IME - Instituto de
Matemática e Estatística

USP - Universidade
de São Paulo

Avaliação da ferramenta Hibernate em um ambiente de produção de larga escala

Rogério Manente

Orientador: Prof. Dr. João Eduardo Ferreira

Universidade de São Paulo
Instituto de Matemática e Estatística
Departamento de Computação

São Paulo – 2007

Apresentação

Este texto é minha monografia final da disciplina MAC0499 – Trabalho de Formatura Supervisionado do curso de Bacharelado em Ciência da Computação do Instituto de Matemática, Estatística e Computação, da Universidade de São Paulo. O projeto é a análise da ferramenta **Hibernate** como utilitário para persistência de dados de uma aplicação real.

Neste texto é descrita a aplicação com a qual a comparação foi feita, os métodos para análise assim como seus resultados e as conclusões finais. Também é descrita minha experiência com este projeto assim como com o curso de Ciência da Computação.

Sumário

I – Parte Técnica

Introdução

1.1 Persistência de dados

No desenvolvimento de sistemas com grande volume de informação um problema recorrente é a persistência dos dados. As principais questões envolvidas são o desempenho, o tempo/custo de desenvolvimento da solução e a facilidade de manutenção da aplicação. Normalmente, a necessidade de grande desempenho implica em problemas nas outras duas características desejáveis, ou seja, acarreta num extenso tempo/custo de desenvolvimento (em geral, gastos com tarefas simples e repetitivas, o que ainda gera grande margem para erros) e em código confuso e mal estruturado, que resulta em grande dificuldade na manutenção.

A solução tradicional para persistência de dados em aplicações orientadas a objetos e que utilizam o modelo de *Banco de Dados Relacional* para o gerenciamento dos dados (a imensa maioria dos sistemas desenvolvidos atualmente) é o acesso direto ao Gerenciador de Banco de Dados. No entanto, aplicações desenvolvidas com este paradigma resultam em código confuso e de muito difícil manutenção.

Por outro lado, uma das soluções para diminuição do tempo/custo de desenvolvimento e para criação de código estruturado e de fácil manutenção é o *Mapeamento Objeto/Relacional*. A técnica consiste na criação de uma camada na aplicação que mapeia objetos em tuplas no banco de dados relacional e torna, desta forma, transparente a persistência dos objetos. Argumenta-se que a utilização desta camada adicional compromete o desempenho da aplicação tornando essa solução marginal em ambientes nos quais desempenho é essencial.

A ferramenta mais utilizada para o *Mapeamento Objeto/Relacional* é a ferramenta da qual este texto trata - o **Hibernate**.

1.2 Objetivos

É muito difícil para arquitetos de software decidirem entre as diversas opções de persistência de dados sem referências com análises das soluções em ambientes reais nos quais as soluções já foram aplicadas e testadas.

Já vi diversos profissionais decidirem usar o **Hibernate** simplesmente pela comodidade ou por acreditarem que a perda de performance não é significativa. Assim

como outros optam pelo acesso direto ao banco de dados usando *stored procedures* por assumirem que desta forma o desempenho é muito superior.

O principal objetivo deste trabalho é fazer uma análise dos limites do **Hibernate** num ambiente real de larga escala. Pretendo mostrar uma comparação sólida e quantitativa entre a persistência usando o **Hibernate** e o acesso direto ao banco utilizando *stored procedures* armazenadas e pré compiladas.

Os principais critérios de comparação foram desempenho, tempo de desenvolvimento e facilidade de manutenção da aplicação.

Pretendo que esse texto seja uma referência útil para arquitetos de software que tem dúvidas na solução do problema da persistência e consideram o **Hibernate** como uma opção. Uma referência que ter-me-ia sido muito útil quando projetava uma aplicação com grande demanda por desempenho.

Pretendi escrever, além de um texto acadêmico e técnico, uma descrição de uma experiência pessoal. Além dos dados quantitativos contribuí com minha experiência no desenvolvimento da aplicação assim como nos testes e comparações realizados.

1.3 Hibernate – *Stored Procedures*

A persistência através de *stored procedures* é o paradigma 'tradicional' de persistência. Historicamente as grandes aplicações foram evoluindo com uma integração muito forte com o banco de dados e, portanto, faziam acesso direto a ele. Para tirar o maior proveito dos grandes avanços dos gerenciadores de banco de dados, a melhor forma de acesso é por meio de *stored procedures*, armazenadas e pré-compiladas.

