivos contendo a versão serializada dos relatórios. Esse formato é útil para manipulação dos dados de instrumentação por outros aplicativos[2].

[2]Para a análise do desempenho do driver dCache, por exemplo, foram escritas algumas aplicações em Java que recebem como entrada relatórios serializados e geram como saída arquivos CSV (*Comma Separated Values*) prontos para manipulação por programas de análise estatística e plotagem de gráficos.

Apêndice B

Manual do Driver

Apesar do recurso de caches, do ponto de vista da aplicação, o driver dCache é um driver JDBC como qualquer outro. Assim sendo, sua utilização e configuração, em princípio, é realizada como a de qualquer outro driver JDBC. Ou seja, basicamente, é necessário apenas que o driver esteja no *classpath* da aplicação, que a classe do driver seja carregada na JVM e que a *Uniform Resource Locator* (URL) de conexão apropriada seja utilizada para obter conexões [24]. Se o nome da classe do driver e a URL de conexão são mantidas em um arquivo de configuração separado do código-fonte da aplicação, a troca de um driver JDBC para outro não exige sequer a re-compilação da aplicação.

Veremos que, no caso da mudança para o driver dCache, são necessários os três cuidados mencionados (inerentes a qualquer driver JDBC) e, adicionalmente, também é necessária a preparação de dois tipos de arquivos de configuração. Estes arquivos de configuração adicionais são utilizados para permitir que as instâncias do driver encontrem o servidor dCache e para permitir que o usuário tenha maior controle sobre os parâmetros de funcionamento do driver, sobretudo, o funcionamento das caches.

B.1 Configuração básica

O driver dCache pode ser usado por aplicações *standalone* ou aplicações Web. Esta seção descreve como configurar e inicializar o driver dCache para ambos os casos. Estes são os passos fundamentais que devem ser seguidos para usar o driver dCache corretamente:

1. Criação do arquivo de configuração do servidor e dos arquivos de configuração das instâncias;
2. Ajuste da URL utilizada pela aplicação para conexão com o banco de dados;

3. Inicialização do servidor dCache;
4. Inicialização da aplicação *standalone* ou Web.

B.1.1 Arquivos de configuração

O driver dCache utiliza dois tipos de arquivos de configuração: *arquivo de configuração do servidor* e *arquivo de configuração de instância*. O *arquivo de configuração do servidor* é lido pelo servidor dCache e contém informações que serão utilizadas para configurar conexões aos bancos de dados e outros parâmetros de funcionamento. Os arquivos de configuração de instância são arquivos de configuração lidos pelas instâncias do driver dCache e contém as informações necessárias para que a instância conheça a localização do servidor e possa a ele se conectar.

Configuração do servidor

A configuração do servidor é feita através de um arquivo *Extensible Markup Language* (XML). O nome padrão para o arquivo de configuração é *dcacheserver.xml*. Outro nome pode ser escolhido, mas neste caso, o nome escolhido deverá ser especificado no momento da inicialização do servidor. O elemento XML raiz é o `server`. Ele espera, pelo menos, quatro elementos *top-level*: `tcpPort`, `udpPort1`, `udpPort2` e `schemas`. O elemento `tcpPort` indica a porta de comunicação TCP que será utilizada pelo servidor para enviar informações de configuração para as instâncias do driver. Os elementos `udpPort1` e `udpPort2` indicam portas UDP que serão usadas, respectivamente, pelo relógio global e pelo gerenciador de *locks* distribuídos (ver seções 2.4.1 e 2.4.2). Qualquer porta disponível no máquina do servidor dCache pode ser escolhida, para os três casos.

Os elemento `schemas` espera um ou mais sub elementos `schema`. Cada elemento `schema` contém informações relativas a um *schema* de banco de dados acessado através do driver dCache. Um atributo `id` deve ser especificado no elemento `schema`, cujo valor será usado para identificar o *schema* de banco de dados que está sendo descrito (veja a seção B.1.2). Opcionalmente, um atributo `description` pode ser utilizado para fornecer alguma informação adicional sobre o *schema* de banco de dados descrito. A listagem B.1 apresenta um fragmento de um arquivo de configuração de servidor. Neste fragmento, podemos ver a configuração para acessar o *schema tpcw* através do driver dCache.

```
1 <schema id="tpcw" description="TPC-W Benchmark Database">
```

```
2    <driver>org.postgresql.Driver</driver>
3    <url> jdbc:postgresql: //10.10.20.237/ tpcw?user=tpcw</url>
4
5    <statementsCacheMaxSize>1000</statementsCacheMaxSize>
6    <statementsCacheBufferSize>4096</statementsCacheBufferSize>
7
8    <relationsCacheMaxSize>1000</relationsCacheMaxSize>
9    <relationsCacheBufferSize>4096</relationsCacheBufferSize>
10
11   <tuplesCacheMaxSize>1000</tuplesCacheMaxSize>
12   <tuplesCacheBufferSize>4096</tuplesCacheBufferSize>
13
14   <instance address="10.10.20.123">
15     <statementsCachePort>1234</statementsCachePort>
16     <relationsCachePort>1235</relationsCachePort>
17     <tuplesCachePort>1236</tuplesCachePort>
18   </instance>
19
20   <instance address="computer1">
21     <statementsCachePort>1234</statementsCachePort>
22     <relationsCachePort>1235</relationsCachePort>
23     <tuplesCachePort>1236</tuplesCachePort>
24   </instance>
25 </schema>
```

Listagem B.1: Configuração para acesso ao banco de dados *tpcw* através do driver dCache.

A configuração informa ao driver dCache que ele deve utilizar o driver escravo `org.postgresql.Driver`, cuja URL de conexão é: "*jdbc:postgresql://10.10.20.237/tpcw?user=tpcw*". Duas instâncias dCache serão capazes de proporcionar o acesso a este base de dados. Uma está localizada em *10.10.20.123* e a outra em *computer1*. Ambas serão capazes de armazenar até 1000 itens em cada cache (não é possível configurar capacidades de caches diferentes para instâncias distintas). *Buffers* de 4096 bytes serão utilizados por ambas para a troca de mensagens entre as caches (ver seção 2.4.4).

Podem ser especificados quantos elementos `schema` se desejar. Basta que seja atribuído um valor diferente para cada atributo `id`. Dentro de cada elemento `instance` deve ser definido o número da porta que será utilizada por cada uma das três caches do driver: a cache de comandos SQL, a cache de relações e a cache de tuplas. Você pode escolher qualquer porta disponível na máquina que hospeda a instância do dCache. No exemplo, as portas 1234, 1235 e 1236 serão usadas em ambas as instâncias pela cache de comandos SQL, pela cache de relações e pela cache de tuplas, respectivamente. Supondo que o servidor dCache será executado no endereço *10.10.20.124*, a figura B.1 mostra a arquitetura descrita no arquivo de configuração da listagem B.1.

