Joana Isabel da Costa Roque

Licenciada em Engenharia Informática

Um estudo de performance de uma ferramenta de *Object/Relational Mapping*

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientador : João Manuel dos Santos Lourenço, Prof. Auxiliar,
Universidade Nova de Lisboa

Júri:

Presidente:	Prof. Vasco Miguel Moreira Amaral Universidade Nova de Lisboa
Arguente:	Prof. João Coelho Garcia Instituto Superior Técnico
Vogal:	Prof. João Manuel dos Santos Lourenço Universidade Nova de Lisboa

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Dezembro, 2013

Um estudo de performance de uma ferramenta de *Object/Relational Mapping*

Ao Guilherme e à Inês

Agradecimentos

O primeiro agradecimento é destinado ao meu orientador, Prof. João Lourenço, por ter aceite orientar esta dissertação, pela paciência e também pelos inestimáveis comentários e conselhos ao longo deste trabalho. Um agradecimento especial também para o Ricardo Dias e para o Tiago Vale por terem a disponibilidade de contribuir com opiniões sobre o trabalho e também pela ajuda que me prestaram quando foi necessário.

Também quero deixar o meu apreço às seguintes instituições pelo apoio financeiro e condições logísticas que ajudaram na realização deste trabalho: Departamento de Informática e a Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa; ao Centro de Informática e Tecnologias de Informação da FCT/UNL; e à Fundação para a Ciência e Tecnologia, no projecto de investigação Synergy-VM (PTDC/EIA-EIA/113613/2009).

Gostaria também de agradecer a todos os docentes do Departamento de Informática da FCT/UNL que durante a minha Licenciatura e o meu Mestrado contribuíram para a minha formação humanística e profissional.

Aos meus colegas que partilharam comigo a sala 254, pelo bom ambiente que houve e pela compreensão mútua nos momentos de frustração.

A todos os amigos que conviveram comigo, em especial nestes dois anos, pelos bons momentos que passámos juntos e pelo apoio nos maus momentos. São eles Nuno Pimenta, Diogo Sousa, Lara Luís, Laura Oliveira, João Silva, Gabriel Marcondes, Andy Gonçalves, João Martins.

Aos meus pais pelo apoio constante e pela paciência quando necessitava de trabalhar até mais tarde. À minha família em geral pelo apoio e ânimo que me deram ao longo do meu mestrado e da minha licenciatura. Um agradecimento especial aos meus padrinhos, Sónia Roque e Mário Costa pelo apoio e pelos bons conselhos que me deram ao longo de todos os meus anos de vida. Para o meu afilhado Guilherme que sem perceber me fez lembrar sempre o objetivo de todo este esforço.

Ao Helder, por toda a coragem, carinho, amor e compreensão que foram necessários para pudéssemos ambos realizar as nossas dissertações com muito mais empenho e dedicação.

Resumo

Atualmente existem inúmeras ferramentas que ajudam no desenvolvimento de aplicações multi-camada, retirando aos programadores a responsabilidade de definir a estrutura e as interações com o sistema de gestão de base de dados que serve de repositório permanente para os dados. Porém, a introdução de mais uma camada entre a aplicação e a base de dados tem necessariamente implicações no desempenho global do sistema. Estes sistemas recorrem por isso a múltiplas estratégias para diminuir o *overhead* imposto pela sua utilização, incorporando múltiplas soluções de *caching*.

O trabalho reportado nesta dissertação visa realizar um estudo para compreender a dimensão das perdas de performance introduzidas pela utilização de ferramentas de mapeamento objeto relacional numa arquitetura multi-camada e o impacto da utilização dos mecanismos de *caching* disponíveis nesses sistemas, tanto num contexto centralizado como num contexto distribuído. Para isso iremos medir a produtividade do *benchmark* TPC-W em duas variantes: uma (*standard*) que trabalha sobre uma base de dados relacional; e outra (*adaptada*) que trabalha sobre um sistema de mapeamento objeto-relacional, suportado também pela mesma de dados relacional. Nesta última variante também será testada a inclusão das várias caches suportadas pela ferramenta Object Relational Mapping (ORM) e avaliar os seus potenciais benefícios para o desempenho da aplicação. Da análise da produtividade das duas variantes do *benchmark*, pretende-se quantificar as perdas de desempenho decorrentes da introdução de mais uma camada entre a aplicação e a base de dados, bem como identificar a origem dessas perdas e potenciais formas de as mitigar.

Palavras-chave: Mapeamento objeto-relacional, Aplicações multi-camada, Hibernate, Persistência de dados

Abstract

Nowadays, there are various tools available to helping in the development of three-tier applications, relieving the software developers from the responsibility of defining the structure and the interactions with the database management system. However, the introduction of an additional software layer between the application and the database necessarily impose an overhead to the system's overall performance. These systems make also use of multiple strategies to reduce the overhead they impose, by incorporating multiple cache elements.

The work reported in this thesis aims at making a study to understand the dimension of the performance overheads introduced by the usage of object/relational mapping tools in an three-tier architecture. For that it will be measured the performance of the benchmark TPC-W in two versions: a standard version, using a relational database; and other version, adapted from the previous one, that uses a object/relational mapping framework, supported with a relational database. The latter version will be also tested to verify how the inclusion of multiple caches supported by the ORM tool can improve the application's performance. From the performance analysis of the two benchmark versions, it is intended to quantify the performance losses resulting from the introduction of an additional layer between the database and the application, and as well identify where the performance losses arise and understand potential ways to mitigate them.

Keywords: Object/Relational Mapping, Three-tier Applications, Hibernate, Data Persistence Mechanisms

Conteúdo

Lista de Figuras

Listagens

1

Introdução

1.1 Motivação

A utilização de linguagens orientadas a objetos para a construção de aplicações tem-se tornado uma escolha frequente para o desenvolvimento de aplicações de raiz. Isto deve-se principalmente à facilidade como se consegue modelar os processos e operações do mundo real, devido às características de organização dos dados do paradigma orientado a objetos. No entanto, estas aplicações necessitam de poder persistir os seus dados de execução, mas de forma a que sejam conservadas todas as propriedades do modelo orientado a objetos.

Atualmente, as aplicações orientadas a objetos têm ao seu dispor uma grande variedade de métodos para tornar os seus dados persistentes. A utilização de bases de dados relacionais é a opção que tem prevalecido como escolha, em detrimento de soluções desenhadas com vista à fácil integração com aplicações que utilizem linguagens com o mesmo paradigma [DU11], como por exemplo as bases de dados orientadas a objetos.

Para que as aplicações orientadas a objetos consigam persistir os seus dados numa base de dados relacional, necessitam de transformar a representação dos dados entre o formato relacional e o formato de objeto usado pela linguagem de programação. Esta transformação nem sempre é óbvia devido à diferença de representação dos objetos em memória e em tabelas de uma base de dados relacional [Ire+09]. Entre os exemplo desta dificuldade estão o facto de os sistemas de base de dados não suportarem de forma nativa a representação de hierarquias de objetos, ou a incapacidade de se garantir o isolamento e o encapsulamento dos dados de uma entidade numa tabela.

Estas dificuldades originadas pelas diferenças entre os formatos de objeto e o relacional são um problema já conhecido o *impedance mismatch* objeto/relacional [Ire+09]. Para

ajudar na resolução do problema de *impedance mismatch* objeto/relacional, surgiram várias ferramentas que reduzem e facilitam a transformação e a estruturação dos dados entre os formatos relacional e objeto. Tais ferramentas classificam-se de ferramentas *object/relational mapping* e encarregam-se de criar a estrutura da base de dados e de fazer a gestão dos objetos da aplicação que são persistidos, transformando os dados da representação interna em memória para o formato relacional, retirando assim aos programadores a responsabilidade de ter que gerir estas transformações entre formatos.

Estas ferramentas ORM são bastante úteis na construção de aplicações multi-camada *three tier*. No entanto, este tipo de arquiteturas pode apresentar alguns problemas de escalabilidade com um número crescente de utilizadores, mais precisamente ao nível da base de dados. Devido ao posicionamento destas ferramentas entre a aplicação e a base de dados é importante que estas consigam adequar-se ao *workload* da aplicação e que utilizem mecanismos para conseguir dar resposta aos pedidos dos utilizadores da maneira mais eficiente possível, de modo a não se tornarem uma limitação ao desempenho do serviço. O que se tem observado, e dependendo da ferramenta em questão, é que as ferramentas ORM não suportam muitos destes mecanismos *out-of-the-box*, desativando-os por omissão e/ou suportando componentes implementados por terceiros. Isto leva a que estas ferramentas sejam mais utilizadas para aplicações que lidam com *workloads* limitados [ONe08].

Surge então a necessidade de se perceber como estes componentes que auxiliam o desempenho se podem combinar entre si para gerar os melhores ganhos de performance em aplicações que utilizem ferramentas ORM. Para isso deve-se analisar as variantes das arquiteturas que são suportadas por estas ferramentas para que sejam detetadas as combinações de componentes que rentabilizam mais a performance.

1.2 Problema

As ferramentas de ORM são bastante populares hoje em dia para se criarem aplicações multi-camada. Porém, se não apresentarem uma arquitetura correta, estas ferramentas podem impor custos de performance que degradam a escalabilidade das aplicações e consequentemente baixam o valor dos serviços oferecidos. Isto torna-se um problema consideravelmente importante quando estas aplicações têm um aumento significativo do número de utilizadores. Esta tarefa torna-se algo complicada quando as aplicações já se encontram instaladas, e normalmente o processo de transição da arquitetura implica que tenham que existir alguns períodos de indisponibilidade do sistema.

Apesar de algumas ferramentas disponibilizarem componentes para aumentar a escalabilidade destes sistemas é difícil verificar quais são os mais apropriados para cada aplicação sem serem realizados alguns testes prévios. No caso específico de uma ferramenta ORM para Java, o Hibernate, existem múltiplas caches disponíveis que podem ser combinadas, de forma a se poder adequar à carga de trabalho da aplicação. Portanto,

para se garantir a escalabilidade de aplicações que utilizem ferramentas de ORM, os programadores têm que fazer pelo menos alguns testes para aferirem se os componentes utilizados podem ou não ser adequados à carga de trabalho das aplicações. Esta tarefa pode implicar aumentos no tempo de desenvolvimento, e consequentemente, agravar o custo de desenvolvimento das aplicações multi-camada que utilizem ferramentas ORM.

Identificar as falhas de performance que persistem nas arquiteturas de ferramentas ORM abre a possibilidade a serem encontradas soluções de escalabilidade adequadas para aplicações multi-camada, e desta maneira reduzir o tempo e o custo de produção destas aplicações.

1.3 Abordagem

Este trabalho pretende estudar uma ferramenta ORM que é bastante popular para aplicações Java, o Hibernate, de modo a se detetar as melhores configurações para a sua arquitetura, de acordo com os vários *workloads* possíveis aplicacionais. Este estudo de performance irá recorrer ao *benchmark* TPC-W [Cou01] que representa o *workload* de uma aplicação multi-camada padrão. Este *benchmark* será adaptado para utilizar a ferramenta Hibernate e serão efetuados testes comparativos com combinações de várias soluções arquiteturais possíveis de se utilizar com o *benchmark*. Serão realizados testes com uma arquitetura que não utilize ORM, e só utilize uma base de dados, com uma arquitetura que utilize o Hibernate mas sem a ativação de caches opcionais, e com arquiteturas que utilizem as várias caches disponibilizadas pelo Hibernate.

Os resultados serão analisados e estudados para se perceber, primeiro, o impacto na performance causados pela utilização de ferramentas de ORM, e depois, como as diferentes combinações de componentes nas ferramentas ORM podem beneficiar ou não o desempenho do sistema.

1.4 Contribuições

As contribuições deste trabalho são as seguintes:

- Uma demonstração do processo de adaptação de aplicações que usam bases de dados relacionais e SQL através de uma API JDBC ao sistema ORM Hibernate;
- Avaliação do comportamento de um sistema ORM (Hibernate) perante diferentes *workloads* e diferentes configurações de arquitetura, designadamente, a comparação entre o desempenho quando o Sistema de Gestão de Base de Dados (SGBD) está a ser executado na mesma máquina que a aplicação e quando está a ser executado numa máquina diferente;
- A identificação dos componentes que são suportados pelo Hibernate que mais beneficiam mais o desempenho das aplicações multi-camada;

- Uma versão do TPC-W adaptada para utilizar o Hibernate, estando disponível para a comunidade científica;

1.5 Visão Geral

Neste capítulo foi apresentado o tema deste trabalho, uma possível abordagem para os problemas que foram identificados e também as contribuições deste trabalho.

No capítulo 2 serão resumidos todos os tópicos de trabalho relacionado com este trabalho. Primeiro, serão apresentados alguns *benchmarks* que têm sido utilizados para medir o desempenho dos sistemas de gestão de bases de dados relacionais e orientadas a objetos. De seguida, serão apresentados trabalhos que fazem uma análise de resultados destas *benchmarks*, ou outras criadas especificamente em cada trabalho, comparando os resultados obtidos quando se utilizam ferramentas ORM nas aplicações com os resultados quando se utilizam bases de dados, quer relacionais, quer orientadas a objetos. Depois são apresentadas algumas técnicas que têm sido utilizadas para melhorar o desempenho de aplicações multi-camada e finalmente será descrita a ferramenta ORM que será utilizada neste trabalho, o Hibernate, fazendo-se um resumo das suas funcionalidades atuais mais relevantes neste trabalho.

No capítulo 3, será apresentada uma descrição mais detalhada das características do *benchmark* TPC-W e também como decorreu o processo de adaptação de uma implementação deste *benchmark* para que utilize o sistema Hibernate, explicando as modificações que foram efetuadas. De seguida, no capítulo 4 apresentam-se todos os testes efetuados para avaliar o Hibernate, e faz-se uma análise dos resultados obtidos. Finalmente, no capítulo 5 apresentam-se as conclusões e o trabalho futuro que decorrem deste trabalho.

2

Trabalho Relacionado

Para haver uma compreensão correta do âmbito deste trabalho, foi necessário estudar outros trabalhos que estão relacionados com este tema. Este capítulo apresenta ao leitor a síntese dos temas e de outros trabalhos estudados no decorrer deste trabalho.

Primeiro será apresentado na secção 2.3 o estado da arte relativo às *benchmarks* existentes para avaliar as bases de dados relacionais e orientadas a objetos. Depois, na secção 2.4, serão apresentados alguns trabalhos que realizaram estudos de performance de ferramentas ORM, em que também se efetua uma comparação de performance com outros métodos de persistência disponíveis para estas aplicações. De seguida, na secção 2.1, também serão abordados alguns trabalhos que apresentam diferentes técnicas que podem ser utilizadas para melhorar o desempenho de aplicações multi-camada, não necessariamente construídas com ferramentas ORM. Finalmente, na secção 2.2, será feita uma apresentação detalhada da ferramenta ORM que será estudada neste trabalho, o Hibernate.

2.1 Melhorias de Performance em Aplicações Multi-Camada

Como já foi referido na introdução, o aumento drástico do número de utilizadores nas aplicações multi-camada pode trazer alguns problemas de escalabilidade que estão relacionados com o aumento de carga de pedidos à camada de persistência. Se esta camada de persistência for uma base de dados relacional, e como os acessos à base de dados tendem a ser demorados, este aumento do número de pedidos à base de dados pode causar grandes limitações ao desempenho geral da aplicação.

Têm surgido algumas soluções para resolver estes problemas de escalabilidade que se baseiam em retirar ou equilibrar o número de pedidos à base de dados. No entanto,

verifica-se que algumas destas soluções ainda não tiveram a adesão devida nas ferramentas ORM, e no caso do Hibernate é apenas disponibilizado suporte à integração de componentes externos que implementam algumas destas soluções. É importante perceber então estas soluções para se tentar perceber com estas poderiam beneficiar o desempenho das aplicações que utilizam as ferramentas ORM.

2.1.1 Replicação

A solução mais simples para se melhorar o desempenho de aplicações multi-camada é a replicação do código ou dos dados da aplicação.

A solução de replicação mais comum é a replicação do código da aplicação [Siv+07], com cada cópia, ou nó, a aceder a uma base de dados central. Se as réplicas da aplicação estiverem geograficamente distantes da base de dados, a utilização de uma base de dados centralizada pode causar um *bottleneck* e desta maneira aumentar a latência em geral.

Este *bottleneck* pode ser solucionado se a base de dados for também replicada por vários nós, associando-se cada um destes nós a uma réplica da aplicação [Siv+07]. Cada réplica da aplicação pode desta maneira aceder à réplica da base de dados que lhe é associada, ou no caso de indisponibilidade desta, aceder à que lhe está mais próxima. Esta solução requer um *middleware* que possa garantir a sincronização das modificações dos dados entre as várias réplicas da base de dados [Siv+07].

2.1.2 Caches

As soluções que utilizam caches baseiam-se em guardar em memória os dados que são utilizados mais frequentemente, e desta maneira reduzir os acessos à base de dados. Como as consultas à base de dados podem ser por vezes demoradas, a utilização de caches pode melhorar drasticamente a performance em geral da aplicação, se for corretamente aplicada.

Pode se colocar em cache os registos da base de dados utilizados pelas consultas efetuadas com mais frequência [Siv+07], que depois são obtidos dinamicamente a partir da base de dados, à medida que as consultas não conseguem ser efetuadas com os registos presentes na cache. No entanto, a alternativa mais utilizada é a de colocar na cache os resultados às consultas, que são guardados em memória, de forma a ficarem independentes entre si.

A utilização dos resultados das consultas para colocar em cache pode trazer vantagens, se os pedidos utilizarem frequentemente o mesmo subconjunto de dados, mas a utilização dos registos da base de dados pode ser mais eficiente para aplicações que tenham outros padrões de trabalho [Siv+07].

Quando surgiram inicialmente, as caches eram muito simples, apenas guardando os dados da base de dados através de uma interface do género *GET/PUT*. À medida que a complexidade das aplicações e o número de utilizadores aumentou, os sistemas de *caching* que eram utilizados apresentavam algumas limitações. Soluções como *memcached*

apresentavam alguns problemas aos programadores, pois tinham uma interface fraca e não tinham suporte, por exemplo, a transações [Por+10].

Foram então desenvolvidas outras soluções, de maneira a se resolverem os problemas de concorrência de modificações e também de maneira a oferecer aos programadores uma interface mais adequada à complexidade das aplicações. Algumas dessas novas soluções são apresentadas de seguida.

2.1.2.1 Caches *Thread-Safe*

De maneira a tirar partido dos benefícios da utilização mecanismos de programação concorrente nas aplicações, as caches devem oferecer suporte ao processamento de múltiplos pedidos concorrentes. Se uma aplicação é dividida em múltiplos processos, e se esses processos não podem interagir com a cache de maneira concorrente, a cache irá limitar a performance e podem ocorrer algumas anomalias no estado dos dados, originadas das modificações concorrentes nos elementos em cache.

Um exemplo de uma cache que oferece suporte à utilização por múltiplas *threads* é apresentada em [VBZ08]. Esta solução consiste numa cache, que pode também ser utilizada como base para outras caches, que contém os dados indexados por chave, e pode receber pedidos concorrentemente e assincronamente.

Esta cache tem três componentes básicos: uma interface para a rede, as *threads* que executam os pedidos (ou *threads* trabalhadoras) e uma *datastore*. A interface de rede consiste numa única *thread* que recebe os pedidos e designa cada um a uma das *threads* trabalhadoras, utilizando para isso uma fila de pedidos *thread-safe*. O número de *threads* trabalhadoras pode ser parametrizado.

A *datastore* é o componente principal desta cache, e consistem tem duas estruturas de dados guardadas em memória.

A primeira destas estruturas de dados é uma tabela de *hash*, com tamanho fixo, de árvores vermelhas e pretas, com cada destas árvores protegidas por um *reader-writer lock*. Cada elemento da árvore vermelha e preta consiste num par chave-valor. Esta estrutura tem como objetivo ser o principal arquivo dos dados da cache.

A segunda estrutura de dados vai auxiliar a manipulação dos elementos da cache em grupos e consiste numa árvore vermelha e preta onde cada elemento é um par que consiste num marcador, como chave e numa raiz de uma árvore vermelha e preta, como valor. Cada árvore vermelha e preta desta estrutura contém os apontadores para os elementos da primeira estrutura de dados que são associados com o marcador da chave. Estas árvores estão também protegidas por um *reader-writer lock*.

A utilização de *reader-writer locks* permite que as *threads* que só executem operações de leitura nunca bloqueiem com outras *threads* que tenham o mesmo tipo de comportamento.