Outro paradigma muito comum atualmente é o uso de utilitários que fazem o *Mapeamento Objeto/Relacional*. Neste caso procura-se remover de uma aplicação orientada a objetos a solução da persistência. Utiliza-se uma ferramenta capaz de armazenar objetos da aplicação em um gerenciador relacional. Apesar de existirem centenas de soluções desse tipo, a mais utilizada é o **Hibernate**, para Java - ou a versão para Microsoft .Net – **NHibernate**.

1.4 A aplicação de teste

A aplicação na qual os testes foram feitos integra um sistema de *roteamento de ordens* desenvolvido para uma grande corretora de valores. No Brasil, todo envio de ordem às bolsas de valores (**Bovespa e BM&F**) é feito por meio de uma corretora de valores. Para clientes não profissionais, geralmente as ofertas são enviadas por meio dos sites de *Home Broker* das corretoras. Já clientes institucionais, muitas vezes, enviam ordens a diversas corretoras de valores espalhadas pelo mundo. A comunicação entre esses clientes e as bolsas de valores é feita através do *roteamento de ordens* por corretoras. Ou seja, a corretora recebe mensagens desses clientes e as transmite às bolsas e vice-versa.

Evidentemente, o sistema responsável pelo *roteamento de ordens* deve ser muito **robusto e estável**, já que qualquer erro pode acarretar em perdas financeiras muito substanciais.

Diversos ativos negociados na Bovespa (como, por exemplo, ações da Companhia Vale do Rio Doce, Petrobras, ou Banco Itaú) são negociados, também, nas bolsas norte americanas, na forma de ADRs – American Depositary Receipts[1]. Embora sejam negociados de diferentes formas, estes papéis representam ativos de uma mesma empresa, portanto seus preços seguem a mesma trajetória (quando o preço da ação negociada no Brasil varia, a preço da ADR varia de acordo). O fato do desses papéis serem negociados de forma independente mas seus preços serem fortemente atrelados gera possibilidade de investidores operarem ativos de uma empresa no Brasil e nos EUA em busca de uma taxa pré-calculada de retorno. Esta prática é conhecida como *arbitragem*. Por exemplo, um investidor pode comprar uma ação do Banco Itaú no Brasil e vender uma ADR do mesmo banco na bolsa americana, se os preços dos dois papéis estiverem "descolados" e esta operação lhe render uma taxa atraente. [2]

Como existem diversos investidores institucionais realizando esta prática – ou seja, colocando ofertas muitas vezes idênticas simultaneamente – os investidores que obtiverem a comunicação com as bolsas em menor tempo conseguirão os melhores negócios ou as melhores taxas. Logo, um bom sistema de *roteamento de ordens* tem uma demanda muito grande por **desempenho**. [3]

Na corretora de valores para a qual foi desenvolvida a solução que descrevo nesse texto havia uma necessidade de uma solução eficiente de *roteamento de ordens*. No entanto, havia uma urgência muito grande - como geralmente acontece em empresas, quase todas os projetos já nascem atrasados! O projeto foi muito bem sucedido. Um dos resultados mais significativos é que atualmente aproximadamente 20% das ordens enviadas à Bovespa passa por esse sistema. Esse número chegou a ser maior do que 30%.

[1] Maiores detalhes sobre a negociação na forma de ADRs podem ser encontradas neste artigo da Wikipédia [9]

[2] Existem diversos tipos de *arbitragem*, maiores informações podem ser vistas nesta outra página da Wikipédia [8]

[3] Na prática, além de ter o menor tempo possível no envio de ordens às bolsas, o investidor também deve procurar ter as *cotações* dos preços dos ativos o mais próximo do tempo real possível.

Ambiente

2.1 Arquitetura geral

A arquitetura da aplicação para a solução do problema de *roteamento de ordens* foi feita da seguinte forma. O problema foi dividido em três partes: Comunicação com os clientes, comunicação com as bolsas, tratamento da lógica e regras de negócio[4].