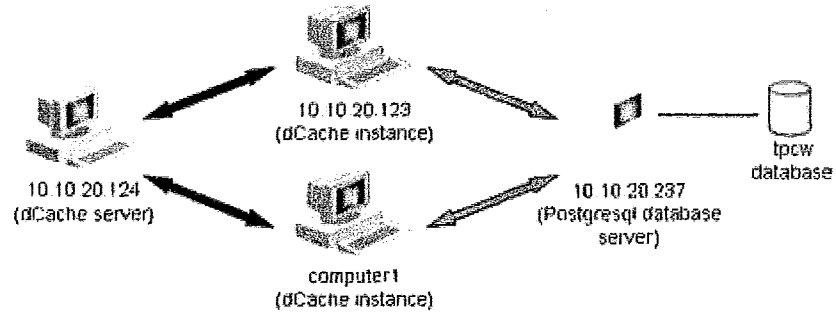


Figura B.1: Topologia representada pelo arquivo de configuração da listagem B.1.

Para permitir que o driver dCache acesse outro banco de dados além do *tpcw*, um novo elemento `schema` deve ser criado. Na listagem B.2, podemos ver um arquivo completo de configuração do servidor. O novo elemento `schema` é responsável por configurar o acesso ao banco de dados *testSchema*. Apenas uma instância de driver dCache acessa o *testSchema*. Essa instância está situada na mesma máquina que o servidor dCache. Logo, o endereço da instância é *localhost*.

```
1  <?xml version="1.0"?>
2
3  <!DOCTYPE server SYSTEM "dcacheserver.dtd">
4
5  <server>
6      <tcpPort>8978</tcpPort>
7      <udpPort1>8979</udpPort2>
8      <udpPort1>8980</udpPort2>
9      <schemas>
10         <schema id="tpcw" description="TPC-W Benchmark Database">
11             <driver>org.postgresql.Driver</driver>
12             <url> jdbc:postgresql: //10.10.20.237/ tpcw?user=tpcw</url>
13
14             <statementsCacheMaxSize>1000</statementsCacheMaxSize>
15             <statementsCacheBufferSize>4096</statementsCacheBufferSize>
16
17             <relationsCacheMaxSize>1000</relationsCacheMaxSize>
18             <relationsCacheBufferSize>4096</relationsCacheBufferSize>
19
20             <tuplesCacheMaxSize>1000</tuplesCacheMaxSize>
21             <tuplesCacheBufferSize>4096</tuplesCacheBufferSize>
22
23             <instance address="10.10.20.123">
24                 <statementsCachePort>1234</statementsCachePort>
25                 <relationsCachePort>1235</relationsCachePort>
26                 <tuplesCachePort>1236</tuplesCachePort>
27             </instance>
28
```

```
29              <instance address="computer1">
30                  <statementsCachePort>1234</statementsCachePort>
31                  <relationsCachePort>1235</relationsCachePort>
32                  <tuplesCachePort>1236</tuplesCachePort>
33              </instance>
34          </schema>
35
36          <schema id="testSchema" description="the tests schema">
37              <driver>org.hsqldb.jdbcDriver</driver>
38              <url>jdbc:hsqldb:hsql: //10.10.20.238/ testSchema</url>
39
40              <statementsCacheMaxSize>800</statementsCacheMaxSize>
41              <statementsCacheBufferSize>1024</statementsCacheBufferSize>
42
43              <relationsCacheMaxSize>5000</relationsCacheMaxSize>
44              <relationsCacheBufferSize>32768</relationsCacheBufferSize>
45
46              <tuplesCacheMaxSize>3000</tuplesCacheMaxSize>
47              <tuplesCacheBufferSize>4096</tuplesCacheBufferSize>
48
49              <instance address="localhost">
50                  <statementsCachePort>1234</statementsCachePort>
51                  <relationsCachePort>1235</relationsCachePort>
52                  <tuplesCachePort>1236</tuplesCachePort>
53              </instance>
54          </schema>
55      </schemas>
56  </server>
```

Listagem B.2: Configuração para acesso ao banco de dados *tpcw* através do driver dCache.

Note que uma definição DOCTYPE é feita no começo do arquivo de configuração. Para uma definição formal da sintaxe esperada pelo arquivo de configuração do servidor, o seu arquivo *Document Type Definition* (DTD) é mostrado na listagem B.3. Caso o arquivo DTD do servidor dCache seja especificado dentro da declaração DOCTYPE, ele precisará estar acessível pelo servidor dCache no momento da sua inicialização. Caso contrário, uma exceção de *parsing* será lançada. Na figura B.2 é representada a topologia representada pelo arquivo de configuração completo (listagem B.2).

```
1  <!-- DTD for the dCache server's XML configuration file . -->
2
3  <!ELEMENT server (tcpPort,udpPort1,udpPort2,reports ?,schemas)>
4
5    <!ELEMENT tcpPort (#PCDATA)>
6    <!ELEMENT udpPort1 (#PCDATA)>
7    <!ELEMENT udpPort2 (#PCDATA)>
8
```

```
<!ELEMENT reports EMPTY>
  <!ATTLIST reports path CDATA #REQUIRED>

<!ELEMENT schemas (schema+)>

  <!ELEMENT schema (driver,url,statementsCacheMaxSize,
      statementsCacheBufferSize ,
      relationsCacheMaxSize, relationsCacheBufferSize ,
      tuplesCacheMaxSize,tuplesCacheBufferSize ,
      reporting ?, instance+)>
    <!ATTLIST schema
      id ID #REQUIRED
      description CDATA #IMPLIED>
    <!ELEMENT driver                    (#PCDATA)>
    <!ELEMENT url                       (#PCDATA)>
    <!ELEMENT statementsCacheMaxSize (#PCDATA)>
    <!ELEMENT statementsCacheBufferSize (#PCDATA)>
    <!ELEMENT relationsCacheMaxSize  (#PCDATA)>
    <!ELEMENT relationsCacheBufferSize (#PCDATA)>
    <!ELEMENT tuplesCacheMaxSize     (#PCDATA)>
    <!ELEMENT tuplesCacheBufferSize  (#PCDATA)>

    <!ELEMENT reporting (automaticReporting?)>
      <!ATTLIST reporting enabled ( true | false ) " false ">
      <!ELEMENT automaticReporting EMPTY>
        <!ATTLIST automaticReporting
          interval CDATA "60"
          enabled ( true | false ) " false "
          format (html|stream) "html">

    <!ELEMENT instance (statementsCachePort, relationsCachePort ,
                        tuplesCachePort , reportServerPort ?)>
      <!ATTLIST instance address CDATA #REQUIRED>
      <!ELEMENT statementsCachePort (#PCDATA)>
      <!ELEMENT relationsCachePort (#PCDATA)>
      <!ELEMENT tuplesCachePort  (#PCDATA)>
      <!ELEMENT reportServerPort  (#PCDATA)>
```

Listagem B.3: DTD para o arquivo de configuração do servidor dCache.

O arquivo de configuração do servidor é usado também para habilitar e configurar a funcionalidade de *relatórios dCache*. Os relatórios dCache fornecem ao usuário informações sobre o funcionamento interno do driver[1]. A configuração desta funcionalidade é explicada na seção B.2.