2.1.2.2 Caches Transaccionais

Em aplicações que utilizam transações para consultar e modificar a base de dados, é importante que as caches tenham mecanismos que possam guardar os dados de maneira a que durante cada transação, os dados que são obtidos e guardados na cache estejam consistentes com o estado da base de dados na *timestamp* dessa transação, de acordo com as propriedade de isolamento garantidas pelo SGBD. As caches também precisam de oferecer mecanismos que indexem a cache de acordo com o identificador da transação ou a sua *timestamp*, e precisam de receber notificações quando os dados são modificados na base de dados, de maneira a que os dados que ficaram desatualizados sejam propriamente removidos da cache.

Um exemplo de uma cache que implementa estas funcionalidades é a TxCache [Por+10]. A TxCache é uma cache transacional replicada. A aplicação não comunica diretamente com as replicas da cache, mas utiliza a API oferecida pela cache. Porém, este sistema apresenta outra abordagem no tipo de dados que são colocados na cache. Em vez de se colocar em cache os registos da base de dados ou os resultados das consultas mais efetuadas, a TxCache guarda resultados de funções da aplicação. Estas funções necessitam de ser puras, isto é, as funções não podem ter efeitos secundários, e devem sempre depender apenas dos seus argumentos e do estado da base de dados. O programador apenas necessita de assinalar as funções que os resultados devem entrar na cache e a TxCache vai automaticamente acompanhar as invalidações dos dados de que as funções dependem.

Para transações apenas de leitura a TxCache tem a responsabilidade de manter a consistência dos dados, já que essas transações apenas vão consultar dados já em cache. Para transações de leitura e escrita a consistência é assegurada pela base de dados, e desta maneira não se introduzem mais anomalias de concorrência.

Como há a possibilidade das transações consultarem dados ligeiramente desatualizados, o programador tem que definir um intervalo de tempo em que os dados possam ficar desatualizados, no qual as transações têm que ser capazes de processar sem erros os dados que estejam desatualizados.

A verificação da validade dos resultados é feita no SGBD, que precisa de ser modificado para que se possam acompanhar as modificações dos dados e propagar notificações de invalidação para os nós da cache, de maneira que sejam retirados da cache os resultados que utilizem estes registos da base de dados que foram alterados.

2.1.3 Otimizações no Controlo de Concorrência

A utilização de caches não é a única maneira de melhorar a eficiência de aplicações multi-camada, e atualmente estão a surgir muitas outras abordagens, sempre com o objetivo de se retirar alguma carga de trabalho da base de dados. Estas novas abordagens estão mais focadas em otimizar o controlo de concorrência como uma maneira de melhorar a performance das aplicações multi-camada.

A ideia principal é retirar a responsabilidade do controlo das operações concorrentes à

base de dados, pois por vezes é necessário utilizar mecanismos demorados, prejudicando a performance em geral. Como há mecanismos em memória similares aos utilizados pela base de dados, como a memória transacional e a memória transacional por software, alguns trabalhos [CFG09; Car+08] tentam explorar os benefícios de se utilizarem estes mecanismos de controlo de concorrência em memória, em vez de serem utilizados nas bases de dados.

A memória transacional é uma tecnologia recente para controlo de concorrência que é baseada nos mecanismos transacionais já existentes nas bases de dados para controlar os acessos concorrentes. O controlo é feito com recurso a técnicas otimistas, onde as transações são executadas livremente e espera-se que não haja muitos conflitos. Quando ocorre um conflito entre duas transações, é escolhida uma, que é abortada, sendo os seus efeitos revertidos [ST95].

A abordagem mais básica é simplesmente remover o controlo de concorrência da base de dados para a memória, utilizando memória transacional por software. Normalmente as bases de dados utilizam mecanismos de *locking* para executar o controlo da concorrência entre as transações. Como isto pode causar demoras e *deadlocks* inesperados, adicionando ao facto das transações na base de dados serem operações que consomem muito tempo, a performance da aplicação em geral pode ficar prejudicada. A solução requer que as transações que eram executadas na base de dados passem a ser executadas através de memória transacional por software, mas esta mudança nem sempre é fácil e é muito dependente nas características das ferramentas que são escolhidas para construir as aplicações.

Em [CFG09] é apresentado uma solução que é especificamente desenhada para se utilizar com a *framework* Java EE.

Em Java EE os pedidos são processados por entidades específicas chamadas *beans*. São utilizados *beans* diferentes para cada pedido de interação diferente, mas para o acesso concorrente aos dados só é admissível utilizar um *bean*, que é o que se utiliza para aceder à base de dados. Neste caso, isso significa que o controlo da concorrência é executado com os mecanismos oferecidos pelo SGBD, o que pode por vezes provocar *deadlocks* ou inesperadamente impor demoras na execução de operações.

Esta solução propõe a utilização de outro tipo de *bean*, o *TMBean*, que executa o controlo de concorrência nos acessos a dados utilizando memória transacional e desta maneira já não se depende dos mecanismos da base de dados.

Este *bean* tem uma estrutura de dados especial, implementada utilizando objetos de memória transacional por software, que representa o estado partilhado entre as *threads* da aplicação. Quando uma *thread* executa uma operação num TMBean, cria-se uma transação de memória para englobar essa operação.

Outras soluções são desenhadas com certas características na arquitetura de modo a tirar partido dos benefícios da replicação da aplicação em muitos nós. É necessário também uma maneira que os diferentes nós possam comunicar entre si e coordenar as

suas operações nos dados de modo a evitar inconsistências e modificações perdidas.

Alguns trabalhos apresentam por isso alguns esforços em utilizar memória transacional para coordenar os dados partilhados em aplicações que utilizem vários nós simultaneamente.

Em [Car+08] é apresentada uma solução que foi construída especificamente para melhorar a escalabilidade e performance de uma aplicação web já implementada. Essa solução é baseada em utilizar um componente já existente construído com memória transacional por software para gerir as transações nos vários nós, em vez de cada nó verificar os dados da sua cache individualmente, fazendo chamadas adicionais à base de dados. Esta verificação de consistência na base de dados era a fonte das limitações de performance neste sistema, conforme foi detetado no trabalho.

Nesta nova arquitetura, os nós partilham as modificações que efetuam aos dados através de comunicação atómica, e desta maneira assegura-se que cada um recebe a informação das modificações e os objetos que cada nó utiliza permanecem consistentes.

2.1.4 Modelo Unificado

Existem algumas operações que não podem ser executadas dentro de uma transação de memória, ou por não conseguirem ser revertidas ou por essa reversão ter algum custo na performance. Exemplos de operações deste tipo são transações de base de dados e instruções de *input/output*. Alguns sistemas de memória transacional proíbem a execução de operações como estas e os sistemas que as permitem executam-nas em exclusão mútua, limitando desta maneira a concorrência da aplicação. No entanto, as transações de base de dados são um caso especial. Se uma transação da base de dados tentar fazer *commit* dentro de uma transação de memória, no caso da transação de memória abortar, faz sentido que a transação da base de dados aborte também.

Uma solução para este problema é apresentada em [DL09]. A ideia é criar um modelo unificado de transações de memória e de base de dados. Este modelo permite que as transações da base de dados possam ser utilizadas em conjunto com transações de memória, preservando as propriedades ACI para as transações de memória e as propriedades ACID para as transações de base de dados. Para garantir totalmente o isolamento, ambos sistemas transacionais devem correr no mesmo nível de isolamento, que deve ser o mais restritivo dos dois sistemas. Os escalonamentos das transações também devem ser iguais nos dois sistemas.

Foi criada um tipo de transação, que o programador deve utilizar quando necessita de conjugar transações de base de dados e transações de memória. O *commit* com os dois sistemas necessita de ser atómico, isto é, se a transação de memória abortar a transação da base de dados que estiver englobada na anterior também tem que abortar. Por isso é utilizado o protocolo de *commit* por duas fases, de modo a se garantir a atomicidade da operação de *commit* entre as transações de memória e as da base de dados. Desta maneira o Modelo Unificado garante que se uma transação aborta, a outra aborta também.

O Modelo Unificado pode ser utilizado para melhorar o controlo de concorrência em aplicações multi-camada, utilizando as transações de memória para executar as operações, cujos resultados são persistidos com as transações de base de dados. Desta maneira são explorados os benefícios da velocidade das transações de memória para controlar as modificações concorrentes nos dados, mas mantendo um estado coerente entre a base de dados e os dados da aplicação guardados em memória.

2.2 Hibernate

Atualmente as aplicações orientadas a objetos podem usar uma variedade de mecanismos para persistir os seus dados. Porém, a opção que tem vindo a ser mais escolhida é a de utilizar bases de dados relacionais, apesar do aparecimento de bases de dados orientadas a objetos, que por usarem o mesmo paradigma e guardarem os dados no formato de objeto seriam mais adequadas a este tipo de aplicações.

A generalização da utilização de Sistema de Gestão de Base de Dados Relacional (SGBDR) está relacionada com a popularidade das técnicas de *data mining* e de *data warehousing* para analisar os dados aplicacionais, pois este tipo de ferramentas requer que os dados estejam estruturados num formato relacional. Para as aplicações, no entanto, isto faz com que os dados estejam constantemente a ser transformados do formato relacional para a sua representação em objeto e vice-versa, o que pode causar alguns problemas durante a execução destas aplicações. Estas dificuldades são um problema já conhecido, denominado de *O-R impedance mismatch*, que se refere às diferenças que existem entre o acesso orientado ao conjunto e declarativo aos dados nos SGBD, e o acesso imperativo, de um elemento de cada vez, aos dados durante a execução de um programa [DU11].

As ferramentas ORM surgiram para resolver este problema, sendo responsáveis pelo mapeamento e transformação dos dados das aplicações para uma representação relacional que possa ser armazenada na base de dados, e desta maneira aliviando a responsabilidade destas tarefas propícias a erros dos programadores.

Existem muitas ferramentas ORM para a linguagem Java, mas atualmente a mais utilizada é o **Hibernate**. O Hibernate é uma ferramenta ORM que é baseada em Java EE, e o seu objetivo é ser o menos intrusiva possível para os programadores, sem deixar de ser altamente configurável, com suporte a muitos componentes de terceiros, desde a *pool* de conexões para a base de dados até aos mecanismos de *logging*.

2.2.1 Arquitetura

O Hibernate é uma ferramenta muito flexível, e como tal pode ter muitas configurações possíveis, adequadas às necessidades específicas de cada aplicação. Como tal, não é viável detalhar todas essas combinações possíveis de componentes.

No entanto, a figura 2.1 representa uma visão geral da arquitetura do Hibernate.

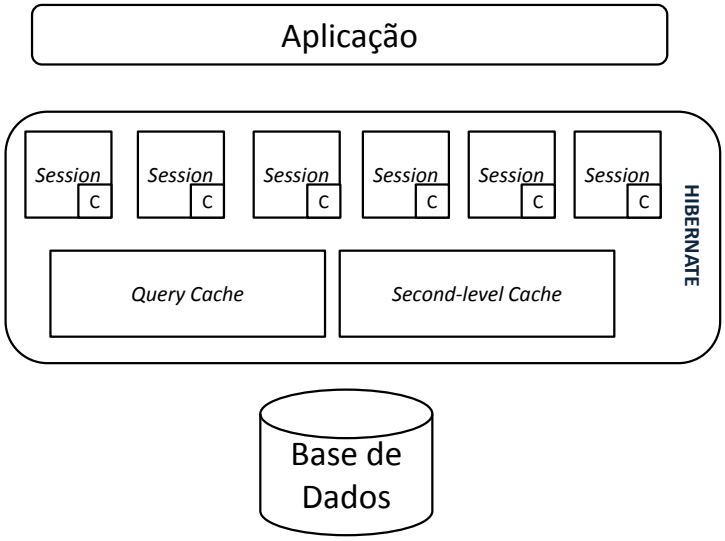


Figura 2.1: Uma vista geral da arquitectura do Hibernate.

Como se pode observar na figura, o Hibernate constitui uma camada entre a aplicação e a base de dados, abstraindo os programadores do esquema da base de dados e os detalhes de representação das entidades que são persistidas.

Na execução de uma aplicação com Hibernate vão sendo criadas instâncias do objeto *Session*, que agregam um conjunto de operações sobre objetos persistentes. Cada instância *Session* tem associada uma cache (na figura 2.1 é representada por um "C"), que contêm a representação dos objetos persistentes que são utilizados na correspondente *Session*. A secção 2.2.2 oferece mais alguns detalhes acerca da função que o objeto *Session* tem na execução das aplicações com Hibernate.

O Hibernate também oferece suporte à utilização de outro tipo de caches, como a cache para as consultas à base de dados que são mais efetuadas durante a execução da aplicação (identificada como "Query Cache" na figura 2.1). Também existe suporte para outra cache para entidades, que contém as instâncias dos objetos persistentes mais utilizadas durante a execução da aplicação (identificada como "Second Level Cache", na figura 2.1).

Para se integrar o Hibernate na aplicação é necessário também um ficheiro XML com as configurações gerais para a aplicação, especificamente chamado de `hibernate.cfg.xml`. Este ficheiro deve ser incluído no *classpath* da execução da aplicação. Para além deste ficheiro, é também necessário fornecer informação adicional para que o Hibernate possa saber que elementos dos objetos persistentes correspondem que tabelas na base de dados. Essa informação pode ser fornecida de duas maneiras, ou através de anotações no código Java de cada classe, ou através de ficheiros XML específicos a cada classe. Esta informação sobre os objetos é detalhada na secção 2.2.3. O ficheiro de configuração do Hibernate também tem que conter a indicação de que classes representam entidades a

serem persistidas.

2.2.2 Componentes da execução

Existem vários componentes do Hibernate que são utilizados na execução de uma aplicação. Esta secção pretende dar uma apresentação geral de cada um desses componentes, com alguns deles a serem mais detalhados noutras secções ao longo deste documento.

Os dados utilizados pela aplicação e que necessitam de ser persistidos utilizando o Hibernate são denominados de objetos persistentes ou entidades. Para se especificar uma entidade, o programador precisa de estruturar os dados da entidade numa classe Java e depois fornecer alguma informação adicional sobre cada elemento dessa classe. O Hibernate vai então usar essa informação para traduzir a estrutura dos objetos que são gerados pela classe, para uma estrutura equivalente na base de dados. Esta informação pode ser dada através de um ficheiro XML específico para cada classe, ou através de anotações no código Java. A secção 2.2.3 apresenta mais detalhes acerca deste processo.

O componente que tem um contacto mais direto com a aplicação são os objetos da classe *Session*. Estes objetos representam um conjunto de interações entre a aplicação e a SGBD, têm uma única *thread* e normalmente têm um tempo de vida curto. Todas as operações nas entidades geridas pelo Hibernate têm que ser efetuadas depois de ter sido criada uma *Session*, através do método `SessionFactoryUtil.getSessionFactory().openSession()`, que retorna um objecto *Session*. Depois de serem especificadas as operações nas entidades, deve encerrar-se a execução da *Session* com o método `Session.close()`, pois pelo comportamento dos mecanismos de *caching* da *Session*, não é conveniente ter a *Session* aberta durante muito tempo.

A *Session* vai guardando as conexões com o SGBD e vai criando instâncias do objeto *Transaction*, à medida que vão sendo criadas interações com a base de dados. Estes objetos *Transaction* vão englobar as interações com a base de dados e representam uma abstração aos mecanismos transacionais utilizados pelo Hibernate (transações Common Object Request Broker Architecture (CORBA), Java Database Connectivity (JDBC) e Java Transaction API (JTA)).

A utilização de *Transaction* é opcional para operações só de leitura (apesar de ser recomendado), mas é sempre obrigatória para operações de escrita. Normalmente, o padrão de interação utilizado é o se criar uma instância *Transaction* no tempo de execução de uma instância *Session*.

Cada *Session* é criada por um objeto único, denominado de *SessionFactory*, que contém toda a informação relacionada com uma determinada base de dados, incluindo, entre outras coisas, meta-dados sobre o processo de ORM e as informações sobre a conexão da base de dados. Cada *Session* é criada pela *SessionFactory* a pedido das *threads* que respondem aos pedidos dos utilizadores, sendo o objeto *Session* apenas associado a um único pedido. Deste modo, uma instância *Session* não pode ser trocada ou reutilizada por outras *threads*.

A *SessionFactory* pode conter outras caches que são úteis para manter o estado de entidades de várias instâncias *Session*, como detalhado na secção 2.2.5. Todos os conteúdos da *SessionFactory* são imutáveis ao longo da execução.

O código 2.1 apresenta um exemplo de uma aplicação básica que utiliza Hibernate, onde se pode observar a utilização dos objetos *Session*, *Transaction* e *SessionFactory*.

Listagem 2.1: Um exemplo de código de uma aplicação que utiliza Hibernate

```
1  import org.hibernate.Session;
2  import org.hibernate.Transaction;
3
4  public class TestPerson {
5
6      public static void main(String[] args) {
7          Session session =
8              SessionFactoryUtil.getSessionFactory().getCurrentSession();
9          createPerson(session);
10         queryPerson(session);
11     }
12
13     private static void queryPerson(Session session) {
14       session = SessionFactoryUtil.getSessionFactory().openSession();
15       Transaction tx = session.beginTransaction();
16       Person p = (Person) session.load(Person.class, new Long(1));
17
18       System.out.println("NOME: "+p.getName()+" ADDRESS:"+p.getAddress());
19
20         tx.commit();
21         session.close();
22     }
23
24     public static void createPerson(Session session) {
25       Transaction tx = session.beginTransaction();
26         Person person = new Person();
27
28         person.setName("Josefina");
29         person.setSurname("Andrade");
30         person.setAddress("Caparica");
31
32         session.save(person);
33
34         tx.commit();
35         session.close();
36     }
37 }
```

2.2.3 Entidades e Objetos Persistentes

As entidades são os blocos em que são construídas as aplicações que utilizam o Hibernate e podem ser definidas com os dados que são persistidos no SGBD, tendo uma representação num objeto Java.

Uma entidade é estruturada na base de dados utilizando informação sobre a sua representação em objeto, definida pelo programador através de uma classe Java e de anotações sobre essa classe, ou de um ficheiro XML específico a essa classe. Com esta informação, o Hibernate pode então, através da representação de uma entidade na base de dados, criar o objeto Java que a representa (e vice-versa), para que os dados possam ser manipulados pela aplicação.

O código 2.2 representa um objeto persistente com as anotações que correspondem aos seus dados e o código 2.3 representa o ficheiro XML que poderia ser utilizado alternativamente às anotações.

Listagem 2.2: Exemplo da definição de um objeto persistente

```
1
2  @Entity
3  @Table(name="Person")
4  public class Person {
5
6    @Id
7    @Column(name="ID")
8    @GeneratedValue
9    Long id;
10
11   @Column(name="NAME",nullable=false,length=16)
12   String name;
13
14   @Column(name="SURNAME",nullable=false,length=16)
15   String surname;
16
17   @Column(name="ADDRESS",nullable=false,length=16)
18   String address;
19
20   public Long getId() {
21     return id;
22   }
23
24   public String getName() {
25     return name;
26   }
27
28   public String getSurname() {
29     return surname;
30   }
31
32   public String getAddress() {
33     return address;
34   }
35 }
```

Na base de dados, o exemplo apresentado no código 2.2 será transformado na base de dados para uma tabela *Person* com quatro colunas: ID, NAME, SURNAME e ADDRESS.