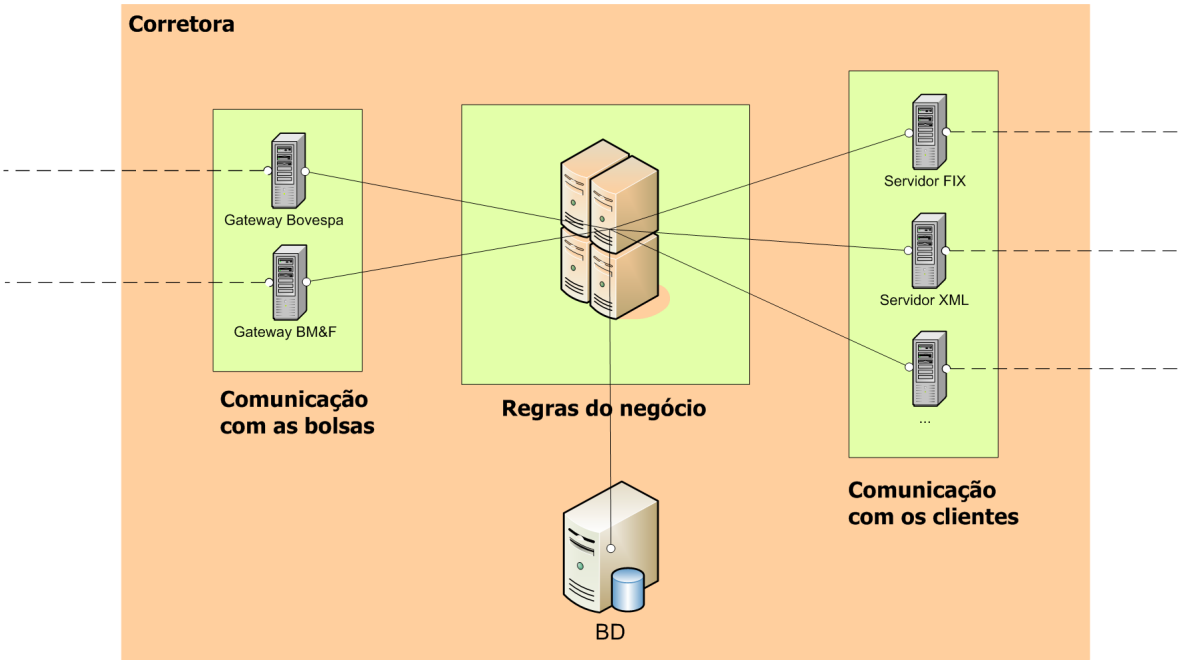

Figura 2.1. Arquitetura geral do sistema

A comunicação com os clientes é feita de diversas formas. Os tipos mais comuns são a comunicação por meio de um protocolo definido entre o cliente e a corretora (normalmente baseados em XML) ou por meio do protocolo FIX[5]. O protocolo FIX é uma especificação de domínio público desenvolvida para resolver os problemas de comunicação entre entidades participantes do mercado financeiro. Evidentemente é muito desejável que corretoras, clientes e bolsas consigam se comunicar através de um protocolo internacional.

A comunicação com as bolsas de valores (no nosso caso a Bovespa e a BM&F) é feita por meio dos protocolos e soluções criadas por cada uma delas. No caso da Bovespa é utilizado um componente que disponibiliza métodos e eventos usados para esta comunicação. Este servidor responsável pela interface com uma bolsa recebeu o nome de *Gateway*.

[4] Na verdade existem mais partes do sistema que foram omitidas. Foram mostrados apenas os módulos que interessam ao assunto deste texto. No sistema real existem outros módulos, como um sistema responsável pela integração com o Sinacor, sistema de gestão das corretoras da Bovespa.

[5] Mais informações sobre o protocolo podem ser encontradas no site da organização do protocolo FIX [2]

Portanto, foram criados servidores para comunicação com os clientes e servidores para comunicação com as bolsas. Mas era necessário um servidor central responsável por unir todos estes servidores e aplicar todas as regras e verificações (como controle de saldo, exposição financeira etc.) Na minha solução, este servidor recebeu o nome de *Engine*.

Logo uma mensagem comum faz o seguinte percurso. É recebida pelo servidor FIX, enviada ao *Engine* – que faz todas as verificações necessárias – transmitida ao *Gateway* que trata do envio da mensagem à bolsa. As mensagens recebidas pela bolsa fazem exatamente o caminho inverso.