[1]Os relatórios dCache são utilizados na avaliação do driver (ver seção 3.2). No apêndice A são mostrados maiores detalhes sobre a implementação da funcionalidade de relatórios.

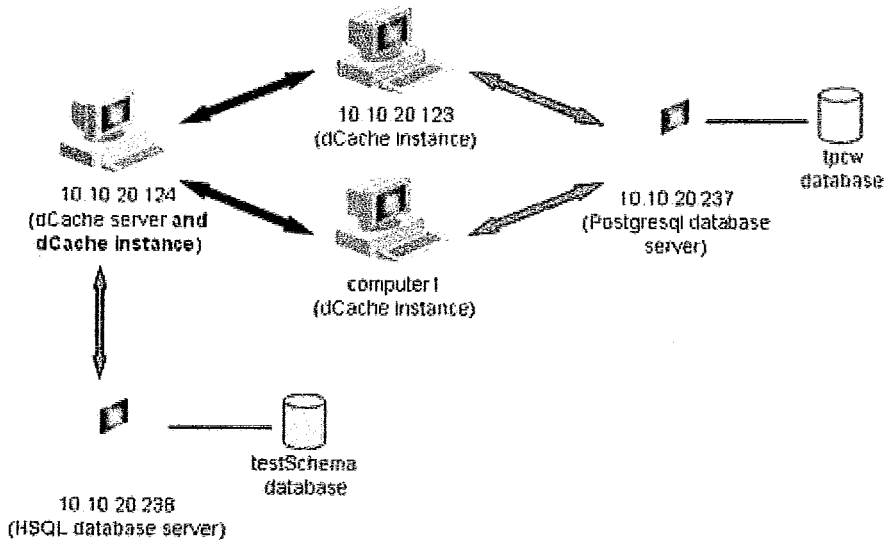


Figura B.2: Topologia representada pelo arquivo de configuração da listagem B.2.

Configuração de instâncias

A configuração das instâncias do driver dCache é também realizada através de um arquivo de configuração XML. O nome padrão para o arquivo de configuração de instância é *dcache.xml* e não pode ser alterado. O arquivo de configuração de instância é mais simples que o de configuração de servidor. O elemento XML raiz é o `dCache`, que espera quatro sub elementos: `serverAddress` e `tcpPort`, `udpPort1` e `udpPort2`. O `serverAddress` contém o endereço do servidor dCache. Tanto o endereço *Internet Protocol* (IP) do servidor quando o nome da máquina podem ser utilizados. Os valores dos três outros elementos devem coincidir com os números de porta especificados nos elementos de mesmo nome no arquivo de configuração do servidor. Na listagem B.4, podemos ver o arquivo de configuração de instância que deve ser usado nas instâncias dCache descritas na seção B.1.1.

Note que uma definição DOCTYPE é feita no começo do arquivo de configuração. Para uma definição formal da sintaxe esperada no arquivo de configuração das instâncias, o arquivo DTD apontado pela definição DOCTYPE é mostrado na listagem B.5. Caso o arquivo DTD seja especificado dentro da declaração DOCTYPE, ele precisará estar acessível pela instância dCache no momento da sua inicialização. Caso contrário, uma exceção de *parsing* será lançada.

```
1  <?xml version="1.0"?>
2
3  <!DOCTYPE dCache SYSTEM "dcache.dtd">
4
5  <dCache>
6      <serverAddress>10.10.20.124</serverAddress>
7      <tcpPort>8978</tcpPort>
8      <udpPort1>8979</udpPort1>
9      <udpPort2>8980</udpPort2>
10 </dCache>
```

Listagem B.4: Exemplo de arquivo de configuração de uma instância dCache.

```
1  <!-- DTD for the dCache instance's XML configuration file . -->
2
3  <!ELEMENT dCache (serverAddress,tcpPort,udpPort1,udpPort2)>
4
5    <!ELEMENT serverAddress (#PCDATA)>
6    <!ELEMENT tcpPort (#PCDATA)>
7    <!ELEMENT udpPort1 (#PCDATA)>
8    <!ELEMENT udpPort2 (#PCDATA)>
```

Listagem B.5: DTD para os arquivos de configuração de instâncias dCache.

B.1.2 URL de conexão

O driver dCache aceita conexões cuja URL de conexão começa com *"jdbc:dCache:"*. Após esse prefixo, deve ser especificado o banco de dados ao qual se deseja conectar. Supondo que uma aplicação quer se conectar com o banco de dados *tpcw*, a URL de conexão será: *"jdbc:dCache:tpcw"*. O nome do banco de dados deve ser igual ao valor do atributo `id` de algum dos elementos `schema` previamente definidos no arquivo de configuração do servidor. A listagem B.6 mostra um código-fonte típico, obtendo uma conexão Postgres para o banco de dados *tpcw*.

Supondo o arquivo de configuração de servidor descrito na seção B.1.1, o código-fonte que obtém uma conexão para o mesmo banco, porém usando o driver dCache é mostrado na listagem B.7.

Opcionalmente, os parâmetros de conexão podem ser especificados através de um objeto `java.util.Properties`. Este objeto pode ser usado para especificar tanto parâmetros do driver escravo quanto o próprio identificador do *schema* dCache ao qual se deseja conectar. Na listagem B.8 é mostrado um código-fonte que utiliza um objeto de propriedades para passar argumentos para o driver dCache e para o driver escravo.

```
1  try {
2      Class.forName("org.postgresql.Driver");
3
4      Connection con = DriverManager.getConnection(
5          "jdbc:postgresql://10.10.20.237/tpcw?user=tpcw");
6
7      con.createStatement().executeQuery("SELECT * FROM TABLE");
8  } catch (Exception e) {
9      e.printStackTrace();
10 }
```

Listagem B.6: Exemplo de código Java obtendo uma conexão para o banco *tpcw* usando o driver Postgres.

```
1  try {
2      Class.forName("br.ufrj.coppe.lcp.dcache.Driver");
3
4      Connection con = DriverManager.getConnection(
5          "jdbc:dcache:tpcw");
6
7      con.createStatement().executeQuery("SELECT * FROM TABLE");
8  } catch (Exception e) {
9      e.printStackTrace();
10 }
```

Listagem B.7: Exemplo de código Java obtendo uma conexão para o banco *tpcw* usando o driver dCache.

```
1  try {
2      Class.forName("br.ufrj.coppe.lcp.dcache.Driver");
3
4      Properties prop = new Properties();
5      prop.put("user", "tpcw");
6      prop.put("dCache.schemaName", "tpcw");
7
8      Connection con = DriverManager.getConnection("jdbc:dcache:",prop);
9
10     con.createStatement().executeQuery("SELECT * FROM TABLE");
11 } catch (Exception e) {
12     e.printStackTrace();
13 }
```

Listagem B.8: Exemplo de código Java obtendo uma conexão para o banco *tpcw* usando um objeto `Properties`.