Listagem 2.3: Exemplo do mapeamento de um objeto persistente utilizando XML

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
  <class name="com.sample.Person" table="Person">
    <id name="id" column="ID">
      <generator class="native" />
    </id>
    <property name="name">
      <column name="NAME" length="16" not-null="true" />
    </property>
    <property name="surname">
      <column name="SURNAME" length="16" not-null="true" />
    </property>
    <property name="address">
      <column name="ADDRESS" length="16" not-null="true" />
    </property>
  </class>
</hibernate-mapping>
```

2.2.3.1 Estados dos Objetos Persistentes

Os objetos persistentes podem ter vários estados, de acordo com a sua associação naquele momento a uma instância de uma *Session*. A figura 2.2 apresenta um esquema representativo das possíveis mudanças de estado dos objetos persistentes.

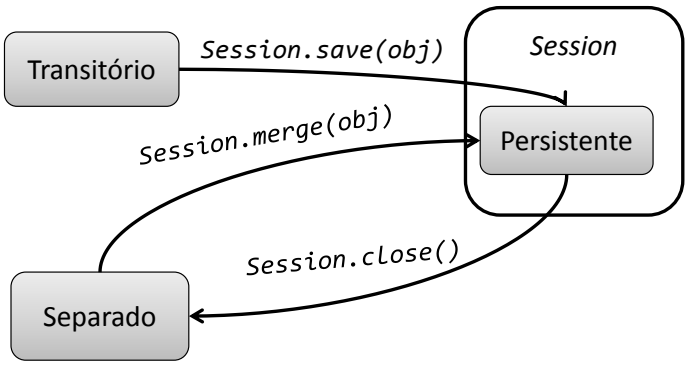


Figura 2.2: Transições entre os estados de um objeto persistente

Se um objeto tiver sido criado, mas ainda não tiver sido associado a uma *Session*, nem ter uma representação na base de dados, está no estado *transitório*. Se um objeto no estado transitório não for associado a nenhuma *Session* os mecanismos de *garbage collection* vão acabar por destruí-lo.

Quando um objeto é associado a uma *Session* passa ao estado de *persistente*. O Hibernate vai então verificar as mudanças ocorridas nesse objeto e vai persistir essas mudanças para a base de dados quando a *Transaction* corrente é *committed*.

Quando a *Session* do objeto termina, o objeto passa ao estado *separado*. A referência do objeto ainda fica válida e o objeto pode ser associado a outra instância *Session*, mudando outra vez o estado para persistente.

O Hibernate gere os objetos pelo seu estado, o que significa que os programadores apenas necessitam de pensar em termos dos estados dos objetos, em vez de pensarem em termos de consultas e estruturas de base de dados. Para um programador que está mais habituado a desenvolver aplicações orientadas a objetos é mais fácil pensar em termos de estados de objetos do que criar estruturas num paradigma diferente, em que tem menos prática, como o paradigma relacional - SQL.

2.2.4 Mecanismos de Concorrência

Apesar do Hibernate oferecer suporte para mecanismos de concorrência pessimista e otimista nos objetos da aplicação, todas as interações com a base de dados, isto é, todas as alterações a objetos persistentes, têm que ser englobadas numa transação e é o único modo possível para se interagir com a base de dados numa aplicação utilizando Hibernate.

Para a gestão das transações, o Hibernate utiliza as transações da interface JDBC e a API JTA. Como tal, o programador deve esperar o mesmo comportamento transacional que é definido pelo nível de isolamento da base de dados, pois o Hibernate não introduz outras anomalias de concorrência.

O Hibernate oferece controlo de concorrência otimista por adicionar versões aos objetos persistentes, mas a única coisa que o Hibernate se encarrega é a de ir incrementando o número de versão quando ocorrem modificações num objeto. Os programadores devem ser então capazes de gerir eles mesmos as versões, já que o Hibernate não faz qualquer tipo de controlo ou gestão automática de alterações nos objetos com versões.

O controlo de concorrência pessimista que o Hibernate oferece utiliza os mecanismos de *locking* suportados pela base de dados. São suportados vários modos de *locking* de acordo com os modos oferecidos pela base de dados, e o programador deve especificar o modo de *locking* que pretende usar no ficheiro `hibernate.cfg.xml`. Os mecanismos de *locking* devem ser utilizados com cuidado, pois isso impõe uma perda de performance adicional por ser a base de dados a fazer *lock* às entidades e gerir as suas modificações.

2.2.5 *Caching*

Para aliviar um pouco a base de dados de alguma carga de trabalho, o Hibernate oferece suporte a mecanismos de *caching* em três componentes: *first-level cache*, *second-level cache* e *query cache*.

A *first-level cache* é localizada dentro da *Session* e é utilizada por todas as aplicações que usam Hibernate, de forma transparente para o programador, desde que sejam utilizadas no decorrer do tempo de vida da *Session* as funções corretas. Como esta cache é a única que todas as aplicações com Hibernate utilizam, na secção 2.2.5.1 encontra-se uma

explicação mais detalhada do seu funcionamento.

A *query cache* é localizada na *SessionFactory* e é utilizada para guardar os resultados das consultas mais frequentes à base de dados.

Como esta cache precisa de manter informação sobre as modificações nas entidades de base de dados, para que as respetivas entradas na cache possam ser invalidadas, são causadas algumas percas de performance. Consequentemente os ganhos de performance da utilização desta cache dependem das vezes que os dados são alterados no tempo de execução, mas geralmente são muito limitados.

A secção 2.2.5.2 apresenta mais detalhes sobre o funcionamento desta cache.

Porém, a cache opcional que é mais utilizada é a *second-level cache*. A *second-level cache* é também localizada na *SessionFactory* e é oferecido suporte para versões implementadas por outras entidades. O programador pode utilizar as implementações que são distribuídas com o Hibernate, utilizar outras implementações, ou até mesmo construir uma implementação sua, apesar de ser recomendado pela equipa que desenvolve o Hibernate que sejam utilizadas implementações fornecidas por outras entidades.

Como os objetos dentro da *second-level cache* podem ser representados de maneira diferente ou conter alguns dados adicionais, é necessário transformar esses objetos para uma forma que o Hibernate possa manipular. Portanto é possível que a utilização da *second-level cache* imponha alguma perda de performance. Na secção 2.2.5.3 encontram-se mais detalhes sobre o funcionamento de algumas caches que são distribuídas juntamente com o Hibernate.

Tanto a *query cache* como a *second-level cache* são de utilização opcional, e cada uma se adequa a um tipo de trabalho específico. A sua utilização pode ser combinada para dar o máximo de rendimento possível para a aplicação.

2.2.5.1 First Level Cache

A *first level cache* é utilizada para guardar objetos que estejam no escopo de uma instância *Session*. Está localizada e é gerida por um componente na *Session* denominado de *PersistenceContext*. O componente *PersistenceContext* representa o estado das entidades e das coleções que o Hibernate mantém dentro da *Session*.

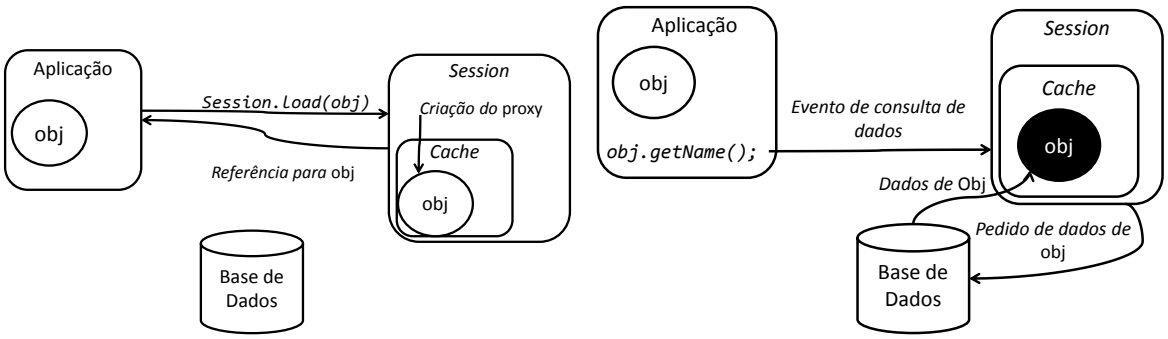
Esta cache guarda os objetos utilizados pela *Session* à medida que o Hibernate constrói a sua representação a partir das entidades da base de dados, ou quando um objeto no estado separado é associado com a *Session*. Depois dos objetos estarem na cache, esta vai mantendo informação sobre as alterações feitas aos objetos durante o tempo de vida da *Session*.

Duas instâncias *Session*, em execução simultânea, podem ter nas suas *first level cache* referências para objetos que representem a mesma entidade, porém, essas referências são necessariamente diferentes. Os objetos são colocados na cache automaticamente, com a utilização de métodos específicos da *Session*. Estes métodos normalmente são os utilizados para obter ou aceder a entidades persistentes (*load()*, *save()*, entre outros).

As entidades que estão em cache só podem ser modificados dentro de uma *Transaction*, e depois do *commit* essas alterações são propagadas para a base de dados, a partir da utilização do método *Session.flush()*. Os objetos permanecem na cache, de modo a poderem ser utilizados durante outra *Transaction* dentro da *Session*. O processo de manter as modificações realizadas aos objetos que estão na cache, durante o tempo de vida de uma *Transaction*, e de depois propagar as modificações para a base de dados, denomina-se de *automatic dirty checking*.

Pode se utilizar o método *Session.clear()* para se remover todos os objetos na cache, e desta maneira prevenir que as modificações guardadas sejam persistidas. No entanto, se só um objeto necessitar de ser removido da cache, pode-se utilizar o método *Session.evict(Object object)*. Não há remoção automática de elementos da cache, pois é esperado que as instâncias *Session* têm curta duração.

A figura 2.3 apresenta um diagrama ilustrativo, que representa a utilização normal da *first level cache*.



(a) Obtenção da referência para a entidade

(b) Obtenção dos dados da entidade, a partir da base de dados

Figura 2.3: Exemplo da utilização da *first level cache*.

Quando um objeto é carregado para a *Session*, em vez de se ir buscar todos os dados da entidade à base de dados, cria-se um objeto especial para criar um espaço reservado na *first level cache* que só contém apenas o identificador da entidade. Este objeto é chamado de *proxy*. Depois de estar em cache, a referência para esse *proxy* é devolvida para a aplicação. Quando a aplicação tenta aceder a qualquer campo da entidade sem ser o identificador, os campos vazios do *proxy* são preenchidos com a informação da entidade obtida da base de dados. O Hibernate utiliza este mecanismo de ir reservando espaço na cache, também denominado de *lazy fetching*, para minimizar os acessos à base de dados, e também se pode aplicar a coleções de objetos, onde vários objetos *proxy* podem ser guardados na *first level cache*, com uma só operação.

Como já foi referido, esta cache é gerida pela interface *PersistenceContext*. Na implementação desta interface que é utilizada pelo Hibernate, a classe *StatefulPersistenceContext*, a *first level cache* é implementada utilizando um *HashMap* para guardar os elementos,

utilizando o identificador como chave. Esta classe também guarda outros objetos para auxiliar as funcionalidades de *lazy fetching*, incluindo outros objetos *Map* que são manipulados pelos mesmos métodos que utilizam a cache.

2.2.5.2 *Query Cache*

A *query cache* é uma cache que guarda os resultados de consultas que vão sendo feitas ao longo da execução de uma aplicação que use Hibernate.

Esta cache é composta por duas regiões: uma que guarda os resultados das consultas que vão sendo colocadas na cache, e outra que guarda a data e a hora das últimas modificações nas tabelas que estão em cache.

Como já foi referido, muitas aplicações podem não beneficiar desta cache. Como tal, esta cache não está ativa por omissão, tendo que o programador especificar no ficheiro de configuração do Hibernate que a pretende utilizar, com a inclusão da propriedade `hibernate.cache.use_query_cache`.

A determinação das consultas cujos resultados vão sendo colocados na cache fica a cargo do programador, que deve utilizar o método *Query.setCacheable(true)* quando constrói a consulta com o Hibernate.

2.2.5.3 *Second-level Cache*

A *second-level cache* é uma cache partilhada por todas as instâncias *Session* ao longo de uma aplicação. Deste modo os objectos em cache podem ser reutilizados ao longo da execução das instâncias *Session* que estiverem abertas no momento. Esta cache não tem conhecimento de mudanças efetuadas por outras aplicações nas entidades na base de dados, portanto há que ter alguma precaução quando se utilizam bases de dados que sejam partilhadas por outras aplicações externas e adequar as políticas de invalidação dos dados em cache.

Para se utilizar esta cache há que colocar no ficheiro de configuração do Hibernate a informação de que implementação da cache se irá utilizar. Também há que designar algumas entidades a serem colocadas na cache, por por omissão o Hibernate não coloca todas as entidades na *second-level cache* [The13]. Isto pode ser feito através da utilização da anotação `@Cacheable` ou através da inclusão do elemento `<cache>` nos ficheiros XML de mapeamento das entidades.

Como a *second-level cache* é partilhada por todas as instâncias *Session* da aplicação, há que garantir que a implementação utilizada fornece garantias de consistência do estado das entidades em caso de ocorrerem modificações concorrentes. Cada implementação da *second-level cache* deve suportar várias estratégias de concorrência para os elementos em cache. O programador pode especificar a estratégia que pretende utilizar ao nível da aplicação e ao nível de cada entidade. As estratégias que podem ser escolhidas são:

- `read-only` – a cache permite apenas leituras das entidades em cache e não permite modificações.

- `read-write` – a cache permite alterações concorrentes nas entidades, assegurando com métodos de *locking* que as entidades da cache não são alteradas por duas *threads* ao mesmo tempo.

- `nonstrict-read-write` – a cache permite alterações concorrentes nos elementos mas não garante que há acesso exclusivo aos elementos. Aplica-se em casos onde há pouca probabilidade que duas *threads* acedam ao mesmo elemento.

- `transactional` – assegura que os elementos da cache se mantêm consistentes à medida que as transações que os modifiquem sejam *comitted* na base de dados.

Para se especificar a estratégia da cache ao nível de todas as entidades, tem que ser incluir no ficheiro de configuração do Hibernate, através da propriedade `hibernate.cache.default_cache_concurrency_strategy`.

Existem várias implementações disponíveis da *second-level cache*, havendo algumas que são distribuídas juntamente com o Hibernate, nomeadamente uma cache que apenas se chama Caching Region e é implementada pela equipa do Hibernate, e duas caches implementadas por entidades externas, a EhCache e a Infinispan.

A Caching Region trata-se apenas de uma estrutura que utiliza um objecto *ConcurrentHashMap* para ir guardando as instâncias dos objectos utilizados pelas várias sessões.

A EhCache tem origem num projeto *open-source* mantido pela empresa Terracotta e que se pode integrar no Hibernate como uma *second-level cache*. A Infinispan é uma cache *open-source* que é mantida pela JBoss, a mesma empresa que mantém o Hibernate. A EhCache é compatível com todas as estratégias de concorrência para a *second-level cache* suportadas pelo Hibernate, mas a Infinispan apenas suporta duas - `read-only` e `transaccional`.

Estas duas caches suportam topologias muito similares, que possibilitam a utilização de vários modos na sua arquitetura [Tea13b] [Tea13a]. Isto permite que estas caches possam utilizar em contextos de aplicações mais simples até aos contextos de aplicações complexas, que requerem alta escalabilidade e desempenho. Porém, na EhCache o acesso a alguns modos são pagos.

Numa visão comum entre as duas caches, as topologias disponíveis são:

- **Independente** – a cache localiza-se no nó da aplicação. Se a aplicação estiver a correr simultaneamente em vários outros nós, as caches não irão comunicar entre si e como tal só asseguram consistência fraca entre as mesmas entidades que estiverem em caches diferentes.

- **Distribuída** – cada nó da aplicação contem um nó da cache que apenas contém um subconjunto dos dados em cache, que foram recentemente utilizados por aquele nó. No caso do Infinispan, é efetuado um mapeamento das entidades nas várias caches através de um algoritmo de *hash* consistente. Isto faz com que se reutilize as entradas em cache e que se reduzam os acessos à base de dados.

- **Replicada** – as entidades que estão em cache estão localizadas em todos os nós da aplicação. Deste modo quando uma entrada é adicionada a uma cache todos os nós da aplicação também poderão aceder a essa entidade. A replicação dos novos dados em cache pode ser síncrona ou assíncrona, e no último caso a cache onde as alterações aconteceram bloqueia enquanto se propagam as alterações. O único modo de consistência disponível é consistência fraca.

Nas topologias distribuída e replicada, quando ocorrem modificações na cache há dois métodos para se propagarem essas alterações pelas outras caches associadas: ou por notificação de invalidação ou por cópia. No caso da notificação de invalidação, a *thread* que efetua uma modificação numa entrada em cache dissemina uma notificação a todas as outras caches associadas para que estas vão buscar o estado dessa entidade à base de dados, de modo a que fique em cache o estado mais recente dessa entidade. No caso da cópia, a *thread* que efetua a modificação em vez de enviar uma notificação envia antes a entidade que foi alterada para as outras caches.

2.3 *Benchmarking* de Bases de Dados Relacionais e Orientadas a Objectos

É bastante importante para este trabalho que se escolha uma bateria de testes adequada para realizar o estudo de performance sobre o Hibernate. Neste caso, o teste mais adequado seria ou um *benchmark* especificamente desenhado para representar os *workloads* possíveis das aplicações multi-camada mais comuns, ou então um *benchmark* específico para ferramentas ORM. Como ainda há pouco trabalho publicado sobre *benchmarks* nestes âmbitos, a solução mais aproximada é adaptar um *benchmark* já existente, que esteja focado para ambientes orientados a objetos.

Devido à existência de múltiplos *benchmarks* para ferramentas orientadas a objetos, é importante analisar as características de cada um de modo a se poder escolher o que melhor refletir a carga de trabalho que é comum a uma ferramenta ORM. Esta secção pretende apresentar alguns *benchmarks* que foram selecionados e avaliar se os testes que neles são efetuados correspondem às operações comuns de uma aplicação baseada em ORM.

2.3.1 *Benchmark* OO7

O *benchmark* OO7 foi originalmente desenhado para avaliar sistemas de gestão de bases de dados orientadas a objetos [Car+94], e foi criado para incluir mais funcionalidades que os *benchmarks* que surgiram anteriormente, como a OO1 e a Hypermodel. Existem várias implementações deste *benchmark*, e apesar do seu foco estar nas bases de dados orientadas a objetos, surgiram outros trabalhos que aplicam a OO7 para testar sistemas de ORM [Zyl10].

Este *benchmark* pretende simular uma biblioteca de desenho de componentes eletrónicos, utilizando uma hierarquia de objetos que é representada na figura 2.4.

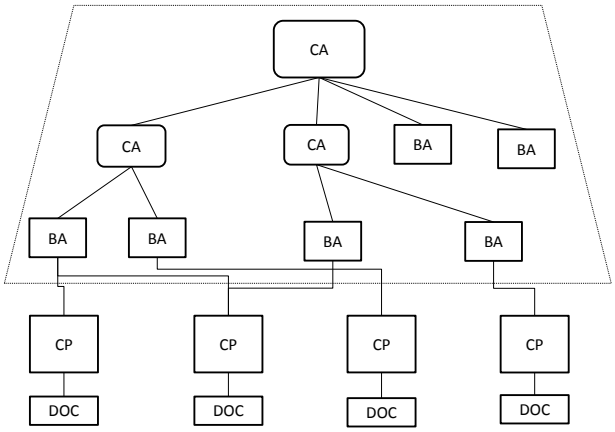


Figura 2.4: Diagrama representativo das relações entre os objetos no *benchmark* OO7.

Como o diagrama mostra, um *Model* (representado pelo trapézio) é composto por objetos *ComplexAssembly* ("CA"). Um objeto *ComplexAssembly* pode ser composto por outros objetos *ComplexAssembly* ou por objectos *Basic Assembly* ("BA"). Um objeto *Basic Assembly* é composto por muitos objetos *CompositePart* ("CP") e cada um destes objetos tem associado um objeto *Document* ("DOC").

Todas as relações entre os objetos podem ser de um-para-muitos, muito-para-muitos ou mesmo um-para-um, e estas relações são representadas como coleções em cada objeto. A maioria das relações entre os objetos são bidirecionais.

O trabalho exercido pelo *benchmark* é baseado em fazer operações de consulta, inserção e remoção de objetos e percursos pela hierarquia dos objetos, fazendo consultas e modificações dos dados de cada objeto [Zyl10].

O número de relações e o número de conexões entre os objetos pode ser parametrizável. Também é possível especificar o número de objetos criados durante a execução.

Todas as operações no *benchmark* são realizadas em transações e cada operação é repetida de acordo com um parâmetro específico. Cada repetição é executada dentro de uma nova transação [Zyl+09].

A carga de trabalho deste *benchmark* não representa uma carga de trabalho das aplicações multi-camada atuais, mas no entanto ainda pode ser utilizada para inferir o comportamento do sistema quando a base de dados é alvo de um enorme número de consultas.

2.3.2 *Benchmark* TPC-W

O *benchmark* TPC-W pertence ao conjunto de *benchmarks* do Transaction Processing Performance Council (TPC), e o seu objetivo é representar a carga de trabalho de uma aplicação de uma loja online, que tenha a camada de persistência numa base de dados relacional.

Este *benchmark* implementa uma loja online de livros (como a Amazon.com), que simula as operações mais comuns de uma aplicação deste tipo: consulta de informação sobre livros, compras de livros, processamento de pagamentos, entre outras. Estas operações são executadas ao longo de sessões, abertas por *browsers* simulados, como se fosse numa ambiente real. As operações numa sessão são executadas concorrentemente com operações noutras sessões, e por isso o *benchmark* também pode ser utilizado para testar a consistência dos dados [Men02].

Apesar de nem todos os padrões de carga de trabalho serem representados neste *benchmark* (por exemplo, um site de leilões pode utilizar com mais frequência um subconjunto de dados), esta *benchmark* ainda é das mais adequadas a testar as arquiteturas de aplicações multi-camada, pois a carga de trabalho é similar à carga de trabalho esperada para este tipo de aplicações, e não se limita só a percorrer os dados ou a fazer consultas aos dados gerados pela *benchmark*. A coisa importante de medir em aplicações com uma camada de dados em base de dados é se a carga de trabalho de certas operações prejudica a performance, de modo a se tentar perceber como se pode melhorar a arquitetura da aplicação para suportar essa carga de trabalho sem serem levantados problemas. Para além disso, as operações desta *benchmark* são muito parecidas com as operações efetuadas por utilizadores reais, e também é importante que as simulações efetuadas se pareçam o máximo possível com o contexto real onde a arquitetura será inserida. Esta *benchmark* encontra-se descrita com mais detalhe na secção 3.1 do capítulo 3.