Deste modo foi possível isolar cada componente do sistema de modo que qualquer alteração por parte dos clientes, das bolsas ou regras de negócio pudesse ser tratada realizando ajustes apenas no componente a que a mudança diz respeito. Além disso foi possível obter um excelente desempenho já que o sistema pôde ser distribuído fisicamente em pelo menos 3 servidores. E, no caso deste texto provavelmente o mais importante, o problema da persistência foi isolado num único componente – o *Engine*.

2.2 *Engine*

A parte do sistema que nos interessa, portanto, é o *Engine*. Ele foi desenvolvido na linguagem C#, da solução .NET, da Microsoft. Logo, a versão do **Hibernate** utilizado é o chamado **NHibernate**[6]. O gerenciador de banco de dados utilizado foi o Microsoft SQL Server, versão 2000.

O *Engine* foi criado dividindo o sistema em 4 partes.

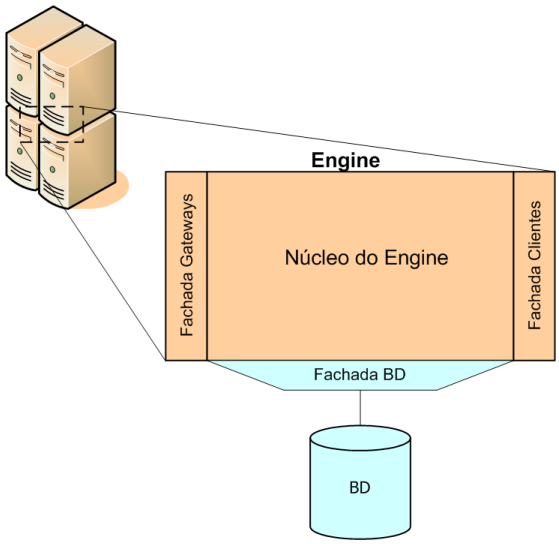

Figura 2.2. Arquitetura do *Engine*

[6] O **NHibernate** começou como uma adaptação independente do **Hibernate** para o .NET. Foi posteriormente incorporado ao projeto **Hibernate** e hoje é mantido pela Jboss. As versões do **NHibernate** procuram seguir uma trajetória semelhante ao produto criado para Java, embora sigam linhas independentes. Mais informações do **NHibernate** podem ser encontradas no site da organização [4]

Assim como houve um isolamento dos componentes na arquitetura do sistema, dentro do *Engine* foi criado um isolamento semelhante. Foi criada uma fachada responsável por fazer a integração dos *Gateways* que fazem a comunicação com as bolsas. Esta fachada isola os *Gateways* de forma que qualquer alteração nestes servidores ou no protocolo de comunicação entre eles e o *Engine* acarreta em alterações apenas nesta fachada. Da mesma forma foi criada uma fachada que garante a integração dos servidores que fazem a comunicação com os clientes (ou seja, os servidores FIX, XML ou qualquer outro).

Portanto toda a lógica do sistema ficou encapsulada no núcleo do *Engine*. Com isso, conseguimos manter o isolamento que nos garante que possíveis mudanças causem alterações em pequenas partes do sistema, ou seja, diminuímos o acoplamento[7].

Com isso, a única parte do sistema que necessita de persistência de dados é o núcleo desta aplicação. Quando a aplicação foi desenhada, a camada que separa o núcleo do *Engine* do próprio banco de dados utilizada foi o **Hibernate**.

A escolha do **Hibernate** não foi fácil. Havia duas questões muito importantes: a demanda por desempenho quando a aplicação ficasse pronta e a necessidade da solução ficar pronta em tempo extremamente curto. Dentre as diversas opções de persistência disponíveis, por diversos motivos (entre eles robustez e credibilidade das soluções), a dúvida ficou entre fazer a persistência diretamente utilizando s*tored procedures*, utilizado **Hibernate** ou utilizar alguma das diversas outras ferramentas de persistência para .NET[8].

Como não tinha relatos de experiências em larga escala utilizando o **Hibernate** foi difícil saber exatamente qual a perda de performance que a aplicação teria e ainda qual o ganho de tempo no desenvolvimento da aplicação. Levantei várias referências bibliográficas, entretanto, nenhuma delas mostrava resultados muito claros. As referências mais importantes que usei foram conversas com amigos que utilizavam intensamente o **Hibernate** em seus trabalhos e podiam ajudar na decisão.