Note que a URL de conexão do driver dCache não especifica o nome do *schema*. Porém, o objeto `Properties` tem uma chave chamada *dCache.schemaName*, responsável pela identificação do *schema*. Da mesma forma, a URL de conexão do driver escravo não precisa especificar o parâmetro *user*, uma vez que ele já está especificado pelo objeto `Properties`. Esta é uma importante estratégia para definir parâmetros dinâmicos, cujos valores não podem ser definidos no arquivo de configuração do servidor dCache, em tempo de configuração.

B.1.3 Inicializando o servidor

Seja uma aplicação *standalone* ou Web, o servidor dCache deve ser inicializado antes da aplicação tentar usar o driver dCache. Supondo que o JRE já esteja corretamente instalado na máquina que executará o servidor dCache e que o arquivo de configuração do servidor tenha sido criado apropriadamente, estas são as etapas para inicializar o servidor dCache:

1. Copiar o arquivo *jar* do driver dCache para um diretório qualquer na máquina onde executará o servidor dCache. O nome do arquivo *jar* varia conforme a versão do dCache. Na versão 2.16.0, por exemplo, o nome do arquivo jar é *dcache-2.16.0.jar*.
2. Copiar todas as dependências do dCache (outros arquivos *jar* contendo bibliotecas) para o mesmo diretório escolhido.
3. Colocar o arquivo de configuração do servidor do dCache no diretório de onde será disparado o servidor (esse diretório não precisa ser o mesmo onde o arquivo *jar* foi colocado, ele é o diretório de onde será executada a JVM).
4. Se o arquivo de configuração do servidor usa uma declaração DOCTYPE, colocar o arquivo DTD correto também no diretório de onde será disparado o servidor.
5. Supondo que o diretório corrente é aquele onde se encontra o *jar* do driver[2], o servidor pode ser iniciado com:
    - Se estiver usando um arquivo padrão de configuração:

      ```
      java -jar dCache-2.16.0.jar
      ```
    - Se estiver usando um arquivo de configuração com nome personalizado:

      ```
      java -jar dCache-2.16.0.jar <nome do arquivo de
      configuração>
      ```

[2]De um outro diretório que não o do arquivo *jar*, o caminho do arquivo *jar* deve ser especificado quando a JVM for executada: `java -jar <path-to-the-jar>dCache-2.16.0.jar`

- para iniciar o servidor em *modo interativo*[3]:

```
java -jar dCache-2.16.0.jar -iterative
```

B.1.4 Inicializando uma aplicação *standalone*

Nessa seção descreveremos como usar o driver dCache numa aplicação *standalone* (independentemente da aplicação estar "empacotada" em um arquivo *jar* ou não). Supondo que o JRE já tenha sido instalado, o arquivo de configuração da instância já tenha sido criado, e a URL de conexão com driver tenha sido ajustada corretamente, estas são as etapas para iniciar a aplicação:

1. Iniciar o servidor dCache (conforme explicado na seção B.1.3).
2. Copiar o arquivo *jar* do driver dCache para um diretório qualquer, na máquina que executará a aplicação.
3. Copiar todas as dependências do dCache para o mesmo diretório escolhido.
4. Colocar o arquivo *jar* dCache e o arquivo *jar* do driver escravo no *classpath* da aplicação (note que as dependências do dCache não precisam ser declaradas no *classpath* da aplicação). O driver escravo deve ser o mesmo definido pelo arquivo de configuração do servidor.
5. Colocar o arquivo de configuração da instância no mesmo diretório de onde será disparada a aplicação.
6. Se o arquivo de configuração da instância usa uma declaração DOCTYPE, colocar o arquivo DTD apropriado também no diretório de onde será disparada a aplicação.
7. Disparar a aplicação convencionalmente.

B.1.5 Inicializando uma aplicação Web

Nesta seção iremos descrever a forma de utilizar o driver dCache em uma aplicação Web instalada em um servidor Tomcat.

1. Instalar a aplicação Web no servidor Tomcat convencionalmente.

[3]Por *default*, o servidor dCache executa em modo *não interativo*. Isso quer dizer que, por *default*, não é possível interagir com o servidor durante a sua execução. Já o modo interativo permite que o usuário envie comandos para o servidor enquanto ele encontra-se em execução. O envio de comandos é feito através de um *prompt* de comandos aberto para usuário quando a opção *-iterative* é utilizada na inicialização do servidor dCache.

2. Se não existe ainda, criar o diretório `lib`, dentro do diretório `WEB-INF` da aplicação. O *layout* final será:
   `tomcat-installation/webapps/your-app/WEB-INF/lib`
3. Copiar o arquivo *jar* do dCache para o diretório `WEB-INF/lib`.
4. Copiar todas as dependências do dCache para o diretório `WEB-INF/lib`.
5. Copiar o arquivo *jar* do driver escravo para o diretório `WEB-INF/lib`. O driver escravo deve ser o mesmo definido no arquivo de configuração do servidor.
6. Colocar o arquivo de configuração de instância no diretório `WEB-INF`.
7. Configurar e iniciar o servidor dCache corretamente.
8. Iniciar (ou reiniciar) sua aplicação Web.

Para utilizar o driver dCache em um cluster de Tomcats, basta repetir os passos acima para cada instância do Tomcat. É importante lembrar que as informações de todas as instância dCache devem ser fornecidas no arquivo de configuração do servidor.

B.2 Configurando o servidor de relatórios

O driver dCache é capaz de gerar relatórios que fornecem informações sobre o seu funcionamento, tais como: taxas de acerto no acesso às caches, número de consultas SQL realizadas, entre muitas outras. Os relatórios são gerados pelas instâncias do driver dCache quando solicitadas pelo servidor dCache (ver apêndice A). O servidor dCache recolhe os relatórios gerados e os armazena no seu sistema de arquivos local. O servidor dCache pode ser configurado para solicitar relatórios às instâncias mediante solicitação do usuário[4] (*ad-hoc*) ou de forma automática, periodicamente. O restante desta seção explica como configurar o servidor dCache para a geração de relatórios.

Independentemente do forma de geração dos relatórios (*ad-hoc* ou automática), a funcionalidade de relatórios deve ser ativada através do arquivo de configuração do servidor dCache (os relatórios são desativados por *default* para todos os *schemas*). A geração de relatórios pode ser ativada ou desativada individualmente para *schema* acessado pelo dCache. O elemento `report` é o responsável por habilitar ou desabilitar a funcionalidade de relatórios. O fragmento do arquivo de configuração de servidor apresentado na listagem B.9 mostra como habilitar os relatórios para o *schema tpcw*.