2.3.3 *Benchmark* OCB

O objetivo deste *benchmark* é, à semelhança do *benchmark* OO7 abordado anteriormente, a de avaliar o desempenho de bases de dados orientadas a objetos e foi criado originalmente para avaliar o desempenho dos algoritmos de *clustering* dos sistemas de gestão de bases de dados orientadas a objetos [DS02].

Uma característica importante deste *benchmark* e que o diferencia em relação aos seus predecessores (OO1 e HyperModel) é a de assentar numa base de objetos rica e com bastantes referências, o que permite que possam ser criadas inúmeras hierarquias entre os objetos do teste, o que faz com que possam ser utilizados diferentes tipos de dados, consoante os vários tipos de aplicação em que se utiliza a base de dados a testar. Esta generalidade é obtida também através de um conjunto extensivo de parâmetros, que permite inclusive, que sejam utilizados vários tipos de *workload* no testes.

Apesar destas características, este *benchmark* ainda não oferece mecanismos para avaliar o controlo de concorrência dos vários sistemas de bases de dados orientadas a objetos,

o que é largamente utilizado nas aplicações mais recentes. Isto faz com que este *benchmark* não possa representar o *workload* de uma aplicação multi camada *three tier*.

2.4 *Benchmarking* em Ferramentas ORM

Apesar de existir ainda muito pouco trabalho publicado acerca do desenvolvimento e aplicação de *benchmarks* específicas para ferramentas ORM, existem alguns trabalhos que fazem estudos de performance deste tipo de ferramentas, através da utilização de versões adaptadas de *benchmarks* já existentes ou através da construção de testes específicos.

No trabalho [Zyl10] é feita uma adaptação do *benchmark* OO7 para ferramentas ORM. O trabalho [Zyl10] tem o objetivo comparar o desempenho de ferramentas ORM com bases de dados orientadas a objetos, através de um estudo de performance. Também se pretende quantificar o impacto na performance de uma aplicação imposto pela utilização de cada um destes mecanismos.

Os testes de performance consistiram na comparação dos resultados do *benchmark* OO7 utilizando dois sistemas de gestão de bases de dados orientadas a objetos, um *open-source* (db4o) e o outro proprietário (Versant) e o Hibernate, a ferramenta ORM que também é abordada neste trabalho. Na primeira fase do trabalho foram efetuados testes com as versões *out of the box* das ferramentas e na segunda fase foram feitas algumas otimizações à adaptação do *benchmark*, de acordo com recomendações das equipas que desenvolvem as várias ferramentas que foram analisadas.

No início, pressupôs-se que a utilização de ferramentas de ORM inseriam mais impacto na performance que a utilização de bases de dados orientadas a objetos, devido à inserção de uma camada adicional entre a aplicação e a base de dados. Observou-se então que com a utilização das otimizações da segunda fase do trabalho (uso de caches e de configurações especiais), que foram obtidos resultados similares com a utilização de bases de dados orientadas a objetos.

Existem outros trabalhos [CJ10], onde são efetuadas comparações entre várias ferramentas ORM, mas foram apenas estudadas ferramentas desenhadas para serem utilizadas na plataforma .NET – o Microsoft Entity Framework e o NHibernate, que é a versão .NET da ferramenta Hibernate.

Para se efetuar a comparação entre as duas ferramentas ORM abordadas em [CJ10] foram efectuados testes de performance, à semelhança dos outros trabalhos abordados anteriormente. Neste caso não foi utilizada uma adaptação de um *benchmark* já existente, mas os testes efectuados consistiram na execução de várias consultas a um conjunto de dados, utilizando as linguagens específicas de cada ferramenta – HQL para o NHibernate e LINQ e EntitySQL para o Microsoft Entity Framework – e utilizando diretamente a base de dados, através de SQL. O conjunto de consultas criado abrange operações comuns de aplicações deste tipo: obtenção, remoção e inserção de dados.

Depois, compararam-se os resultados obtidos com a execução das consultas utilizando cada uma das ferramentas e utilizando também a base de dados. A partir dos resultados obtidos observou-se que a diferença de performance entre as ferramentas ORM e a base de dados era muito pequena e como tal não se justificaria a fraca popularidade das ferramentas ORM. No entanto, este teste não foi exaustivo o suficiente para representar a carga de trabalho real de uma aplicação que utilize ferramentas ORM pois as operações utilizadas não refletem o mesmo tipo de complexidade das operações comuns destas aplicações.

3

Benchmark TPC-W

Observando os *benchmarks* abordados no capítulo 2, verificou-se que o que preenchia melhor os requisitos dos testes para este trabalho seria o *benchmark* TPC-W, pois apesar do TPC ter declarado que o *benchmark* estaria obsoleto, verifica-se que ainda é um bom exemplo para o comportamento real de uma aplicação *web* multi-camada. No entanto, foi necessário efetuar algumas modificações a uma implementação deste *benchmark* para que esta possa utilizar o Hibernate como camada de persistência.

Este capítulo pretende apresentar o TPC-W com mais detalhe, e também apresentar o processo de adaptação da versão que utiliza a ferramenta Hibernate para a persistência dos dados.

3.1 *Benchmark* TPC-W

O *benchmark* TPC-W pertence ao conjunto de *benchmarks* do TPC e o seu objetivo é representar a carga de trabalho de uma aplicação para comércio *online* cuja camada de armazenamento seja uma base de dados [Cou01].

Este *benchmark* simula uma loja online para venda de livros (do mesmo tipo que a Amazon.com), com suporte para operações de consulta de informação sobre livros, vendas de livros, processamento de pagamentos, entre outras funcionalidades. Estas operações são executadas em várias sessões, abertas por *browsers* emulados, para se simular um ambiente real. As operações de uma sessão são concorrentemente executadas com as operações das outras sessões ativas, para que se possa testar a consistência de dados [Men02].

As relações entre as tabelas do *benchmark* estão representadas na figura 3.1. Como

se pode observar temos informação sobre os livros e as encomendas dos livros (nas tabelas ITEM, AUTHOR, ORDER e ORDER_LINE), informação sobre os clientes (nas tabelas COSTUMER, ADDRESS e COUNTRY), e informação sobre os pagamentos das encomendas (tabela CC_XACTS).

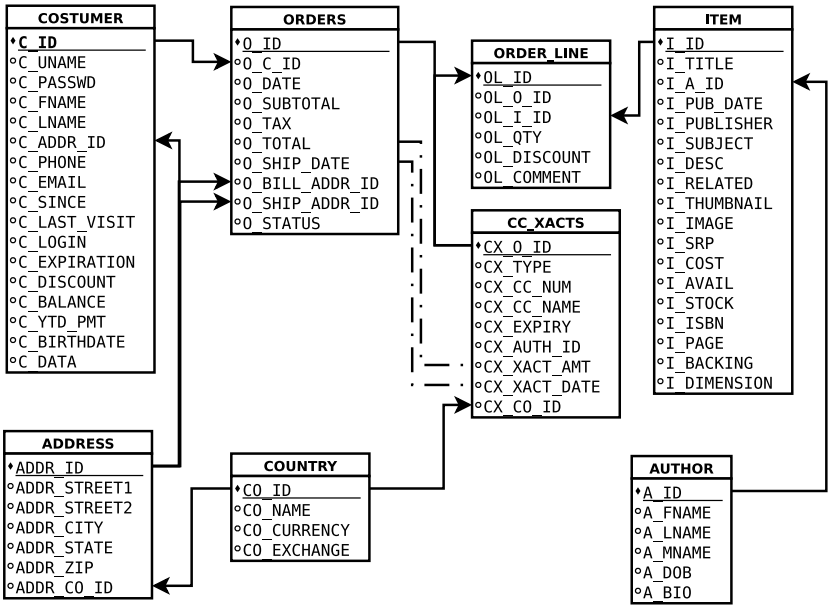


Figura 3.1: Diagrama das entidades da base de dados do *benchmark* TPC-W (retirado de [Cou01]). Atributos a sublinhado indicam as chaves primárias, as setas ligando os atributos indicam chaves externas e as linhas a tracejado indicam relações um-para-um entre atributos que não são chave.

Apesar de este *benchmark* não modelar todos os padrões de utilização de aplicações web, continua a encontrar-se entre os *benchmarks* mais representativas do *workload* de uma aplicação *web* e, por isso, continua a ser uma boa escolha para testar as arquiteturas que suportam as aplicações multi-camada.

3.1.1 Operações

Numa primeira fase do *benchmark*, são inseridas várias entradas na base de dados, para que depois possam ser utilizadas pelas operações do *benchmark*. A criação de novas entradas é apenas restrita às tabelas COUNTRY, ADDRESS, CUSTOMER, AUTHOR, ITEM, ORDER e CCX_ACTS. À excepção dos dados inseridos na tabela COUNTRY, todos são gerados aleatoriamente.

As operações são executadas no contexto de sessões que vão sendo criadas ao longo da execução. A sequência das operações de cada sessão é apresentada na figura 3.2.

A direção das setas entre os vários nós representam a sequência da execução das operações. Quando existe mais que uma seta a sair de um nó, ou seja, várias operações que podem ser executadas, é escolhida com recurso a um método probabilístico que se baseia na atribuição de um número entre 1 e 9999 a cada transição de operações e depois

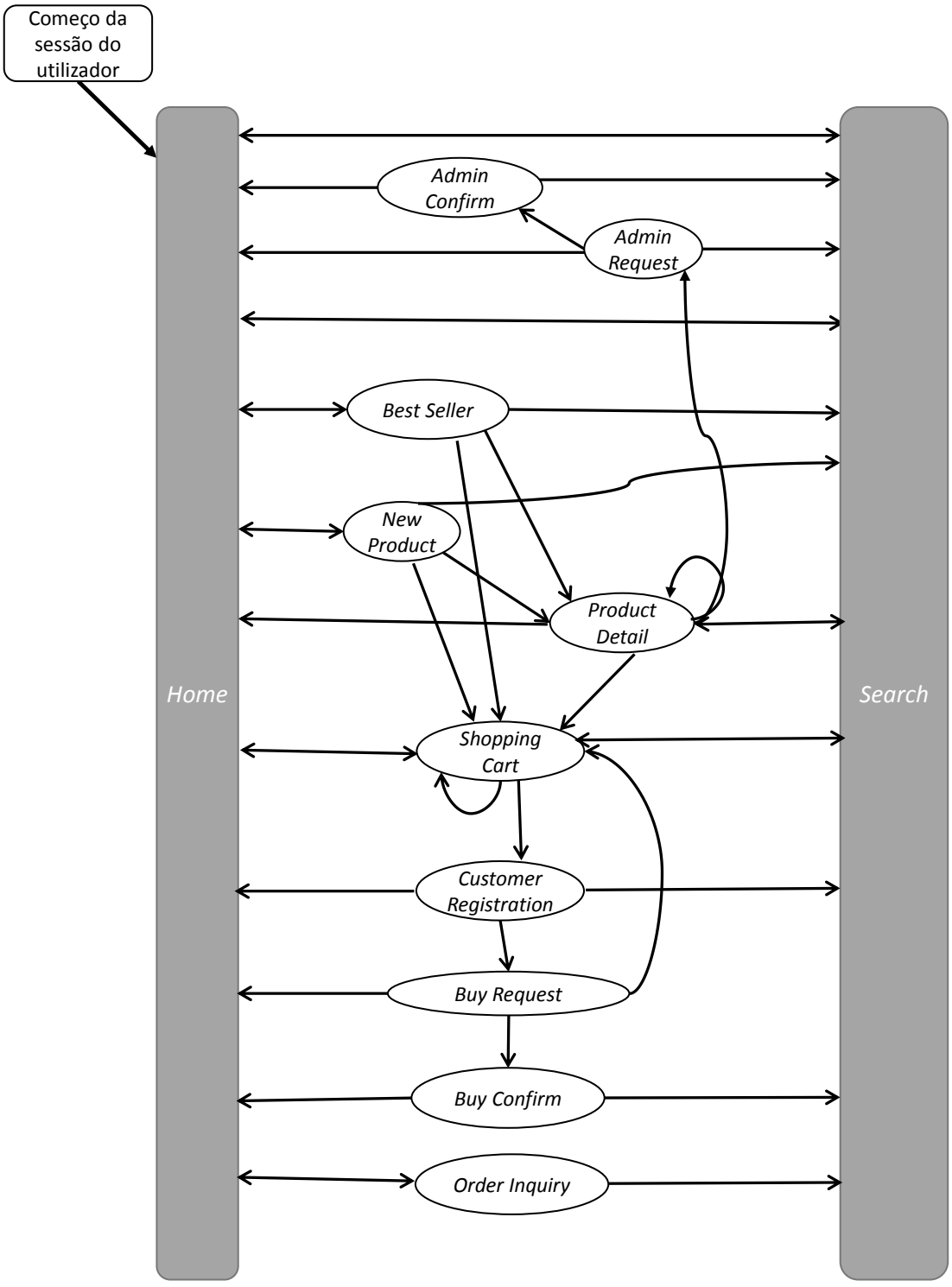


Figura 3.2: Diagrama representativo das transições entre as operações do *benchmark* TPC-W (adaptado de [Cou01]). Cada nó representa uma operação e cada arco entre dois nós representa uma transição de operações.

escolhe-se a operação que for imediatamente igual ou maior a um número gerado aleatoriamente no intervalo de 1 a 9999 [Cou01].

Cada operação pode compreender uma ou mais transações, de acordo com a operação em questão, sendo que cada operação do *benchmark* é descrita abaixo.

Home A operação *Home* inicia sempre a sessão de um utilizador virtual no *benchmark*. Compreende a obtenção do primeiro e último nome do cliente, e a apresentação de informação sobre um item, cujo identificador é escolhido através de uma função pseudo-aleatória.

Shopping Cart A operação *Shopping Cart*, realiza a atualização da lista de itens presentes no carrinho de compras. Esta operação recebe uma lista de pares de itens e a quantidade correspondente, uma *flag* e um identificador do carrinho ao qual os itens devem ser adicionados. A adição/remoção de todos os itens da lista deve ser efetuada de forma atómica. Se o identificador do carrinho de compras passado como argumento não existir, é criado um novo carrinho, que é associado ao utilizador corrente, indo-se obter a informação necessária sobre o utilizador. Depois é feita a atualização das quantidades de itens que existem no carrinho, caso o item já exista, ou a adição de novos itens ao carrinho, caso estes ainda não estejam presentes. Se a *flag* passada como argumento for igual a 'N', a ação é a mesma, mas todos os itens que tenham a quantidade igual a 0 serão removidos do carrinho. Caso a lista itens esteja vazia e o carrinho estiver vazio, escolhe-se um item de forma aleatória e adiciona-se ao carrinho o seu primeiro item relacionado.

Customer Registration A operação de *Customer Registration* representa a operação de criar um utilizador, caso o identificador passado como argumento não exista. Os dados do novo utilizador são gerados aleatoriamente.

Buy Request A operação *Buy Request* faz o registo ou a identificação de um novo utilizador e obtém as informações do carrinho de compras associado ao utilizador. Caso o utilizador já exista, é atualizado o valor do *timestamp* de *login* e o tempo de expiração da sessão do utilizador. Também se efetua a verificação da password do utilizador. Quando se faz o registo de um novo cliente, verifica-se se o identificador do endereço é existente, e caso não seja, cria-se um novo endereço. No final desta operação são atualizados e mostrados os dados do carrinho de compras e são também gerados dados de cartão de crédito para serem associados ao utilizador.

Buy Confirm A operação *Buy Confirm* cria uma nova encomenda a partir da informação que está no carrinho de compras. Também é verificada a informação de pagamento que foi gerada na operação *Buy Request*. Se for passado um endereço de entrega da encomenda, é verificado se o identificador desse endereço existe. Se não existir, é criado um

novo endereço a partir dos dados fornecidos. Para se ser criada uma nova encomenda é necessário criar uma nova entrada na tabela `ORDER` e para cada item no carrinho de compras adicionar uma entrada na tabela `ORDERLINE`. No final desta operação são devolvidos os dados da nova encomenda. Também é criada uma nova entrada na tabela `CC_XACTS` que contém os dados do cartão de crédito que está associado à encomenda.

Order Inquiry A operação *Order Inquiry* verifica os dados de um utilizador já existente e devolve a informação da última encomenda que este efetuou. Esta informação consiste não só na informação da entrada correspondente à encomenda na tabela `ORDER`, mas também na informação do endereço de pagamento e do endereço de entrega da encomenda, na informação de cada item da encomenda – nome, identificador, custo por unidade e quantidade – e na informação do cartão de crédito associado à encomenda – número e tipo de transação.

Search Esta operação representa a operação de pesquisa nos livros, por título, por assunto ou por autor. Realiza uma comparação do termo de pesquisa passado à operação com o atributo do item correspondente. No caso da pesquisa por nome do autor, compara-se com o atributo correspondente ao último nome do autor. Os primeiros 50 resultados obtidos, são devolvidos ao utilizador e adicionalmente são mostrados os 5 itens relacionados de um item escolhido aleatoriamente, da mesma maneira que é executado na operação *Home*.

New Products Nesta operação são obtidos os 50 itens que foram publicados mais recentemente. Depois de ser devolvida a lista dos itens são também obtidos os 5 itens relacionados com um item cujo identificador foi obtido aleatoriamente, da mesma maneira que foi feito na operação *Home*.

Best Sellers Nesta operação são obtidos os 50 itens mais vendidos, ordenados pela soma das suas quantidades vendidas em cada encomenda, por ordem decrescente. Também são obtidos os 5 itens relacionados com um item escolhido aleatoriamente, tal como foi feito na operação *Home*.

Product Detail Esta operação reporta toda a informação acerca de um item, cujo identificador é fornecido à operação. Esta informação é a que se encontra na entrada da tabela `ITEM` com o identificador igual ao que é passado como argumento.

Admin Change Esta operação faz uma alteração a dados de um item. Também é feito o processamento para obter 5 itens que depois vão ser relacionados com o item que está a ser atualizado. Este processamento consiste na ordenação de todas as encomendas por ordem descendente de data e selecionam-se as primeiras 10000. Depois, obtêm-se a lista de todos os clientes, sem repetições, que encomendaram aquele item e cuja encomenda

esteja no grupo de encomendas obtido no passo anterior. Para cada cliente nestas condições é obtida a lista de itens que foram encomendados nas encomendas do grupo inicial, sem repetições, e estes itens são ordenados pela quantidade agregada, isto é, a soma de quantidades das encomendas onde estão presentes. Os primeiros cinco elementos desta lista serão os itens relacionados. Se a lista tiver menos de cinco elementos, é incrementado o identificador do último elemento até se encontrar um identificador válido de um item. Se não for obtido nenhum item, incrementa-se de 7 em 7 unidades o identificador do item que está a ser alterado, de modo a serem encontrados cinco identificadores nestas condições. Se um destes valores ultrapassar o número de itens presentes no sistema, utiliza-se o valor do identificador do primeiro item da tabela `ITEM`.

3.1.2 *Workloads*

As operações deste *benchmark* que são utilizadas neste trabalho estão organizadas em dois *workloads*, denominados de *browsing* e *ordering*. Cada *workload* caracteriza-se pela percentagem total de operações de consulta ou leitura de dados (*Home*, *New Products*, *Best Sellers*, *Product Detail* e *Search*) e de escrita ou modificação de dados (*Shopping Cart*, *Registration*, *Buy Request*, *Buy Confirm*, *Order Inquiry* e *Admin Change*) [Men02].

No *workload browsing* a percentagem de operações de leitura é de 95% e de escrita 5%, sendo este o *workload* utilizado para avaliar o comportamento dos sistemas quando são alvo de um grande volume de consultas e poucas escritas [Men02].

No caso do *workload ordering*, a percentagem de operações de escrita e de leitura são ambas de 50%, pelo que este *workload* pode ser utilizado para avaliar como o sistema em teste reage a uma carga de operações de escrita mais alta que no *workload browsing* [Men02].

3.2 Adaptação do TPC-W para utilizar Hibernate

Para que se pudesse criar uma versão do TPC-W que utilize o Hibernate como camada de persistência existe a necessidade de se procederem a algumas modificações, quer na definição de meta-dados para que o Hibernate possa fazer corretamente o mapeamento entre os objetos e as entradas na base de dados, quer na re-implementação das operações do *benchmark*.

A implementação do TPC-W que se escolheu como ponto de partida para a nossa versão do TPC-W com Hibernate utiliza MySQL [1]. Esta escolha deveu-se ao facto de como o MySQL que também foi a nossa escolha para o sistema de gestão da base de dados para ser utilizado conjuntamente com o Hibernate, o processo de adaptação para uma versão com Hibernate seria mais fácil. Esta secção descreve todas as alterações que foram efetuadas para adaptar esta implementação com MySQL para que esta passe a utilizar o Hibernate como camada de persistência.

[1] `https://github.com/PedroGomes/TPCw-benchmark`

3.2.1 Estrutura da Base de Dados e Configuração

Como esquema para a base de dados que a nossa versão do TPC-W utiliza, escolheu-se utilizar o mesmo esquema utilizado pela nossa implementação do TPC-W que temos como referência. A única alteração que foi necessária efetuar no esquema para a nossa implementação foi a de se introduzir a definição das chaves primárias e das chaves estrangeiras nas tabelas, para que depois se pudessem definir as relações necessárias para que o Hibernate possa efetuar consultas HQL sobre as entidades. Quanto às chaves primárias das tabelas, também foi adicionado que estas seriam geradas automaticamente pela base de dados, retirando assim essa responsabilidade ao Hibernate.

Conforme foi referido na secção 2.2.1 do capítulo 2 que o Hibernate necessita também de saber a informação necessária sobre a base de dados e sobre todas as entidades que têm que ser persistidas utilizando o Hibernate. Como tal, criou-se um ficheiro *hibernate.cfg.xml* contendo a informação referente à implementação do TPC-W: localização da base de dados, *username* e *password*, e a lista das classes que representam todas as entidades.

3.2.2 Modelo de Dados

Para que o Hibernate possa persistir os dados da aplicação na base de dados há a necessidade de se implementarem classes Java para representar as entidades utilizadas pelo *benchmark* TPC-W.

Como a implementação do *benchmark* de referência já utilizava algumas classes que eram análogas às entidades representadas na base de dados, utilizou-se a implementação dessas classes como base para as que a nossa versão do TPC-W utiliza, sendo depois feitas algumas alterações necessárias para que o Hibernate consiga efetuar operações sobre as entidades representadas.

Para que o Hibernate possa perceber as relações que existem entre as entidades e para que se possam efetuar consultas HQL sobre estas entidades, as relações presentes no esquema de base de dados têm que ser refletidas, de alguma forma, sobre o esquema de objetos. Como tal, na nossa adaptação adicionaram-se a cada classe Java os elementos das outras classes necessários para que seja possível representar as relações entre os vários objetos, de acordo com as relações do esquema da base de dados. Como se verá mais à frente neste documento, esses novos elementos irão ser utilizados nas consultas HQL da implementação das operações do *benchmark*.

No esquema da base de dados verificou-se que em algumas tabelas as chaves primárias eram compostas por vários atributos. Para que o Hibernate possa reconhecer esse tipo de chave têm que se criar classes especiais que contenham os atributos que compõem a chave. Depois, na classe que representa a entidade adiciona-se o elemento da classe que representa a chave. Na nossa adaptação foi necessário fazer isto com três entidades: na classe *Results* em que a chave é representada pela classe *ResultsPk*, na classe *ShoppingCarLine* em que a chave é representada pela classe *ShoppingCarLinePk* e na classe

OrderLine em que a chave é representada pela classe *OrderLinePk*.

A figura 3.3 representa o diagrama de classes com as entidades do *benchmark* depois das modificações efetuadas no modelo de dados.

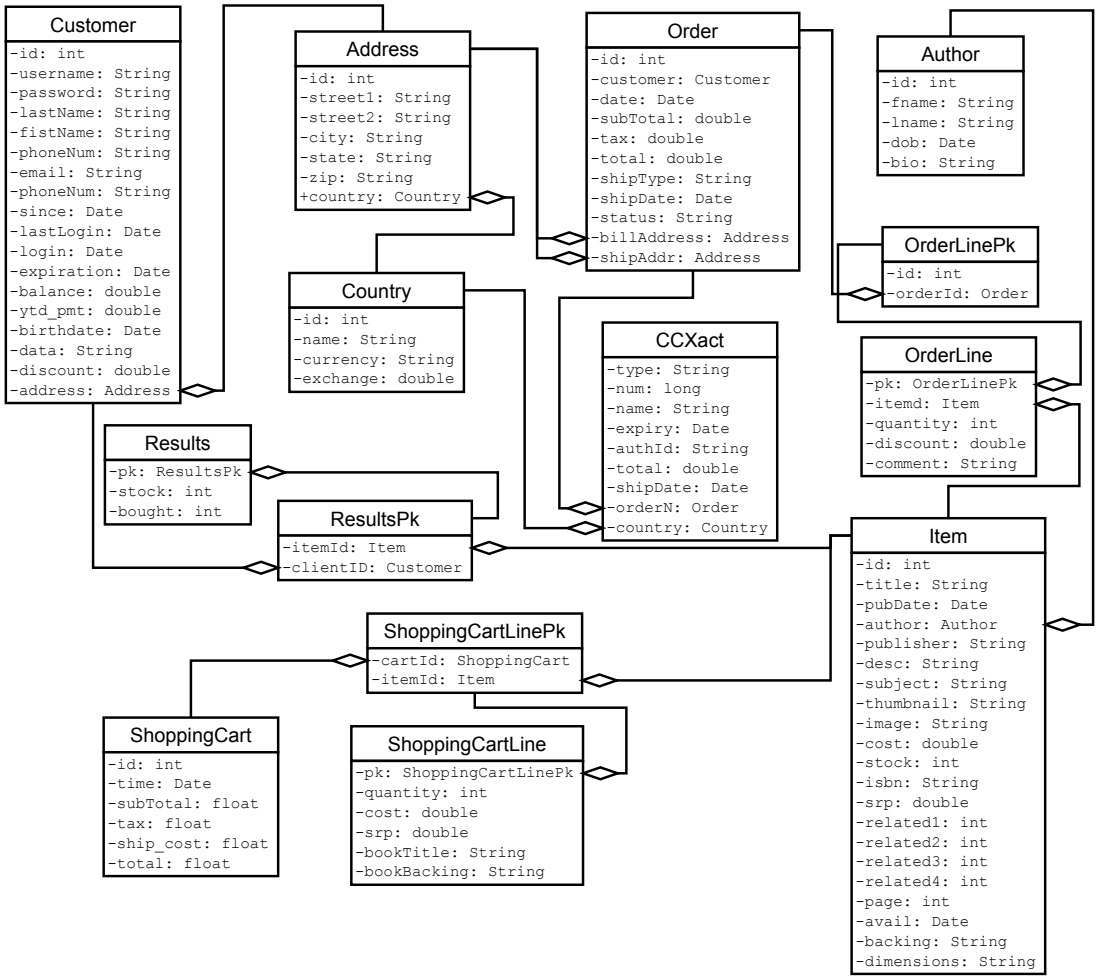


Figura 3.3: Diagrama que representa o esquema de classes da versão do TPC-W que utiliza Hibernate.

Como se pode observar, estão presentes as classes que representam as entidades da base de dados (*Customer*, *Address*, *Country*, *Order*, *CCXact*, *Results*, *ShoppingCart*, *ShoppingCartLine*, *Author*, *OrderLine* e *Item*) e as novas classes que representam as chaves compostas das tabelas (*ResultsPk*, *ShoppingCarLinePk* e *OrderLinePk*).

Também é necessário que o Hibernate tenha alguma informação sobre como corresponder os elementos de cada classe aos atributos de cada tabela, bem como da informação sobre como as relações entre as tabelas são refletidas nas classes Java, isto é, que elementos da classe são chaves primárias e quais são chaves estrangeiras. Normalmente esta informação é dada sob a forma de anotações nos elementos das classes Java ou pela definição de ficheiros XML. Na adaptação apresentada neste trabalho essa informação é fornecida através da introdução de anotações no código das classes que representam as

entidades da base de dados. Adicionaram-se anotações em cada classe, indicando qual a tabela da base de dados que é representada pela entidade, e anotando também cada elemento da classe com a indicação do atributo da tabela que lhe corresponde. A listagem 3.1 apresenta um excerto da classe *Country*, que exemplifica as anotações que foram adicionadas ao código.

Listagem 3.1: Excerto da classe *Country* com as anotações de metadados do Hibernate.

```
1  import org.uminho.gsd.benchmarks.interfaces.EntityInt;
2  (...)
3  import javax.persistence.*;
4
5  @Entity
6  @Table(name="COUNTRY")
7  public class Country implements EntityInt, Serializable {
8
9    private static final long serialVersionUID = -8918452772954105502L;
10
11   @Id
12   @Column(name="co_id")
13   @GeneratedValue
14   private int id;
15
16   @Column(name="co_name",nullable=false,length=50)
17   private String name;
18
19   @Column(name="co_currency",nullable=false,length=18)
20   private String currency;
21
22   @Column(name="co_exchange",nullable=false,
23         columnDefinition = "double precision default 0")
24   private double exchange;
25
26 (...)
27 }
```

No caso das classes que representam as chaves compostas das tabelas também há a necessidade de se introduzir anotações para que o Hibernate consiga processar cada elemento dessa classe como chave da tabela. Na classe que representa a tabela é necessário indicar a relação entre os atributos da chave primária com os atributos da tabela, e na classe que representa a chave apenas é necessário adicionar uma anotação a indicar que representa uma chave composta (*Embeddable*). A listagem 3.2 apresenta um excerto da classe *OrderLinePk* que representa as anotações adicionadas a esta classe, e a listagem 3.3 apresenta um excerto da classe *OrderLine*, onde a classe *OrderLinePk* é utilizada como chave composta.

Listagem 3.2: Excerto da classe *OrderLinePk* com as anotações de metadados do Hibernate.

```
1  import java.io.Serializable;
2  import javax.persistence.*;
3
4  @Embeddable
5  public class OrderLinePk implements Serializable{
6
7    private static final long serialVersionUID = 4993975757068484796L;
8
9    @Column(name="ol_id", nullable=false, updatable=false)
10   private int id;
11
12   @ManyToOne
13   @JoinColumn(name="ol_o_id")
14   private Order orderId;
15
16 (...)
17 }
```

Listagem 3.3: Excerto da classe *OrderLine* com as anotações de metadados do Hibernate.

```
1  import org.uminho.gsd.benchmarks.interfaces.EntityInt;
2  (...)
3  import javax.persistence.*;
4
5  @Entity
6  @Table(name="ORDER_LINE")
7  @AssociationOverrides({
8    @AssociationOverride(name = "pk.id",
9        joinColumns = @JoinColumn(name = "ol_id")),
10   @AssociationOverride(name = "pk.orderId",
11       joinColumns = @JoinColumn(name = "ol_o_id")) })
12 public class OrderLine implements EntityInt , Serializable{
13
14   private static final long serialVersionUID = 2310174331322953698L;
15
16   @EmbeddedId
17   private OrderLinePk pk;
18
19   @ManyToOne
20   @JoinColumn(name="ol_i_id")
21   private Item itemId;
22
23   @Column(name="ol_qty",nullable=false)
24   private int quantity;
25
26   @Column(name="ol_discount",nullable=false,
27         columnDefinition = "double precision default 0")
28   private double discount;
29
30   @Column(name="ol_comments",nullable=false,length=110)
31   private String comment;
```

```
32
33 (...)
34 }
```

3.2.3 Operações

Todas as operações do *benchmark* referidas na secção 3.1.1 deste capítulo necessitaram de ser implementadas para que pudessem utilizar a interface do Hibernate.

O processo de implementação das operações dividiu-se em duas fases: uma fase em que foram adaptados os métodos utilizados no processo de população da base de dados no TPC-W, e outra fase em que foram adaptados os métodos das operações do *benchmark*.

De uma maneira geral, a implementação dos métodos da fase de população do TPC-W consistiram na substituição da utilização das consultas SQL na implementação MySQL que escolhemos como base para a nossa versão, para se utilizar o processo que é recomendado pelo Hibernate. Este processo consiste na criação de um objeto da entidade a guardar, e depois na utilização do método `Session.save()` para o Hibernate guardar o objeto na base de dados. Os construtores das classes das entidades foram então alterados para inicializar todos os campos do objeto, de modo a facilitar este procedimento. A listagem 3.4 apresenta a implementação da operação de inserção de um endereço, utilizando os métodos do Hibernate.

Listagem 3.4: A sequência de instruções utilizadas na operação de adicionar um endereço no *benchmark* TPC-W, na versão que utiliza o Hibernate.

```
1  public static void saveAddress(Address address){
2    Session session =
3      SessionFactoryUtil.getSessionFactory().openSession();
4    Transaction tx = session.beginTransaction();
5
6    session.save(address);
7
8    tx.commit();
9    session.close();
10 }
```

Listagem 3.5: Um exemplo que representa a consulta SQL utilizada na operação de adicionar um endereço no *benchmark* TPC-W, na versão que utiliza bases de dados.

```
1 INSERT into ADDRESS
2 (addr_id, addr_street1, addr_street2, addr_city, addr_state, addr_zip,
3 addr_co_id)
4  VALUES (1, 'Rua dos Cravos', 'n 37 esq', 'Almada', 'Setubal', 9999, 75)
```

Como se pode observar na listagem 3.4, o método da versão Hibernate recebe como argumento o objeto `address` que foi previamente inicializado noutro método, e utiliza-se a instrução `session.save(address);` para persistir o objeto na base de dados.

Foi também criada uma classe auxiliar que contém alguns métodos auxiliares e que são partilhados pelos métodos de população e pelas operações do *benchmark*. Esta classe chama-se `HibernateUtils` e para além de ter métodos para guardar e obter objetos, também contém um método que utiliza a linguagem HQL para obter a *timestamp* atual da base de dados, pois alguns campos de algumas entidades (`Order`, `Customer`, `Author`, `CCXact`, `Item` e `ShoppingCart`) dependem desse valor.

Na segunda fase do processo de implementação, a implementação das operações do TPC-W consiste de forma geral na substituição dos métodos do JDBC, que são utilizados na implementação que serve de referência, para se passar a utilizar os métodos específicos do Hibernate, da classe *Session*, em conjunto com consultas escritas na linguagem de *query* do Hibernate, a HQL. A principal diferença é que a linguagem HQL está desenhada de uma forma orientada a objetos, e como tal, há que utilizar os objetos que representam as entidades em vez de se utilizar as tabelas da base de dados.

Nas listagens 3.6 e 3.7 estão dois exemplos da transformação de uma operação do *benchmark* da versão para bases de dados para a versão que utiliza o Hibernate.

Listagem 3.6: A implementação da operação *Home* do *benchmark* TPC-W, na versão para bases de dados.

```
1  public void HomeOperation(int costumer, int item) {
2     Connection con = getReadConnection();
3     try {
4       con.setAutoCommit(false);
5       PreparedStatement name = con.prepareStatement("
6         SELECT c_fname, c_lname
7         FROM CUSTOMER
8         WHERE c_id = ?");
9
10      name.setInt(1, costumer);
11      ResultSet rs = name.executeQuery();
12      con.commit();
13
14      (...)
15
16      rs.close();
17      name.close();
18    } catch (SQLException e) {
19      e.printStackTrace();
20    }
21
22    try {
23      PreparedStatement related = con.prepareStatement("
24        SELECT J.i_id, J.i_thumbnail
25        FROM ITEM I, ITEM J
26        WHERE (I.i_related1 = J.i_id
27          OR I.i_related2 = J.i_id
28          OR I.i_related3 = J.i_id
29          OR I.i_related4 = J.i_id
```

```
30          OR I.i_related5 = J.i_id)
31          AND I.i_id = ?");
32
33      related.setInt(1, item);
34      ResultSet rs = related.executeQuery();
35      con.commit();
36      (...)
37      rs.close();
38      related.close();
39    } catch (SQLException e) {
40      e.printStackTrace();
41    }
42  }
```

Listagem 3.7: A mesma operação que na listagem 3.6 mas na versão Hibernate do TPC-W.