[4]Os relatórios podem ser solicitados através do *prompt* de comandos do servidor dCache. Para habilitar o *prompt* de comandos basta inicializar o servidor dCache com a opção `-iterative`, conforme mencionado na seção B.1.3.

```
1  <schema id="tpcw" description="TPC-W Benchmark Database">
2      <driver>org.postgresql.Driver</driver>
3      <url>jdbc:postgresql://10.10.20.237/tpcw?user=tpcw</url>
4
5      <statementsCacheMaxSize>1000</statementsCacheMaxSize>
6      <statementsCacheBufferSize>4096</statementsCacheBufferSize>
7
8      <relationsCacheMaxSize>1000</relationsCacheMaxSize>
9      <relationsCacheBufferSize>4096</relationsCacheBufferSize>
10
11     <tuplesCacheMaxSize>1000</tuplesCacheMaxSize>
12     <tuplesCacheBufferSize>4096</tuplesCacheBufferSize>
13
14     <reporting enabled="true"/>
15
16     <instance address="10.10.20.123">
17         <statementsCachePort>1234</statementsCachePort>
18         <relationsCachePort>1235</relationsCachePort>
19         <tuplesCachePort>1236</tuplesCachePort>
20         <reportServerPort>1237</reportServerPort>
21     </instance>
22
23     <instance address="computer1">
24         <statementsCachePort>1234</statementsCachePort>
25         <relationsCachePort>1235</relationsCachePort>
26         <tuplesCachePort>1236</tuplesCachePort>
27         <reportServerPort>1237</reportServerPort>
28     </instance>
29 </schema>
```

Listagem B.9: Exemplo de configuração de instância para geração de relatórios dCache.

Note que, uma vez que a geração de relatórios foi habilitada para um *schema*, todas as suas instâncias dCache devem declarar um elemento `reportServerPort` (no arquivo de configuração do servidor). Este elemento define qual será a porta de comunicação que irá ouvir os pedidos para a geração de relatórios daquela instância.

B.2.1 Modo *ad-hoc*

Uma vez que a geração de relatórios tenha sido ativada para o *schema* de banco de dados do qual se deseja obter relatórios, o *prompt* de comandos do servidor dCache pode ser utilizado para solicitar os relatórios. O comando `report`, do *prompt* de comandos, espera até dois parâmetros. O primeiro, é a máquina onde roda o servidor de relatórios (conforme explicado no apêndice A, cada instância do driver dCache desempenha o papel de um servidor de relatórios). O segundo parâmetro (opcional) é o número da porta do servidor de relatórios. Se o segundo parâmetro for omitido, o servidor dCache irá solicitar o relatório para o primeiro servidor de relatórios localizado na máquina especificada.

No fragmento apresentado na listagem B.9, os relatórios estão habilitados para o *schema tpcw*. Portanto, ao utilizar o *prompt* de comandos do servidor dCache, o endereço das instâncias deste *schema* (*10.10.20.123* e *computer1*) podem ser usados para a obtenção de relatórios. Os relatórios gerados serão armazenados no mesmo diretório a partir do qual o *prompt* de comandos do servidor foi disparado. O formato do arquivo do relatório pode ser escolhido. Atualmente, existem dois formatos disponíveis: HTML, e objeto Java serializado. A partir do *prompt* de comandos, o comando `format` pode ser utilizado, antes da chamada ao comando `report`, para definir o formato desejado para o arquivo.

B.2.2 Modo automático

Os relatórios automáticos também são ativados através do arquivo de configuração do servidor dCache. Os relatórios automáticos podem ser ativados ou desativados individualmente para cada *schema* acessado através do driver dCache. Supondo que a funcionalidade de relatórios já tenha sido habilitada para um determinado *schema*, o sub elemento `automaticReporting` deve ser declarado no interior do elemento `reporting` para configurar os relatórios automáticos. O fragmento apresentado na listagem B.10 mostra o elemento `reporting` de um *schema* cujos relatórios automáticos estão ativados.

A geração automática de relatórios está desabilitada por *default*. O atributo

```
1 <reporting enabled="true">
2     <automaticReporting enabled="true" format="stream" interval="120"/>
3 </reporting>
```

Listagem B.10: Fragmento de um arquivo de configuração do servidor mostrando exemplo de configuração de uma instância dCache para geração automática de relatórios dCache.

`enabled` permite que a geração automática seja habilitada. O atributo `format` permite a definição de qual será o formato usado para gravar, no sistema de arquivos, os relatórios gerados automaticamente. Este atributo aceita os valores `html` ou `stream`. Quando é usado o valor `html`, os relatórios serão escritos como arquivos HTML. Caso contrário, os arquivos serão gravados a partir da serialização do objetos Java que representam relatórios. O formato *default* é `html`. O atributo `interval` define o intervalo (em segundos) entre duas gerações consecutivas de relatórios automáticos. O valor *default* é 60.

Os relatórios gerados automaticamente são separados em diretórios distintos, um para cada *schema* no qual a geração automática de relatórios foi habilitada. O servidor dCache irá criar diretórios usando os nomes dos *schemas* como nomes dos diretórios. Os diretórios referentes a cada *schema* serão criados, por *default*, dentro do diretório `reports`, que será criado, por sua vez, no mesmo caminho de onde o servidor dCache foi disparado. Os relatórios automáticos sempre serão separados em diretórios com o mesmo nome do *schema* relacionado, mas o local onde estes diretórios serão criados pode ser alterado através do arquivo de configuração do servidor dCache.

O elemento *top-level* `reports` permite definir um caminho personalizado para os diretórios de relatórios. O fragmento apresentado na listagem B.11 mostra o início de um arquivo de configuração de servidor que utiliza os elemento `reports`. A configuração define o diretório *myCustomReportsDirectory* (na raiz do sistema de arquivos) como o diretório dCache para relatórios automáticos. Usando esta configuração, os diretórios de relatórios automáticos dos *schemas* serão criados dentro de */myCustomReportsDirectory/*.

```
1  <?xml version="1.0"?>
2
3  <!DOCTYPE server SYSTEM "dcacheserver.dtd">
4
5  <server>
6      <tcpPort>8978</tcpPort>
7      <udpPort1>8979</udpPort1>
8      <udpPort2>8980</udpPort2>
9
10     <reports path="/myCustomReportsDirectory"/>
11
12     <schemas>
13         <schema id="tpcw" description="TPC-W Benchmark Database">
14             <driver>org.postgresql.Driver</driver>
15     ...
```

Listagem B.11: Exemplo de configuração de instância para geração automática de relatórios dCache.

Apêndice C

JGroups

O *JGroups* [57] é um projeto que desenvolve um conjunto de ferramentas (*toolkit*) para comunicação em grupo confiável. A tabela C.1 mostra de forma simplista, porém bastante objetiva, o escopo do projeto JGroups no âmbito da comunicação em rede. Enquanto os protocolos UDP e TCP são utilizados, respectivamente, para comunicação unicast não confiável e confiável, IP multicast e o toolkit JGroups podem ser utilizados para comunicação multicast não confiável e confiável, respectivamente.

As principais funcionalidades do toolkit JGroups são:

- Criação e destruição de grupos (os membros de um grupo podem estar dispersos por LANs ou WANs);
- Inserção e remoção de membros nos grupos;
- Detecção e notificação da inserção, remoção e falhas de membros pertencentes aos grupos;
- Detecção e remoção de membros que falharam;
- Envio e recepção de mensagens de membros para grupos;
- Envio e recepção de mensagens de membros para membros.

Aplicações escritas em Java podem se beneficiar dos recursos fornecidos pelo toolkit (que também é implementado em Java) utilizando uma API que se baseia em uma

	Não Confiável	Confiável
Unicast	UDP	TCP
Multicast	IP Multicast	**JGroups**

Tabela C.1: Escopo do projeto JGroups

abstração, muito semelhante a um *socket* UDP, denominada *canal*. Para se comunicar com um grupo, um membro precisa instanciar um canal e conectá-lo ao grupo desejado. Depois, envia e recebe mensagens através de primitivas *send* e *receive*. No momento da criação de um canal, o usuário tem a oportunidade de definir uma pilha de protocolos que será usada para a transmissão das mensagens. O toolkit disponibiliza uma série de protocolos para que o usuário possa definir com flexibilidade a pilha que melhor se adapta às necessidades da sua aplicação. Recursos como a ordenação de mensagens e criptografia, podem ser habilitados ou não de acordo com a pilha definida.

Outra facilidade oferecida pelo toolkit JGroups aos desenvolvedores de aplicações distribuídas são os blocos básicos de construção (*building blocks*). Ao contrário do canal, que é uma abstração básica, os building blocks são abstrações mais sofisticadas construídas a partir dos canais. Os building blocks têm por objetivo fornecer ao desenvolvedor ferramentas com propósito bastante objetivo e facilmente aplicáveis na elaboração de aplicações mais sofisticadas. São exemplos de building blocks fornecidos pelo JGroups: a tabela hash distribuída (*DHT - Distributed Hash Table*) e o gerenciador de locks distribuídos (*DLM - Distributed Lock Manager*). Um detalhe interessante, é que alguns building blocks, como a distributed hash table implementam interfaces da API JDK, facilitando o uso dessas estruturas distribuídas em aplicações que já utilizavam implementações padrão dessas interfaces.

O JGroups é um projeto de código aberto (*open source project*), distribuído sob a licensa *Library GNU Public License (LGPL)* [72] e está ativo pelo menos desde o ano 2000. O toolkit é utilizado por alguns projetos bastante populares como os servidores de aplicação *Java 2 Platform Enterprise Edition (J2EE)* [73] *JBoss* [14] e *JOnAS* [16], o *container servlet Tomcat* [13] e o servidor Web *Jetty* [11]. O toolkit JGroups é utilizado para a implementação de clusters de servidores de aplicação JBoss e JOnAS. No caso dos servidores Tomcat e Jetty, é empregado na réplica de sessões HTTP.

Apêndice D

JNetwork

Apresentaremos neste apêndice a biblioteca *JNetwork*, que fornece ferramentas de comunicação distribuída para aplicações escritas em Java. A biblioteca foi desenvolvida no decorrer do presente trabalho com objetivo de suprir ferramentas necessárias para a implementação da arquitetura distribuída do driver dCache, mas é genérica o suficiente para ser reutilizada por qualquer aplicação Java que precise dos recursos por ela disponibilizados.

A biblioteca JNetwork é apresentada na forma de um pacote *jar*, por este motivo também nos referiremos a ela como "o pacote JNetwork". O pacote JNetwork fornece as seguintes ferramentas para comunicação distribuída:

- Um conversor de objetos em *array* de *bytes* e vice-versa;
- Um canal de comunicação implementado em cima do protocolo UDP;
- Um gerenciador de *locks* distribuídos.

D.1 Utilitário ByteArrayConverter

Para realizar comunicação distribuída utilizando a API de *sockets* UDP fornecida pelo JDK é comum que o programador se depare com a necessidade de converter objetos em *arrays* de *bytes* para então encapsulá-lo dentro de um datagrama e enviá-lo ao destinatário através do *socket* UDP. Analogamente, quando é recebido um datagrama por um *socket* UDP, é comum que os *bytes* recebidos sejam convertidos em um objeto Java. Para facilitar a programação destas duas conversões, foi criado o utilitário `ByteArrayConverter`.

O utilitário possui apenas dois métodos estáticos: `getBytes` e `fromBytes` (ver figura D.1). O primeiro, recebe um objeto Java serializável e devolve um *array* de *bytes* que o representa. O segundo, recebe um *array* de *bytes* e devolve o objeto Java por ele

representado. A implementação é realizada de forma que a invocação realizada na linha 3 da listagem D.1 sempre retorne o valor "verdadeiro".

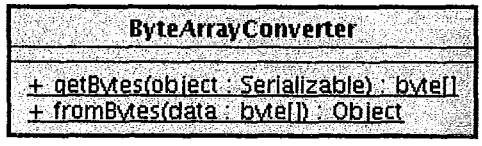


Figura D.1: Classe ByteArrayConverter.

```
1  ClasseSerializavel  objeto1 = new ClasseSerializavel ();
2  ClasseSerializavel  objeto2 = ByteArrayConverter.fromBytes(ByteArrayConverter.
       getBytes(object1));
3  objeto1.equals(objeto2);
```

Listagem D.1: Exemplo de uso da classe ByteArrayConverter do pacote JNetwork.

D.2 Utilitário UDPChannel

A pilha de protocolos IP admite dois protocolos na camada de transporte: UDP e TCP [74, 47]. Fundamentalmente, o protocolo UDP fornece comunicação não-confiável e não-ordenada baseada em troca de datagramas, enquanto o protocolo TCP garante comunicação confiável e ordenada baseada em conexão. Apesar das características desejáveis oferecidas pelo protocolo TCP, muitas aplicações optam pela utilização de UDP graças ao seu melhor desempenho em comparação ao primeiro. O utilitário *UDPChannel* fornece às aplicações Java uma abstração de canal de comunicação confiável e ordenado implementado em cima do protocolo UDP.

A figura D.2 mostra as classes e interfaces relacionadas à implementação do utilitário UDPChannel. A interface `UDPChannelMessage` define uma mensagem que pode ser enviada e recebida através do canal de comunicação. Aplicações que desejem utilizar o utilitário devem criar mensagens que implementem esta interface. A implementação do método `setUDPChannelMessageId` precisa apenas gravar em algum campo inteiro um valor de identificador recebido como argumento. A implementação do método `getUDP-ChannelMessageId` deve retornar o valor gravado no campo. Esse identificador é usado pelo utilitário para garantir a ordenação das mensagens. Para facilitar a utilização da ferramenta, o pacote fornece ao desenvolvedor a alternativa de criar mensagens que herdem

da classe `SimpleUDPChannelMessage`, que já realiza uma implementação padrão dos métodos da interface `UDPChannelMessage`.