```
1   public void HomeOperation(int costumer, int item) {
2     Session session = SessionFactoryUtil.getSessionFactory().openSession();
3     Transaction tx = session.beginTransaction();
4
5     @SuppressWarnings("unchecked")
6     List<Object> listNames = session.createQuery(
7         "select new list(c.lastName, c.firstName) from Customer as c")
8         .list();
9     (...)
10    @SuppressWarnings("unchecked")
11    List<Object> listItemsThumb = session.createQuery(
12        "select new list(j.id, j.thumbnail) "
13            + "from Item as i, Item as j "
14            + "where i.related1 = j.id or i.related2 = j.id or "
15            + "i.related3 = j.id or i.related4 = j.id or "
16            + "i.related5 = j.id and i.id = " + item).list();
17    tx.commit();
18    session.close();
19    (...)
20  }
```

Como se pode observar a versão da implementação desta operação que utiliza o Hibernate apresenta algumas diferenças da implementação que utiliza diretamente a base de dados.

A diferença mais imediata está na estrutura do código, principalmente por terem sido removidas grande parte de instruções *try/catch* que decorriam do tratamento de exceções que são lançadas pelos métodos da interface JDBC por já não serem necessárias, visto que o Hibernate efetua o tratamento desse tipo de exceções internamente.

Outra das diferenças mais notórias é também na forma como as consultas são criadas nas duas linguagens e principalmente na forma como os resultados das consultas são obtidos e tratados. No caso do JDBC utiliza-se a classe `ResultSet` que engloba os resultados das consultas, independentemente do tipo dos resultados (se são únicos, se são

listas, entre outros). No caso da implementação com Hibernate, devido às funcionalidades da linguagem HQL pode-se definir qual será a estrutura de dados Java que melhor se adequa ao tipo de resultados da consulta. No exemplo da listagem 3.7 pode observar-se que os resultados das duas consultas são organizados numa estrutura do tipo de lista ligada, em pares que são definidos na consulta.

A última diferença notória entre estas duas implementações, é a redução visível do número de linhas de código utilizadas para implementar a operação. Isto principalmente deve-se que pelas mudanças descritas acima se utilizaram menos linhas para representar as mesmas funcionalidades.

3.2.4 Utilização da *Second level cache*

Para se poder criar uma versão do *benchmark* TPC-W em que se se utilize a *second level cache* houve a necessidade também de se procederem a algumas alterações no código das classes que representam as entidades do *benchmark*. Designadamente, há que introduzir meta-dados nas classes cujas entidades devem ser mantidas pela *second level cache* e também qual a estratégia de *caching* para as entidades. Esta informação é introduzida, colocando-se duas anotações junto do início da declaração da cache: *@Cacheable* e *@Cache()*.

No caso da adaptação apresentada neste trabalho escolheu-se colocar em cache todas as entidades, tendo-se escolhido como estratégia de *caching* o modo *read-write*. Como consequência, colocaram-se as anotações *@Cacheable* e *@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)* no início da declaração de cada classe.

Para além disto, também houve a necessidade de alterar o ficheiro de configuração do Hibernate, conforme está descrito na secção 2.2.5.3 do capítulo 2.

3.2.5 Utilização da *Query cache*

Para se criar uma versão do TPC-W que utilize a *query cache* há que necessariamente utiliza-se a *second level cache*, pelas razões mencionadas no capítulo 2. Portanto, para além das alterações necessárias para se utilizar a *second level cache*, há que adicionar em cada consulta efetuada no código das operações do *benchmark* uma chamada ao método *.setCacheable(true)*, que indica que a consulta deve ser mantida pela *query cache*.

Também houve a necessidade de se alterar o ficheiro de configuração do Hibernate, conforme está descrito na secção 2.2.5.2 do capítulo 2.

4

Avaliação

Neste capítulo apresenta-se mais detalhadamente o processo de testes que foi efetuado para realizar a avaliação do desempenho da ferramenta ORM Hibernate.

Apresenta-se uma simples aplicação de *benchmarking* que foi construída utilizando Hibernate e também uma base de dados MySQL através de JDBC, que representa operações sobre um esquema de base de dados simplificado. Foram feitos alguns testes que utilizam várias combinações possíveis de componentes da arquitetura do Hibernate, de modo a avaliar as diferenças de performance de cada tipo de operação individualmente.

Aborda-se também como se utilizou o *benchmark* TPC-W, adaptado tal como descrito no capítulo 3, na realização de testes de performance para avaliar o desempenho do sistema.

Também é feita uma análise a todos os resultados obtidos, tendo em conta os vários ambientes onde foram executados, as características de cada teste e também as características de cada arquitetura utilizada.

4.1 Âmbito dos Testes

Para se proceder à avaliação do impacto na performance que decorre da utilização de ferramentas similares ao Hibernate, realizaram-se uma série de testes, onde foi utilizado um *microbenchmark*, desenvolvido no âmbito deste trabalho, e a adaptação do *benchmark* TPC-W, abordada na secção 3.

Para também se perceber que componentes do Hibernate tinham mais impacto na performance, utilizaram-se versões de cada teste com diferentes combinações destes componentes, designadamente uma versão que utiliza a *second level cache* e outra versão que utiliza a *second level cache* juntamente com a *query cache*. No caso dos *microbenchmarks*, não

se utilizou a versão do teste que utiliza Hibernate com *query cache*, pois no contexto do teste não faria sentido, visto que não são utilizadas consultas HQL. A figura 4.1 apresenta esquematicamente todas as arquiteturas que se utilizaram na fase de testes.

Os testes foram realizados em duas máquinas com ambientes distintos, denominadas de máquinas A, B e C:

- Máquinas A e B - 2 processadores AMD Opteron 2376, cada um com 4 cores a 2.2GHz com 16GB de RAM
- Máquina C - 4 processadores AMD Opteron(TM) Processor 6272, cada um com 16 cores a 2.1GHz com 64GB de RAM
- A interconexão entre os nós é de *ethernet gigabit*

Para todos os testes foram utilizadas bases de dados MySQL, tendo sido utilizada a versão 5.6.14 da edição *Community Server*. A versão do *driver* JDBC que é utilizada em todos os testes é a 5.1. Nos testes com Hibernate, é utilizada a versão 4.2.3.Final. Todos os testes correm em OpenJDK Java, na versão 1.6.0_18 para 64 bits.

4.1.1 *Microbenchmark*

Todas as funções que compõem as aplicações multi-camada são, normalmente compostas de operações mais simples. Normalmente estas operações são fornecidas pelas ferramentas de persistência e enquadram as funcionalidades básicas de manipulação dos dados na camada de persistência. Tais operações básicas podem-se dividir em operações de consulta, inserção, remoção e modificação dos dados.

Para que se possa verificar se existem diferenças significativas do desempenho de cada uma das operações básicas, foi implementado um *benchmark* que recolhe os tempos de execução de operações de cada tipo, para cada variante da ferramenta Hibernate que está em análise – variante *vanilla* e variante que utiliza a *second level cache* – e uma versão que utiliza diretamente uma base de dados *MySQL*, através da interface *JDBC*.

4.1.1.1 Estrutura

Este *benchmark* utiliza uma estrutura de dados simples, onde apenas se encontra uma classe denominada `Foo` que é composta por quatro atributos, todos do tipo inteiro, `bara`, `barb`, `barc` e `bard`. Na base de dados isto foi refletido para uma tabela denominada também de `Foo` com os mesmos atributos, onde a chave primária da tabela é o atributo `bara`. As chaves primárias da tabela `Foo` são geradas incrementalmente pelo sistema de gestão de base de dados.

4.1.1.2 Operações

Para cada variante deste *microbenchmark* foi implementado uma operação de cada tipo dos referido acima: inserção, remoção, consulta e atualização de dados. As implementações utilizaram sempre a interface disponível para a ferramenta da versão em causa.

Especificamente, no caso das inserções, na implementação das versões com Hibernate criou-se um objeto da classe `Foo` com valores aleatórios (à exceção da chave primária) e depois utilizou-se o método `Session.save()` para persistir as alterações a esse objeto na camada de persistência associada ao Hibernate. No caso da versão MySQL utilizou-se um `PreparedStatement` com a *query* `INSERT INTO FOO (barb, barc, bard) VALUES (?,?,?)`, em que os `?` são substituídos por valores aleatórios, sendo depois utilizado o método `executeUpdate(query)` na instância `PreparedStatement` que foi previamente criada, para que a *query* seja executada.

No caso das remoções, nas versões com Hibernate, primeiro obtém-se o objeto através do método `Session.get()`, passando-se depois como por parâmetro para o método `Session.remove()`, que vai remover o objeto. Na versão MySQL, utiliza-se também um `PreparedStatement`, com a *query* `DELETE FROM FOO WHERE bara=?`, onde o `?` é substituído por um valor aleatório entre 1 e 1000. Para a *query* ser executada há que se invocar o método `executeUpdate()` na instância da classe `PreparedStatement`.

Para as modificações, nas versões com Hibernate, primeiro obtém-se o objeto através do método `Session.get()` e depois alteram-se três atributos do objeto, `barb`, `barc` e `bard`, com valores aleatórios. Para a versão MySQL é utilizado também um `PreparedStatement`, à semelhança das outras operações, mas neste caso é utilizada a *query* `UPDATE FOO SET barb=?, barc=?, bard=? WHERE bara=?`, onde os caracteres `?` são substituídos pelos valores gerados aleatoriamente.

Para a operação de consulta dos dados, na versão com o Hibernate apenas se gerou um número aleatório que indica a chave primária e depois utiliza-se o método `Session.get()` para obter o objeto cujo identificador é o valor gerado. Para a versão MySQL, o processo foi análogo: gerou-se um identificador e depois utilizou-se a *query* `SELECT * FROM FOO WHERE BARA=?` para a obtenção dos dados, onde o carácter `?` é substituído pelo valor do valor gerado.

Todas as operações foram executadas mil vezes, dentro da mesma *thread*, sempre dentro de uma única transação. Para se conseguir executar as operações todas numa transação, é necessário que, no caso do Hibernate, as operações sejam todas efetuadas dentro de uma instância *Session* ativa, em que seja aberta também uma *Transaction*. No caso em que se utiliza JDBC é apenas necessário garantir que o *PreparedStatement* que foi criado seja reutilizado entre execuções da operação. As listagens 4.1 e 4.2 e demonstram como foi feito o processo, utilizando a operação de inserção.

De referir também que na versão do *microbenchmark* que utiliza o Hibernate com a *second level cache*, foi apenas necessário adicionar ao código da classe *Foo* as anotações `Cacheable` `Cache(usage = CacheConcurrencyStrategy.READ_WRITE)`.

Listagem 4.1: Implementação do teste para a operação de *insert* no *microbenchmark*, utilizando Hibernate.

```
public static void insertTx(){
    Random r = new Random();
    Session session = SessionFactoryUtil.getSessionFactory().openSession();
    Transaction tx = session.beginTransaction();

    for(int i=0;i<1000;i++){
      Foo f = new Foo(r.nextInt(),r.nextInt(),r.nextInt());
      session.save(f);
    }

    tx.commit();
    session.close();
  }
```

Listagem 4.2: Implementação do teste para a operação de *insert* no *microbenchmark*, utilizando a API JDBC.

```
public static void insertTx(){
    Random r = new Random();
    Connection conn;
    PreparedStatement prep;
    String query = "INSERT INTO FOO (barb , barc, bard) VALUES (?,?,?)";
    try {

      conn = getConnection();
      conn.setAutoCommit(false);
      prep = conn.prepareStatement(query);

      for(int i=0;i<1000;i++){
        prep.setInt(1, r.nextInt());
        prep.setInt(2, r.nextInt());
        prep.setInt(3, r.nextInt());
        int rs = prep.executeUpdate();
      }
      conn.commit();
      prep.close();

    } catch (SQLException e) {
      e.printStackTrace();
    }
  }
```

4.1.2 TPC-W

Para se avaliar a performance das aplicações que utilizam Hibernate, e os impactos no desempenho e na escalabilidade da utilização dos múltiplos componentes disponibilizados pela ferramenta, houve a necessidade de se utilizar um teste que seja mais exaustivo que o *microbenchmark* apresentado anteriormente e que simulasse o *workload* de uma aplicação real.

Optámos por utilizar o *benchmark* TPC-W que está descrito no capítulo 3, para se efetuar um conjunto de testes com várias condições de arquitetura que tentaram reproduzir as condições de um ambiente onde a ferramenta Hibernate possa ser utilizada.

Neste caso executaram-se quatro combinações de arquitetura: TPC-W que acede diretamente uma base de dados MySQL, TPC-W Hibernate que não utiliza nenhuma cache, denominado nestes testes Hibernate *vanilla*, TPC-W que utiliza Hibernate com a *second level cache* ativa e o TPC-W que utiliza Hibernate com a *second level cache* e a *query cache* activas.

Para cada uma destas versões executaram-se testes com dois *workloads* disponíveis para o *benchmark*, descritos na secção 3.1.2 do capítulo 3, nomeadamente o *browsing* e o *ordering*, com 3000 operações e com um número de *threads* variável de 1 a 64.

4.2 Resultados

Esta secção apresenta todos os resultados dos testes anteriormente descritos, quer para o *microbenchmark* quer para o TPC-W.

4.2.1 *Microbenchmark*

No caso do *microbenchmark* os resultados são apresentados em milisegundos, e apresentam-se agrupados por tipo de operação.

As figuras 4.2 e 4.3 apresentam os resultados dos testes que foram executados com a aplicação cliente na máquina A, sendo que na primeira, o sistema de gestão de base de dados se encontra a correr na mesma máquina, e na segunda o sistema de gestão de base de dados está a correr na máquina B, que tem as mesmas características que a A.

As figuras 4.4 e 4.5 apresentam os resultados dos testes que foram executados com a aplicação cliente na máquina C, sendo que à semelhança dos resultados anteriores, na primeira o sistema de gestão de base de dados se encontra a correr na mesma máquina, e na segunda, o sistema de gestão de base de dados está a correr na máquina A.

4.2.2 TPC-W

No caso dos testes com a *benchmark* TPC-W todos os resultados são apresentados em operações por segundo, e realizaram-se testes com 1, 2, 4, 8, 16, 32 e 64 clientes.

As figuras 4.6, 4.7, 4.8 e 4.9 apresentam os resultados com para o *workload browsing*. Mais especificamente, a figura 4.6 apresenta os resultados dos testes executados com a

aplicação cliente na máquina A com o sistema de gestão de base de dados a correr na mesma máquina, e a figura 4.7 apresenta os resultados executados com a aplicação cliente na máquina A mas com o sistema de gestão de base de dados a correr na máquina C. A figura 4.8 apresenta os resultados dos testes executados com a aplicação cliente na máquina C e com o sistema de base de dados a correr na mesma máquina, e a figura 4.9 apresenta os resultados dos testes que executaram com a aplicação cliente na máquina C mas com o sistema de gestão de base de dados a correr na máquina A.

As figuras 4.10, 4.11, 4.12 e 4.13 apresentam os resultados dos testes efetuados com o *workload ordering*. Os testes foram executados em condições semelhantes às dos realizados com o *workload browsing*.

4.3 Análise dos Resultados

Observando os resultados obtidos com os diferentes testes, podemos tecer várias considerações, tendo em conta o âmbito e os objetivos de cada tipo de teste. Nesta secção apresentam-se as conclusões que foram obtidas a partir da observação dos resultados dos testes que foram efetuados.

4.3.1 *Microbenchmarks*

Analisando os resultados obtidos para os testes realizados com os *microbenchmarks* executados na máquina A com base de dados local, podemos observar que no contexto deste tipo de operações mais simplificadas, a utilização da *second level cache* prejudica um pouco a performance, sendo que são os resultados piores para este teste. Este comportamento é observado apenas em ambiente com a base de dados local (figura 4.2), pois no teste com a base de dados remota (figura 4.3) a utilização desta cache apresentou alguns ganhos, colmatando assim algum *overhead* causado pelo aumento do tempo de comunicação com a base de dados. Em ambos os testes, os resultados melhores foram os apresentados pela versão que utiliza a base de dados MySQL diretamente, sendo que se observaram poucas alterações na performance quando a base de dados foi deslocada para outro nó.

No contexto dos testes executados com a máquina C, no teste com a base de dados local (figura 4.4) pode-se observar um ganho grande de performance em todos os resultados, comparativamente ao teste análogo com a máquina A (figura 4.2). No entanto mantém se o mesmo padrão em relação aos resultados referentes aos diferentes ambientes, sendo que a utilização da *second level cache* continua a ter algum impacto na performance.

Quanto aos resultados dos testes com a base de dados remota (figura 4.5), estes apresentaram valores ligeiramente melhores do que os do teste efetuado na máquina A com a base de dados remota (figura 4.3). Esta diferença de resultados pode dever-se ao facto das máquinas onde o sistema de bases de dados esteve a executar terem configurações de *hardware* idênticas.

4.3.2 TPC-W

Observando-se os resultados referentes aos testes que executaram o *benchmark* TPC-W utilizando o *workload browsing*, os testes executados com a aplicação cliente na máquina A e com base de dados local (figura 4.6) indiciam que a versão do MySQL é a única que não apresenta escalabilidade, sendo que para 64 *threads* até apresenta resultados ligeiramente piores que a versão Hibernate *vanilla*. A versão que utiliza Hibernate *vanilla* apresenta pouca escalabilidade. Quanto às outras versões com Hibernate, neste teste verifica-se que a *second level cache* e a *query cache* beneficiam bastante a escalabilidade do sistema. Neste teste verifica-se inclusivé que a versão que utiliza a *query cache* apresenta resultados melhores que a versão que só utiliza *second level cache*, à medida que aumenta o número de *threads*. Isto pode estar relacionado com o facto das operações deste *workload* serem principalmente de leitura. Portanto, pode-se concluir que para aplicações onde exista um grande número de consultas a utilização de *second level cache* e *query cache* pode ser a configuração que beneficia mais a performance. Os testes que executaram com a aplicação cliente na máquina A e que utilizam a base de dados remota (figura 4.7) mantêm as mesmas características, se bem que há um aumento no número de *threads* necessárias para os resultados começarem a escalar. Nestes resultados é bem evidente o impacto de performance causado pela utilização da *query cache*.

Utilizando o *workload browsing*, quanto aos resultados que executaram com a aplicação cliente na máquina C, mantém se os mesmos padrões de escalabilidade para todas as versões testadas. No caso do teste em que se utiliza a base de dados local (figura 4.8), verificou-se um aumento do desempenho, à medida que as *threads* do teste aumentam, sendo que o pico de performance para os testes que utilizam Hibernate com a *second level cache* e a *query cache* é só atingido nos 32 *threads*, enquanto que para o mesmo teste, executado com a aplicação cliente na máquina A (figura 4.6), o pico destes resultados está nas 8 *threads*. Também é mais evidente a perda de performance que a versão que utiliza MySQL apresenta, com o aumento do número de *threads*, sendo que com 32 *threads*, a versão com MySQL apresenta resultados piores que a versão que utiliza Hibernate *vanilla*. Nos testes que correram com a base de dados remota (figura 4.9) mantém-se o mesmo padrão que os testes executados na mesma máquina com base de dados local, apenas se verifica um ligeiro aumento no número de operações para os testes que utilizam *second level cache* e *query cache*.

Quanto aos testes que utilizam o *workload ordering* e que são executados com a aplicação cliente na máquina A e com base de dados local (figura 4.10), observa-se o mesmo padrão dos resultados anteriores. A diferença principal nestes resultados é que a *query cache* já não oferece tantos benefícios para a performance, comparando estes resultados com os das mesmas condições, mas utilizando o *workload browsing*, principalmente porque no *workload ordering* há menos operações de leitura. Nos testes com com a aplicação cliente na máquina A, em que a base de dados está remota (figura 4.11), os resultados

são similares, exceto que de entre as versões que utilizam Hibernate, a que tem melhor desempenho é a que utiliza a *second level cache*.

Nos testes com a aplicação cliente na máquina C, em que a base de dados está a correr na mesma máquina (figura 4.12), continua a verificar-se o mesmo padrão de resultados, e a *second level cache* apresenta os melhores resultados do teste, exceto com 64 *threads*. Neste teste os resultados com a versão do Hibernate *vanilla* escalam um pouco melhor. Com a base de dados remota 4.13), obtiveram-se resultados melhores, e as versões do Hibernate com *second level cache* e *query cache* continuam a apresentam resultados melhores, sendo a versão com *second level cache* novamente a apresentar os melhores resultados dos testes. Tendo em conta que na maioria dos resultados com o *workload ordering* a versão do Hibernate com *second level cache* apresentou os melhores resultados, pode-se concluir que esta configuração é a mais adequada para aplicações com muitas operações de escrita de dados.

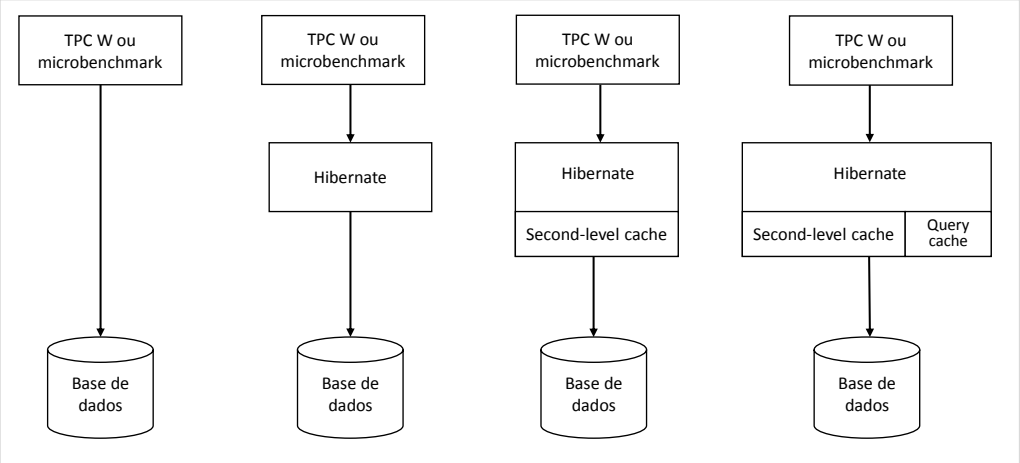


Figura 4.1: Esquema representativo das várias arquiteturas utilizadas durante a fase de testes.

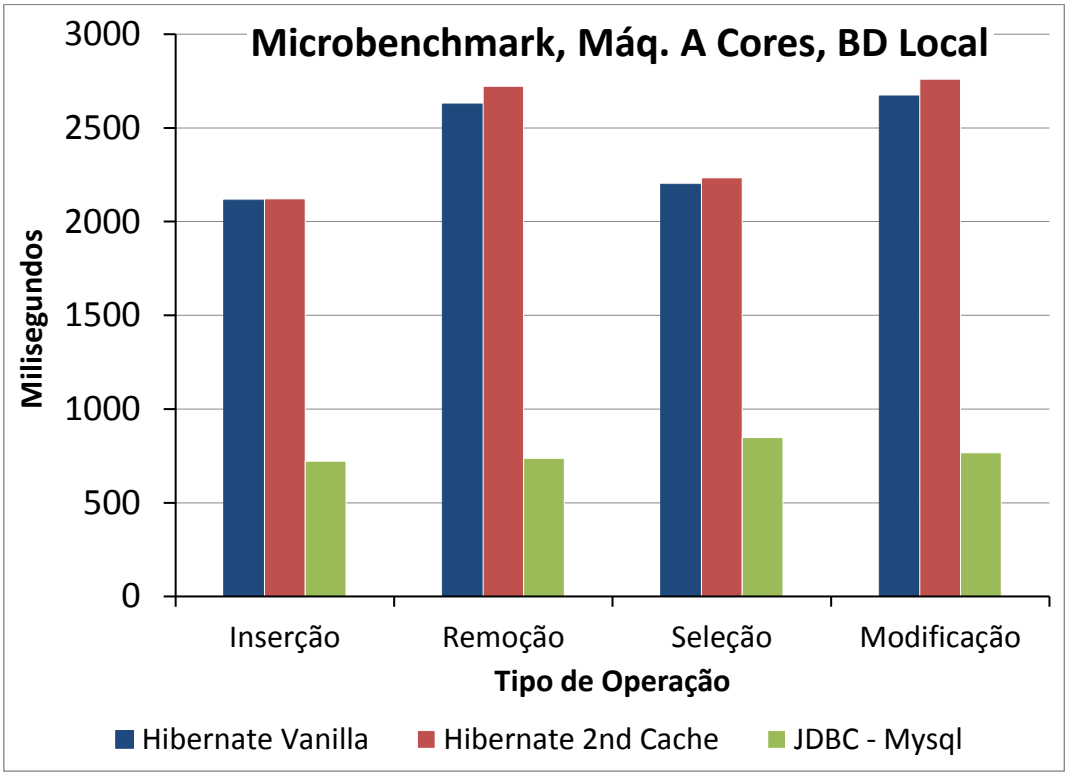


Figura 4.2: Resultados do *microbenchmark* executados com a aplicação cliente na máquina A e com a base de dados localizada na mesma máquina.

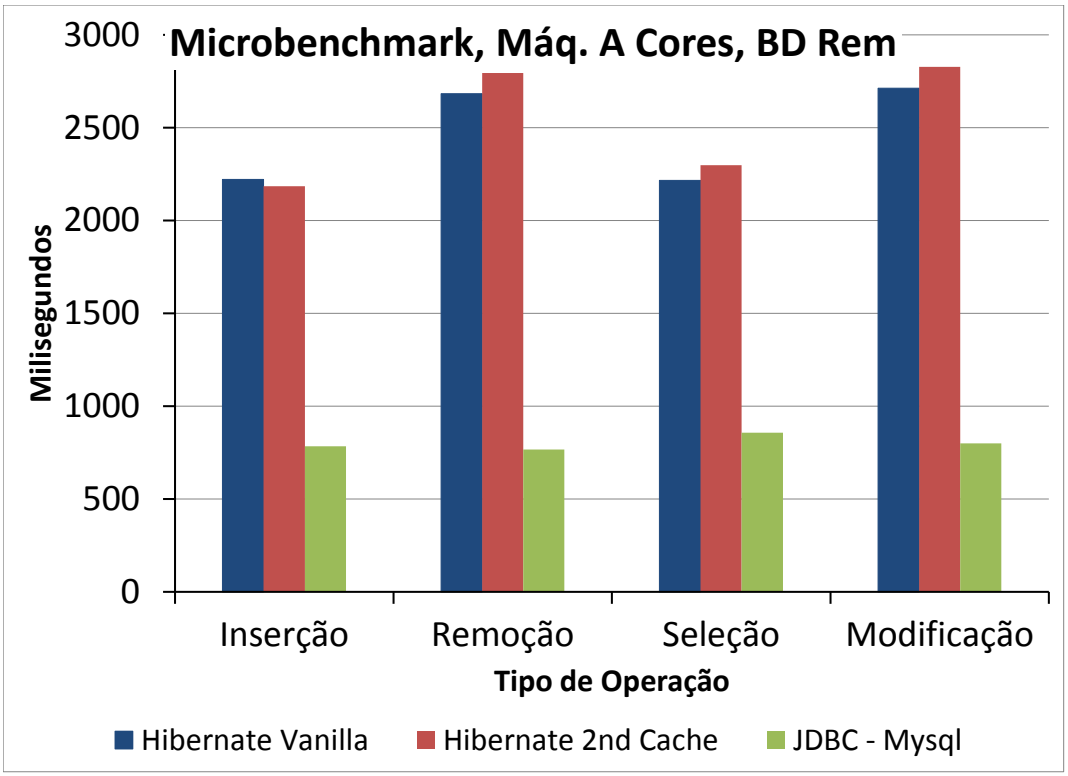


Figura 4.3: Resultados do *microbenchmark* executados com a aplicação cliente na máquina A e com a base de dados localizada na máquina B.

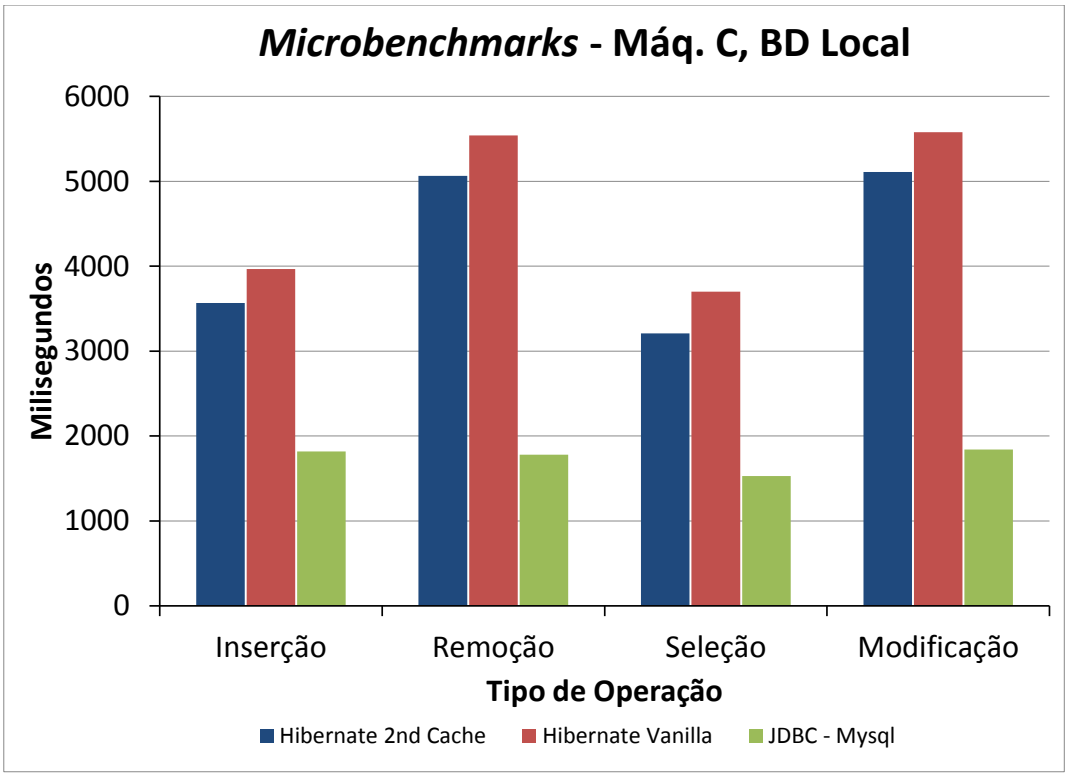


Figura 4.4: Resultados do *microbenchmark* executados com a aplicação cliente no na máquina C e com a base de dados localizada na mesma máquina.

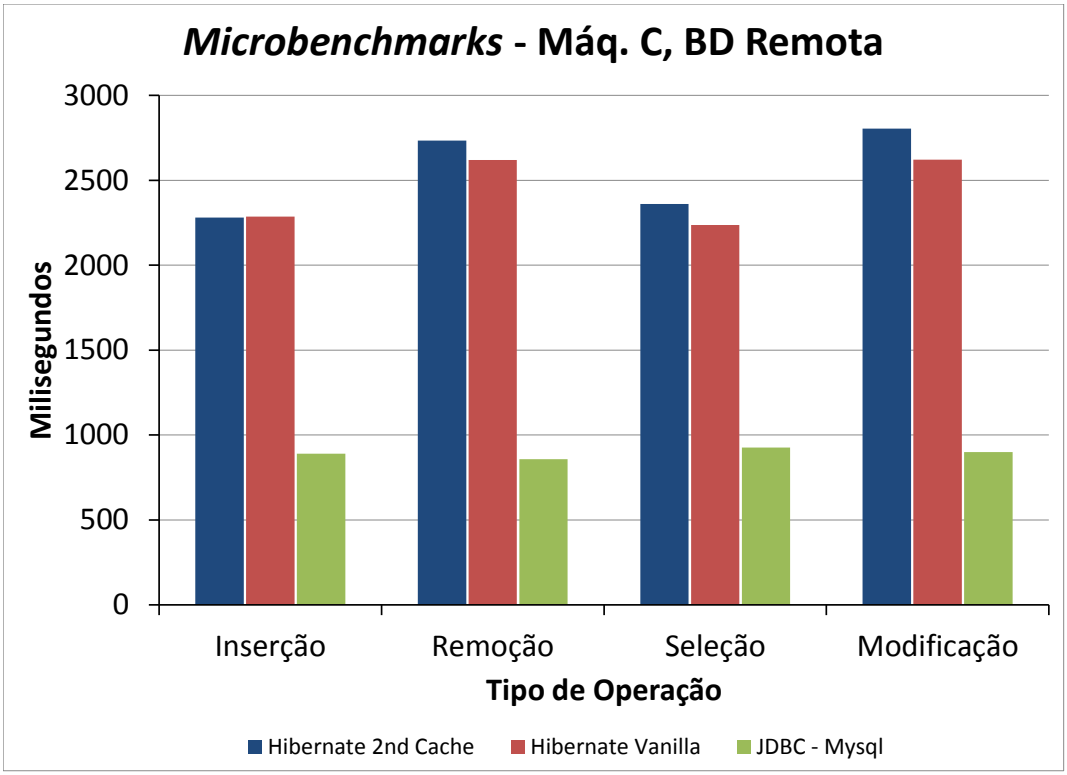


Figura 4.5: Resultados do *microbenchmark* executados com a aplicação cliente na máquina C e com a base de dados localizada na máquina A.

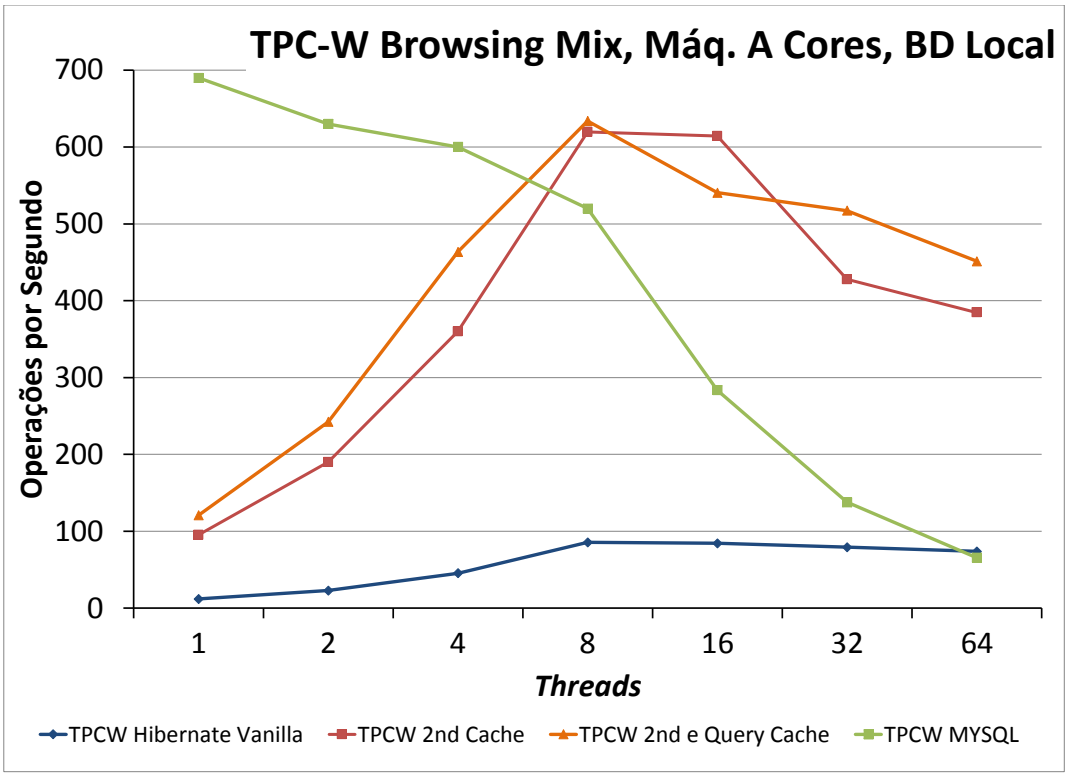


Figura 4.6: Resultados para o *benchmark* TPC-W executados com a aplicação cliente na máquina A e com a base de dados localizada na mesma máquina, utilizando o *workload browsing*.

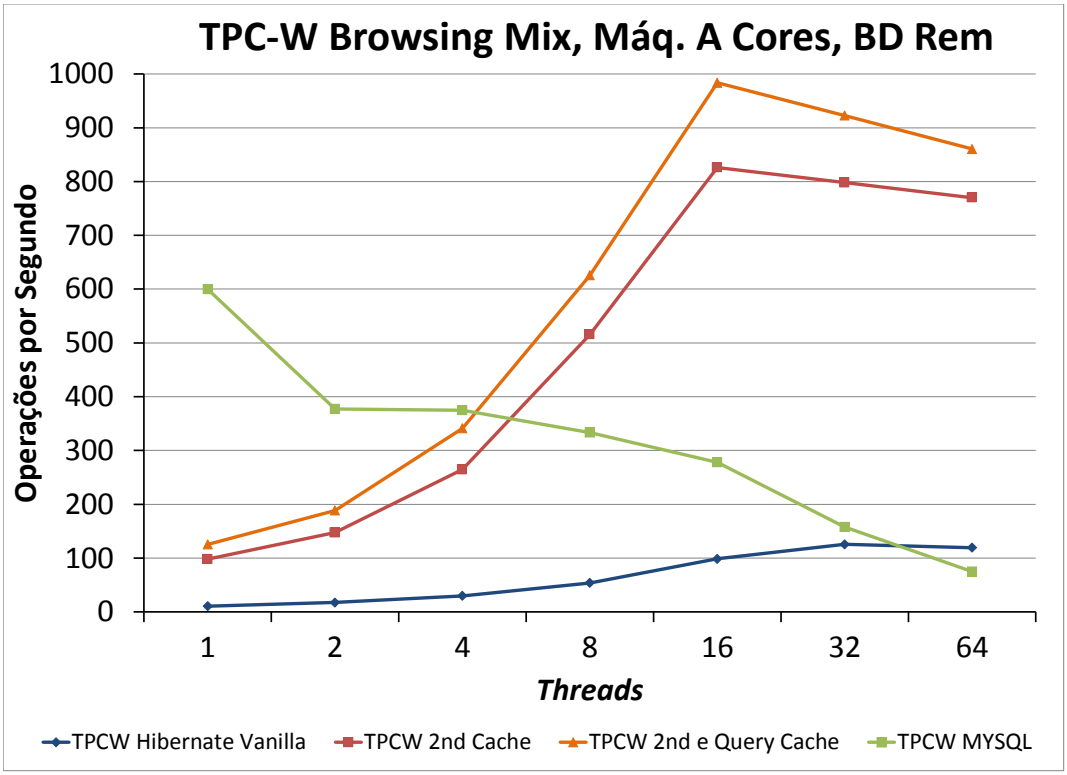


Figura 4.7: Resultados para o *benchmark* TPC-W executados com a aplicação cliente na máquina A e com a base de dados localizada na máquina B, utilizando o *workload browsing*.

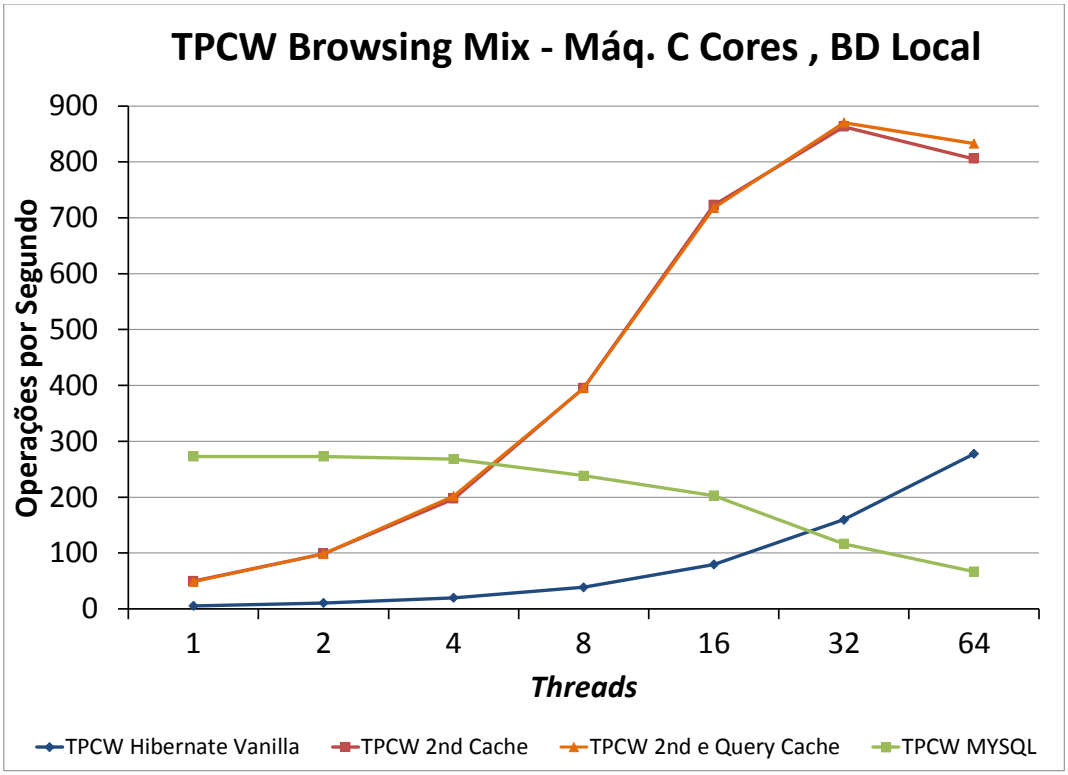


Figura 4.8: Resultados para o *benchmark* TPC-W executados com a aplicação cliente na máquina C e com a base de dados localizada na mesma máquina, utilizando o *workload browsing*.

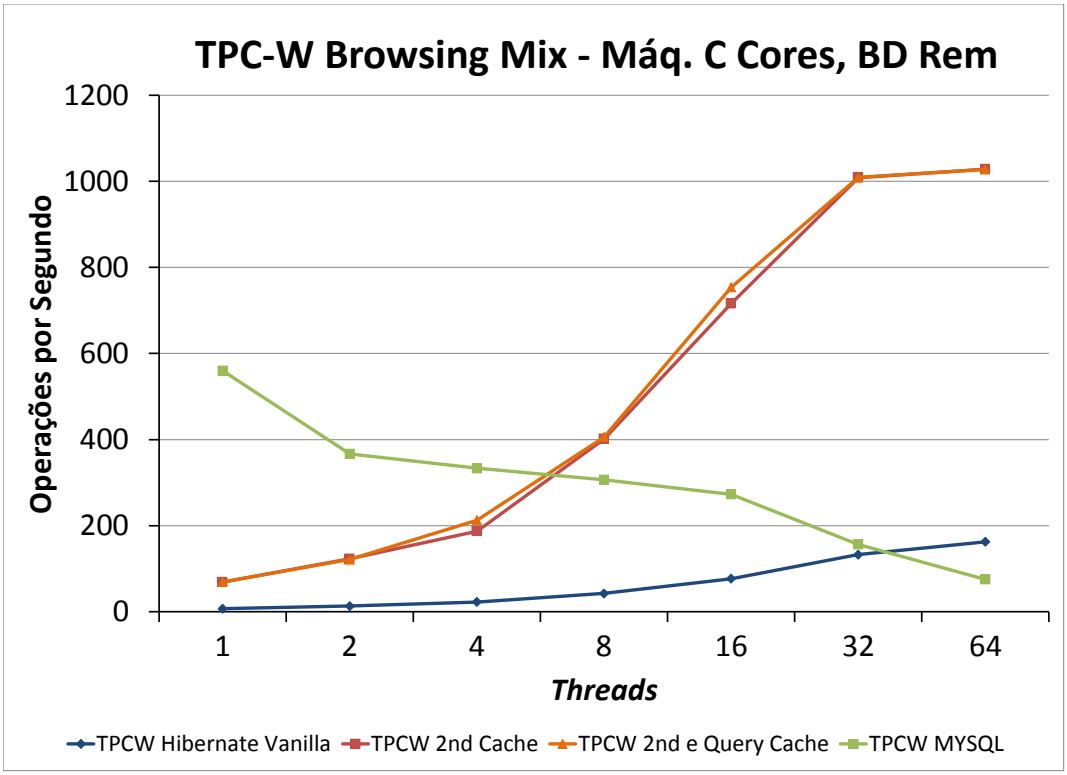


Figura 4.9: Resultados para o *benchmark* TPC-W executados com a aplicação cliente na máquina C e com a base de dados localizada na máquina A, utilizando o *workload browsing*.