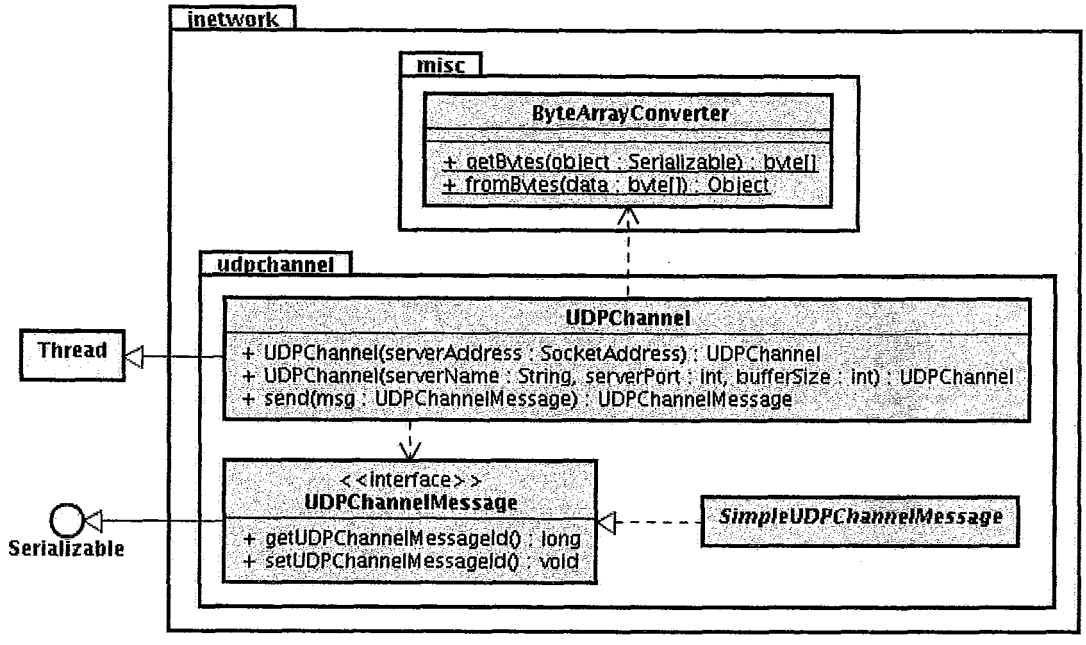


Figura D.2: Classes e interfaces utilizadas na implementação do canal de comunicação UDP.

Criadas as mensagens compatíveis com o utilitário, a classe `UDPChannel` deve ser utilizada para enviar e receber as mensagens. A invocação ao método `send` desta classe faz com que a mensagem passada como argumento seja enviada ao destinatário informado no construtor da classe. O chamador permanecerá bloqueado até que seja recebida uma mensagem de resposta à mensagem enviada. A implementação da classe `UDPChannel` se preocupa com a eventual necessidade de retransmissão, com a ordenação das mensagens e com o casamento das mensagens enviadas com suas respectivas mensagens de resposta.

As mensagens enviadas pelo canal UDP podem ser recebidas convencionalmente por qualquer *socket* UDP. Fora a necessidade de implementação da interface `UDPChannelMessage`, a única exigência adicional feita pelo utilitário é que o servidor, ao receber uma mensagem enviada pelo canal, deve obrigatoriamente respondê-la, e a resposta também deve ser feita através de uma `UDPChannelMessage` contendo o mesmo identificador recebido.

Duas observações interessantes merecem ser feitas: A classe `UDPChannel` herda de `java.lang.Thread`, por isso, pode ser mantida executando em segundo plano, envi-

ando e aguardando a chegada de mensagens. A interface `UDPChannelMessage` herda a interface `java.io.Serializable`, permitindo que a classe `UDPChannel` se utilize do utilitário ByteArrayConverter, fornecido pelo próprio pacote JNetwork, para tratar suas mensagens.

D.3 Gerenciador de *locks* distribuídos

A ferramenta mais sofisticada fornecida pelo pacote JNetwork é o *Distributed Locks Manager* (DLM). O JNetworks se utiliza do seu próprio UDPChannel para construir um gerenciador de *locks* distribuídos com primitivas semelhantes às fornecidas aos programadores Java pelo pacote `java.util.concurrent.locks` da API JSE. O DLM fornecido pelo pacote JNetwork conta, inclusive, com *locks* do tipo somente leitura e leitura/escrita, que facilitam a programação de aplicações que se utilizam do paradigma dos leitores e escritores [75].

A figura D.3 mostra as principais classes envolvidas na implementação do DLM. A classe `DistributedLockManager` implementa o gerenciador em si. Ela estende a classe `java.lang.Thread` podendo ser executada em segundo plano em uma máquina servidora, onde permanecerá recebendo e processando solicitações de travamento e liberações de *locks* em benefício dos seus clientes. Uma única instância de `DistributedLockManager` pode gerenciar diversos *locks* distribuídos.

A classe `DistributedLock` representa os *locks* distribuídos e é utilizada no lado cliente. Cada instância da classe `DistributedLock` pode ser usada por um cliente para obter e liberar um único *lock* distribuído. No construtor da classe, é informado um objeto serializável a ser utilizado para identificar o *lock* desejado. O cliente utiliza os métodos `lock` e `unlock` para, respectivamente, obter e liberar, de forma transparente, o *lock* distribuído. Como argumento na invocações a estes métodos, o cliente deve passar algum objeto serializável utilizado para identificá-lo (identificar o cliente) no gerenciador. Com este identificador, o gerenciador sabe qual o cliente está de posse do *lock* no momento. A implementação dos métodos `lock` e `unlock` utiliza um UDPChannel para se comunicar com o gerenciador de *locks*.

A classe `DistributedReadWriteLock` encapsula duas instâncias de `DistributedLock`. Neste caso, as duas instâncias de `DistributedLock` operam sobre o mesmo *lock* distribuído, porém com semânticas diferentes. Uma das instâncias, é utilizada pelo cliente quando ele deseja operar em modo somente leitura, e a outra, é usada

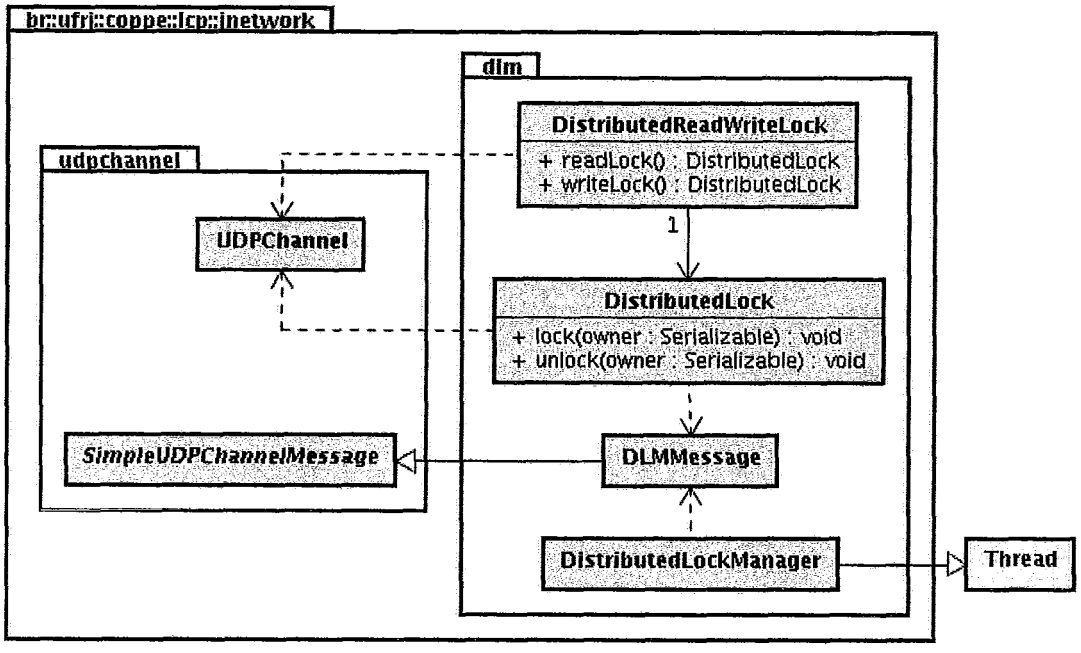


Figura D.3: Classe e interfaces utilizadas na implementação do DLM.

quando ele deseja operar na região crítica protegida pelo *lock* realizando operações de escrita (acompanhadas ou não de operações de leitura)[1].

A listagem D.2 exemplifica a utilização da classe `DistributedReadWriteLock`, assumindo que o gerenciador de *locks* está rodando no servidor "*maquina-dlm*" e aguarda solicitações de *lock* na porta 1234. Nas linhas 1 e 2, o cliente cria[2] e inicializa o canal que será utilizado para comunicação com o gerenciador de *locks*. Em seguida, na linha 4, o cliente instância o objeto `DistributedReadWriteLock` que será usado para operar sobre o *lock* identificado como "*meuLock*". O gerenciador de *locks* passará a gerenciar este novo *lock* automaticamente, assim que a primeira operação sobre ele for solicitada.

Na linha 6, o cliente tenta adquirir o *lock* para, em seguida, realizar operações envolvendo escritas. Ele ficará bloqueado neste ponto até que o DLM lhe conceda a posse do *lock* solicitado. Observe que ele se identifica para o DLM utilizando a *string* "*souEu*"

[1]Essa sistemática é a mesma empregada na classe `ReentrantReadWriteLock` da API JSE.

[2]Observe que o tamanho (512) do *buffer* a ser utilizado pelo canal precisa ser informado pois a ferramenta não tem como prever o tamanho do objetos que serão utilizados para identificar os clientes e os *locks*. Para ajudar o usuário na determinação do tamanho a ser especificado para este *buffer*, a classe `DistributedLockManager` possui o método `getMessageLength`, que devolve o tamanho mínimo necessário para o *buffer* dado um identificador de *lock* e de cliente.

```
1  UDPChannel lockChannel = new UDPChannel("maquina-dlm",1234,512);
2  lockChannel. start ();
3
4  DistributedReadWriteLock drwLock = new DistributedReadWriteLock(lockChannel,"
       meuLock");
5
6  drwLock.writeLock().lock("souEu");
7  // Realiza aqui operacoes envolvendo escritas
8  drwLock.writeLock().unlock("souEu");
9
10 drwLock.readLock().lock("souEu");
11 // Realiza aqui operacoes somente leitura
12 drwLock.readLock().unlock("souEu");
```

Listagem D.2: Exemplo de uso da classe DistributedReadWriteLock do pacote JNetwork.

e depois, na linha 8, libera o *lock* obtido utilizando a mesma identificação. Assim, o gerenciador sabe que este cliente finalizou o uso do *lock* e pode conceder o recurso para outro cliente interessado.

Em seguida, nas linhas 10 e 12, o cliente, respectivamente, adquire e libera o mesmo *lock*, desta vez, realizando operações somente-leitura. Neste caso, mais de um cliente poderiam estar de posse do mesmo *lock*, desde que todos estivessem operando em modo somente-leitura.

Apêndice E

Sysusager

A avaliação experimental do dCache envolve diversas execuções do benchmark TPC-W utilizando uma quantidade variável de máquinas, com endereços também variáveis. Desenvolvemos o monitor *Sysusager* (*System Usage Reporter*) com o propósito de obter informações sobre a utilização de recursos em todas as máquinas envolvidas nas execuções do benchmark TPC-W de forma centralizada e facilmente configurável.

O Sysusager é uma aplicação escrita em Java, configurável pelo usuário através de um arquivo XML. A listagem E.1 traz um exemplo de arquivo de configuração do Sysusager.

```
1  <sysusager>
2      <output>/state/ partition1 /dcache/sysusager-out.ser</output>
3      <mi>220</mi>
4      <delay>60</delay>
5      <preview enabled="true" refresh="60"/>
6      <hosts>
7          <host name="ApacheServer" address="10.10.20.245"/>
8          <host name="PostgresServer" address="10.10.20.231" sshoptions=" -
               l postgres -i /home/dcache/id_rsa"/>
9          <host name="RBE" address="compute-0-14"/>
10         <host name="dCacheServer" address="compute-0-13"/>
11         <host name="tomcat1" address="compute-0-11"/>
12         <host name="tomcat2" address="compute-0-16"/>
13         <host name="tomcat3" address="compute-0-6"/>
14     </hosts>
15 </sysusager>
```

Listagem E.1: Exemplo de arquivo de configuração do Sysusager.

Na linha 2 é defino do nome do arquivo de saída gerado pelo monitor. Nas linhas 3 e 4 é definido que o intervalo de medição durará 220 minutos e que as medições serão

realizadas a cada 60 segundos. Na linha 5, é solicitado que a cada 60 segundos o arquivo de saída do monitor seja atualizado com as informações de monitoramento mais recentes. Este parâmetro é útil para permitir que o usuário acompanhe a utilizações dos recursos sem precisar esperar o término do período de medição definido na linha 3.

Nas linhas 7 até 13 são definas as máquinas a serem monitoradas. Cada linha especifica um nome de máquina (definido a gosto do usuário) e o endereço da máquina. Informações adicionais para a comunicação com a máquina também podem ser especificadas, como por exemplo na linha 8, onde é informado o nome de usuário a ser utilizado para acessar a máquina remota e o arquivo com a chave de autenticação necessária para o acesso.

A saída é gerada em formato de objeto Java serializado. O monitor funciona através de linha de comando e possui um comando para transformar um arquivo de saída por ele gerado em um conjunto de arquivos CSV. O comando gera um arquivo CSV para cada máquina monitorada, contendo a variação da utilização de recursos naquela máquina durante o intervalo de medição.

A implementação do monitoramento das máquinas remotas é realizado com base na classe abstrata `AbstractHostProbe`, que define métodos utilizados pelo monitor para coletar as informações de uso de recursos das máquinas monitoradas de forma transparente. As classes que estendem da classe `AbstractHostProbe` implementando seus métodos abstratos são as responsáveis por definir os detalhes da comunicação e obtenção de recursos a partir das máquinas remotas. Atualmente, a única classe concreta derivada de `AbstractHostProbe` é a `LinuxVmstatHostProbe`.

A implementação `LinuxVmstatHostProbe` se comunica com as máquina monitoradas através de *Secure Shell* (SSH) e obtém as informações de utilização de recursos executando remotamente o utilitário *vmstat*, do Linux. O único recurso monitorado pela implementação atual é a CPU. Entretanto, a implementação está preparada para facilmente disponibilizar informações de uso de memória, obtidas também através do utilitário *vmstat*.