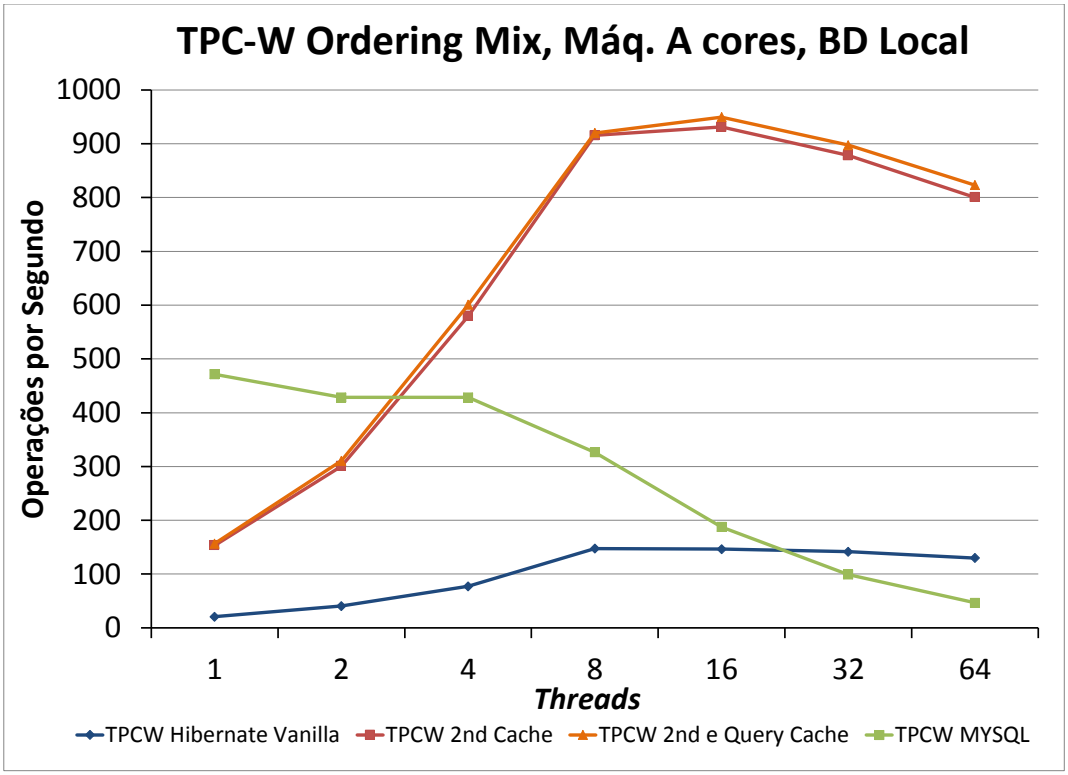


Figura 4.10: Resultados para o *benchmark* TPC-W executados com a aplicação cliente no na máquina A e com a base de dados localizada na mesma máquina, utilizando o *workload ordering*.

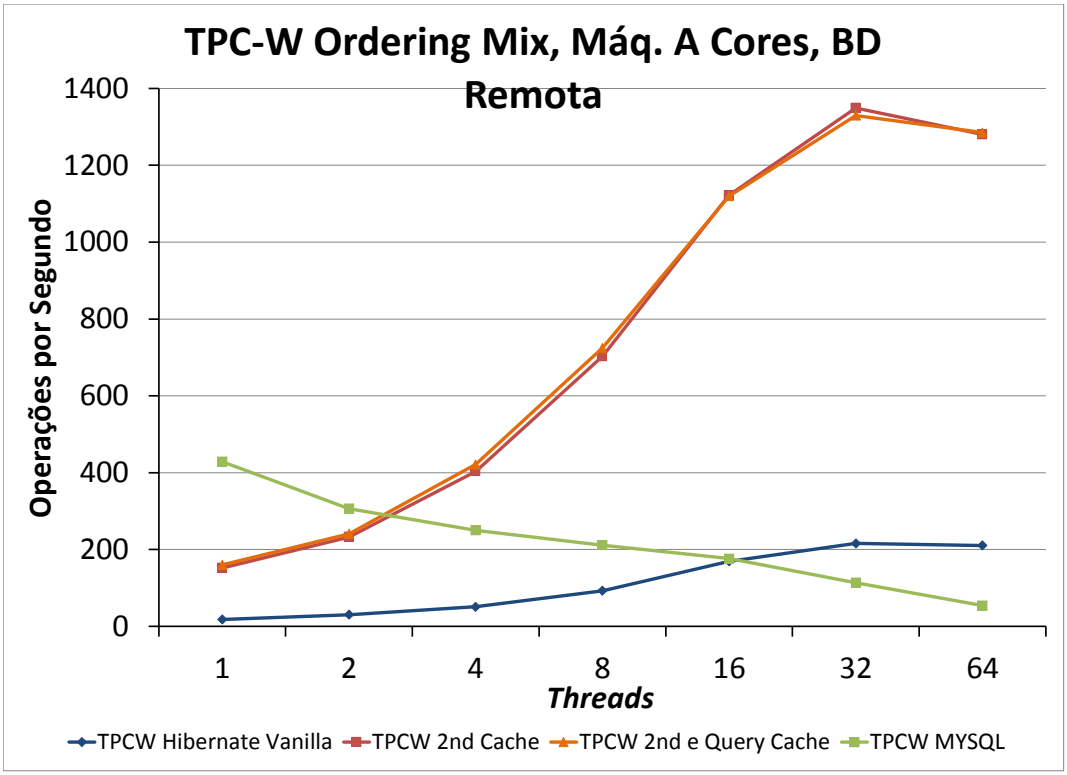


Figura 4.11: Resultados para o *benchmark* TPC-W executados com a aplicação cliente na máquina A e com a base de dados localizada na máquina B, utilizando o *workload ordering*.

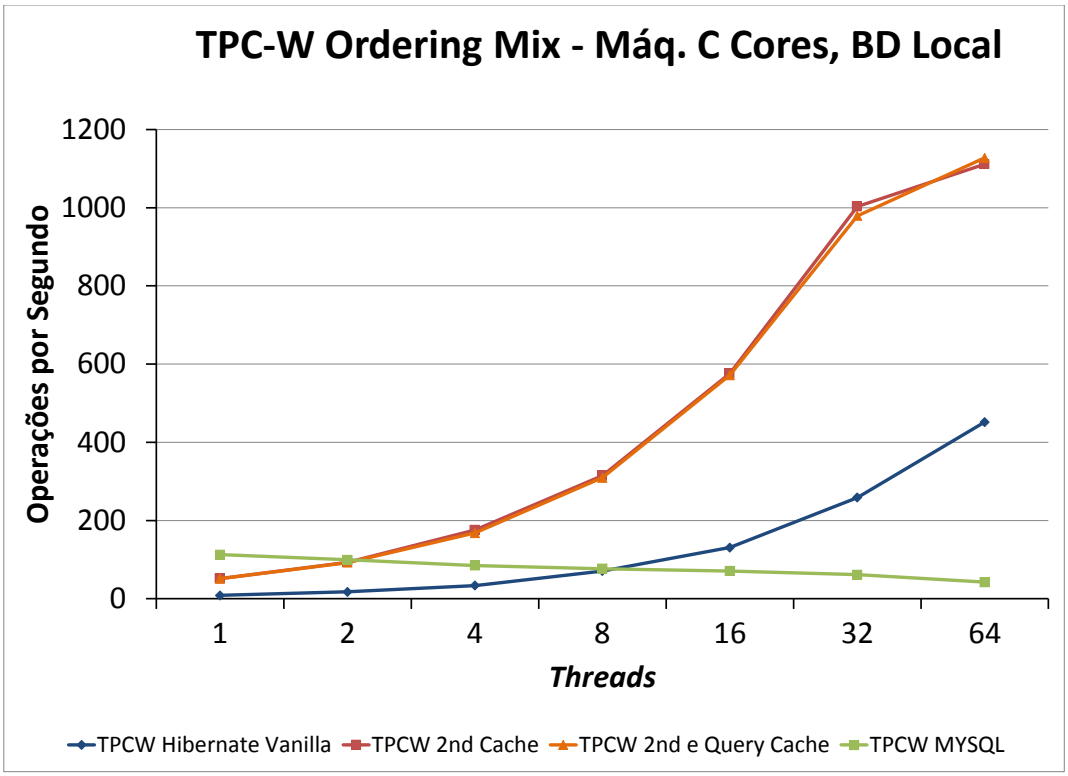


Figura 4.12: Resultados para o *benchmark* TPC-W executados com a aplicação cliente na máquina C e com a base de dados localizada na mesma máquina, utilizando o *workload ordering*.

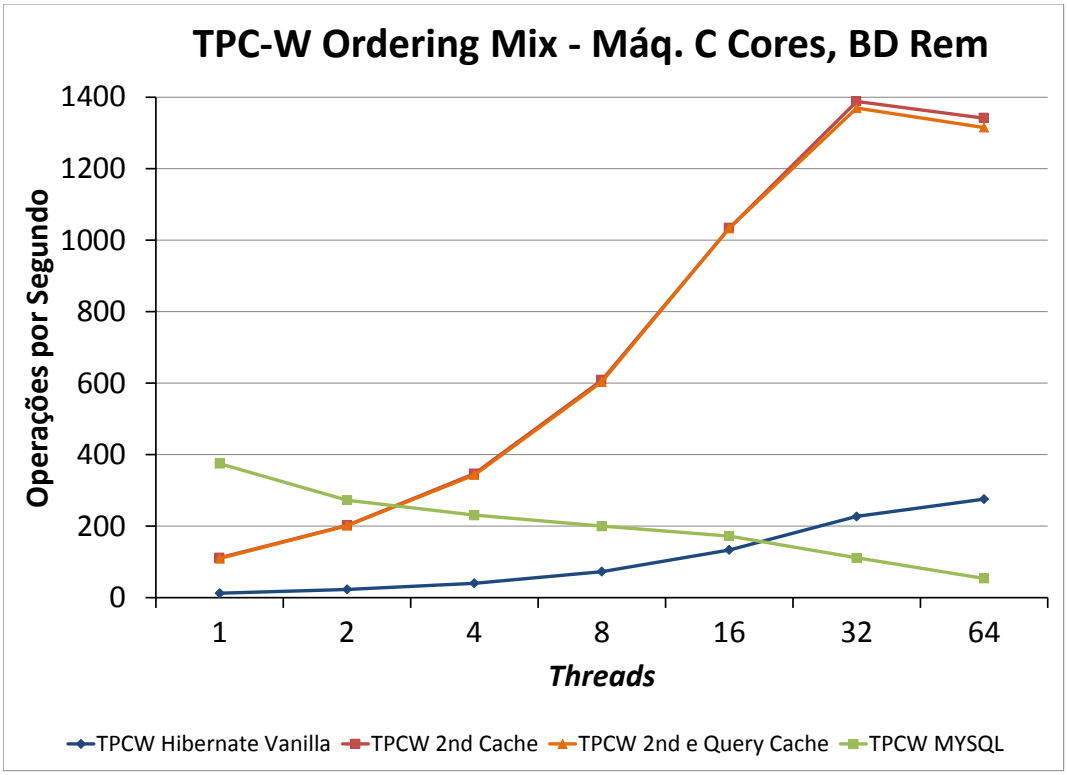


Figura 4.13: Resultados para o *benchmark* TPC-W executados com a aplicação cliente na máquina C e com a base de dados localizada na máquina A, utilizando o *workload ordering*.

5

Conclusão

5.1 Observações Finais

As ferramentas de mapeamento objeto/relacional são um dos componentes mais importantes na arquitetura das aplicações multi-camada que são construídas com linguagens orientadas a objetos e que necessitam de utilizar bases de dados relacionais. Apesar das diferenças existentes entre o modelo orientado a objetos e o modelo relacional, verifica-se que as ferramentas ORM conseguem superar muitas das dificuldades causadas por essas diferenças, e também facilitam o desenvolvimento destas aplicações.

Estas ferramentas, que se apresentam como um *middleware* entre a base de dados e a aplicação, podem facilitar o processo de desenvolvimento de aplicações multi-camada, mas impõem necessariamente custos no seu desempenho.

Neste trabalho procurou-se analisar quais as soluções arquiteturais disponibilizadas por uma ferramenta ORM, o Hibernate, que permitiam obter os melhores resultados de performance e escalabilidade de acordo com o *workload* típico de uma aplicação multi-camada padrão.

Apresentou-se por isso uma implementação deste *benchmark* que faz uso dos mecanismos do Hibernate para persistir os seus dados numa base de dados relacional, no SGBD MySQL. Também se apresentaram duas versões do *benchmark* TPC W que utilizam o Hibernate com os componentes que são disponibilizados para aumentar o desempenho das aplicações: uma versão que utiliza a *second level cache* e outra versão que utiliza a *second level cache* juntamente com a *query cache*.

Também se tentou perceber como alguns mecanismos que já vêm a ser utilizados para melhorar o desempenho de aplicações multi-camada podem contribuir para um aumento da performance e escalabilidade destas aplicações. Para isso estudou-se a utilização de

componentes suportados pelo Hibernate, utilizando as duas versões que implementam estes mecanismos e confrontaram-se os resultados com os anteriores. Verificou-se pela análise dos resultados obtidos para ambientes locais, que se as aplicações apresentarem um *workload* com muitas operações de leitura, a combinação da *second level cache* com a *query cache* oferece os melhores ganhos para a escalabilidade e performance da aplicação.

Por sua vez, se a aplicação apresentar um *workload* com muitas operações de escrita, a melhor configuração para aplicação será a utilização apenas da *second level cache*, pois verificou-se que nestes ambientes é a solução que garante a melhor escalabilidade e performance.

Em ambientes distribuídos também se verificam estas condições, tendo a utilização das caches demonstrado resultados de performance e escalabilidade um pouco melhores, sendo efetuadas mais operações por segundo. Nestes ambientes a utilização de caches reduz o número de acessos à base de dados remota, e como nestes casos estes acessos implicam que haja um tempo de atraso na receção dos dados derivado do RTT à máquina em que a base de dados se encontra, a utilização de caches beneficia muito a performance. Nestes casos, independentemente do *workload*, ambas as caches demonstraram ter resultados com poucas diferenças, excetuando um caso, em que a utilização da *query cache* num *workload* com muitas leituras beneficiou muito a performance da aplicação. Sendo que num contexto real é mais frequente a ocorrência de casos em que a base de dados está localizada numa máquina diferente da máquina que executa a aplicação, esta conclusão ajuda-nos a perceber que a utilização das caches das ferramentas ORM, não só mais benefícios para a performance, como garante que as aplicações ofereçam alguma escalabilidade, face a um aumento do número de utilizadores.

Verificou-se também que a utilização da *second level cache* é particularmente importante em *workloads* onde existam muitas operações de escrita pois ajuda a que as ferramentas ORM não dependam tanto dos mecanismos de controlo de concorrência das bases de dados, que por vezes podem ser ineficientes.

Em conclusão, este trabalho conseguiu quantificar as falhas presentes nas arquiteturas das ferramentas ORM que prejudicam o desempenho e a escalabilidade das aplicações multi-camada. Também foram apresentadas algumas configurações de componentes já suportados por estas ferramentas para que possam minimizar estas perdas.

5.2 Trabalho Futuro

No final deste trabalho pode-se revelar que surgiram algumas ideias finais quanto ao desenvolvimento de outros componentes para as ferramentas ORM que possam trazer outros benefícios à escalabilidade e performance das aplicações multi camada:

- Fazer um estudo mais exaustivo das versões do *benchmark* TPC-W executado num ambiente distribuído, sendo verificados os impactos de performance decorrentes da distribuição da aplicação de teste por várias máquinas. Também seria interessante

utilizar serviços de comunicação entre as máquinas que apresentem muito baixa latência, como o Infiniband;

- Adaptar outros *benchmarks* existentes que também simulem outro tipo de aplicações de contexto real, como por exemplo, o TPC-C;
- A criação de um componente que permitisse a resolução de conflitos de concorrência em memória, que recorresse aos mecanismos apresentados em [DL09]. Isto iria possibilitar que as ferramentas ORM deixassem de depender unicamente dos mecanismos de controlo de concorrência das bases de dados, que podem ser mais ineficientes;

Bibliografia

[Car+94] M. J. Carey, D. J. DeWitt, C. Kant e J. F. Naughton. "A status report on the OO7 OODBMS benchmarking effort". Em: *Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications*. OOPSLA '94. Portland, Oregon, USA: ACM, 1994, pp. 414–426. ISBN: 0-89791-688-3. DOI: 10.1145/191080.191147. URL: http://doi.acm.org/10.1145/191080.191147.

[Car+08] N. Carvalho, J. Cachopo, L. Rodrigues e A. R. Silva. "Versioned transactional shared memory for the FénixEDU web application". Em: *Proceedings of the 2nd workshop on Dependable distributed data management - SDDDM '08* (2008), pp. 15–18. DOI: 10.1145/1435523.1435526.

[CFG09] L. Charles, P. Felber e C. Gête. "TMBean: Optimistic Concurrency in Application Servers Using Transactional Memory". Em: *On the Move to Meaningful Internet Systems: OTM 2009*. Ed. por R. Meersman, T. Dillon e P. Herrero. Vol. 5870. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 484–496. ISBN: 978-3-642-05147-0. DOI: 10.1007/978-3-642-05148-7_37.

[Cou01] T. P. P. Council. *TPC Benchmark W (Web Commerce) – Specification*. Agosto de 2001. URL: http://www.tpc.org/tpcw/spec/tpcw_v101.pdf.

[CJ10] S. Cvetković e D. Janković. "A comparative study of the features and performance of ORM tools in a .NET environment". Em: *Proceedings of the Third international conference on Objects and databases*. ICOODB'10. Frankfurt/Main, Germany: Springer-Verlag, 2010, pp. 147–158. ISBN: 3-642-16091-3, 978-3-642-16091-2. URL: http://dl.acm.org/citation.cfm?id=1926241.1926257.

[DS02] J. Darmont e M. Schneider. "Advanced topics in database research vol. 1". Em: ed. por K. Siau. Hershey, PA, USA: IGI Publishing, 2002. Cap. Object-oriented database benchmarks, pp. 34–57. ISBN: 1-930708-41-6. URL: http://dl.acm.org/citation.cfm?id=960129.960133.

[DL09] R. J. Dias e J. M. Lourenço. "Unifying Memory and Database Transactions". Em: *Euro-Par 2009 Parallel Processing*. Ed. por H. Sips, D. Epema e H.-X. Lin. Vol. 5704. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 349–360. ISBN: 978-3-642-03868-6. DOI: 10.1007/978-3-642-03869-3_35. URL: http://dx.doi.org/10.1007/978-3-642-03869-3_35.

[DU11] S. Dietrich e S. Urban. *Fundamentals of Object Databases: Object-Oriented and Object-Relational Design*. Synthesis Lectures on Data Management Series. Morgan & Claypool, 2011. ISBN: 9781608454761.

[Ire+09] C. Ireland, D. Bowers, M. Newton e K. Waugh. "A Classification of Object-Relational Impedance Mismatch". Em: *Advances in Databases, Knowledge, and Data Applications, 2009. DBKDA '09. First International Conference on*. 2009, pp. 36–43. DOI: 10.1109/DBKDA.2009.11.

[Men02] D. Menasce. "TPC-W: a benchmark for e-commerce". Em: *Internet Computing, IEEE* 6.3 (mai. de 2002), pp. 83–87. ISSN: 1089-7801. DOI: 10.1109/MIC.2002.1003136.

[ONe08] E. J. O'Neil. "Object/relational mapping 2008: hibernate and the entity data model (edm)". Em: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. SIGMOD '08. Vancouver, Canada: ACM, 2008, pp. 1351–1356. ISBN: 978-1-60558-102-6. DOI: 10.1145/1376616.1376773. URL: http://doi.acm.org/10.1145/1376616.1376773.

[Por+10] D. R. K. Ports, A. T. Clements, I. Zhang, S. Madden e B. Liskov. "Transactional consistency and automatic management in an application data cache". Em: *Proceedings of the 9th USENIX conference on Operating systems design and implementation*. OSDI'10. Vancouver, BC, Canada: USENIX Association, 2010, pp. 1–15. URL: http://dl.acm.org/citation.cfm?id=1924943.1924963.

[ST95] N. Shavit e D. Touitou. "Software transactional memory". Em: *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*. PODC '95. Ottowa, Ontario, Canada: ACM, 1995, pp. 204–213. ISBN: 0-89791-710-3. DOI: 10.1145/224964.224987. URL: http://doi.acm.org/10.1145/224964.224987.

[Siv+07] S. Sivasubramanian, G. Pierre, M. Steen e G. Alonso. "Analysis of Caching and Replication Strategies for Web Applications". Em: *IEEE Internet Computing* 11.1 (2007), pp. 60–66. ISSN: 1089-7801. DOI: 10.1109/MIC.2007.3.

[Tea13a] I. Team. *Infinispan 6.0 Documentation*. Ago. de 2013. URL: https://docs.jboss.org/author/display/ISPN/Home.

[Tea13b] T. E. Team. *EhCache 2.7 Documentation*. Ago. de 2013. URL: http://ehcache.org/documentation.

[The13] T. J. V. D. T. The Hibernate Team. *HIBERNATE - Relational Persistence for Idiomatic Java*. Ago. de 2013. URL: http://docs.jboss.org/hibernate/orm/4.2/manual/en-US/html/index.html.

[VBZ08] I. Voras, D. Basch e M. Zagar. "A high performance memory database for web application caches". Em: *Electrotechnical Conference, 2008. MELECON 2008. The 14th IEEE Mediterranean*. Mai. de 2008, pp. 163–168. DOI: 10.1109/MELCON.2008.4618428.

[Zyl10] P. V. Zyl. "Performance Investigation into selected object persistence stores". Tese de mestrado. África do Sul: University of Pretoria, 2010.

[Zyl+09] P. van Zyl, D. G. Kourie, L. Coetzee e A. Boake. "The influence of optimisations on the performance of an object relational mapping tool". Em: *Proceedings of the 2009 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*. SAICSIT '09. Vanderbijlpark, Emfuleni, South Africa: ACM, 2009, pp. 150–159. ISBN: 978-1-60558-643-4. DOI: 10.1145/1632149.1632169.