Como a urgência era imensa decidi utilizar o **Hibernate**, mesmo sem estar completamente convencido de que esta seria a melhor solução. Para resolver este problema criei uma camada extra na aplicação. Esta camada seria uma interface entre o núcleo do *Engine* e o **Hibernate**. A idéia era poder fazer uma substituição na solução de persistência caso o resultado final não fosse bom.

[7] Redução do acoplamento é um conceito essencial na programação orientada a objetos. A mesma idéia está sendo utilizada aqui para o isolamento de partes do sistema, não apenas para o isolamento das classes da aplicação.

[8] Existem muitas ferramentas de persistência para .NET. Dentre as ferramentas para mapeamento Objeto/Relacional (ORM) algumas conhecidas que me lembro agora são Gentle.NET, Wilson OR Mapper e Npersist. Na época do desenvolvimento desta solução a Microsoft também pretendia produzir uma solução para persistência de dados. Chamava-se ObjectSpaces mas acredito que o projeto foi abandonado.

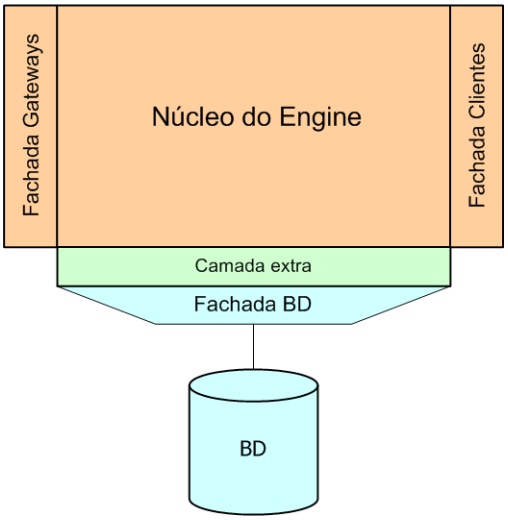

Figura 2.3. Camada adicional de persistência

Criei uma interface do C# que qualquer classe criada para ser uma solução da persistência deveria implementar. O núcleo do *Engine* referencia apenas esta interface, de modo que uma substituição na classe instanciada torna-se bastante transparente para o resto da aplicação. Esta interface foi definida mais ou menos da seguinte forma.

```
interface IPersistencia
{
    Conta getConta(long idConta);
    Oferta getOferta(long idOferta);
    void saveOferta(Oferta oferta);
    /* (...) */
}
```

Para fazer a persistência por **Hibernate** foi criada uma classe que implementa `IPersistencia`. A criação de sessões e configurações do **Hibernate** ficam encapsuladas nesta classe, de forma transparente ao resto da aplicação. A classe tem mais ou menos a seguinte estrutura.

```
class PersistencianHibernate : IPersistencia
{
    Conta getConta(long idConta);
    {
      /* (...) */

      Conta conta = (Conta)session.Load(typeof(Engine.Conta),
                idConta);

      /* (...) */
      return conta;
    }

    /* (...) */

    void saveOferta(Oferta oferta)
    {
      /* (...) */
      session.Save(oferta)[9];
      /* (...) */
    }

    /* (...) */
}
```

Esta solução separa qualquer solução de persistência do resto do *Engine*. Mas, como se pode ver, a interface de persistência (`IPersistencia`) foi criada fortemente baseada na lógica do **Hibernate**. Portanto, como explicarei nos resultados mais adiante, fica muito difícil retirar da aplicação o paradigma de persistência do **Hibernate**.

O resultado da aplicação terminou sendo excelente. O desempenho do sistema ficou muito bom e o **Hibernate** não foi um gargalo muito sério. Alguns ajustes foram feitos depois do sistema entrar em um ambiente de produção mas a solução inicial ficou muito próxima da solução final. Os testes dos quais escreverei adiante foram feitos para saber exatamente qual o impacto do **Hibernate** no resultado final e ainda se é vantajosa a troca por outra solução.

[9] Na realidade, o **Hibernate** faz uma distinção entre *save* e *update*. Para simplificar usarei apenas a notação de *save*.

Análise

3.1 Ambiente para testes

Para que a análise do sistema pudesse ser feita de forma contundente, foi criado um ambiente completo e independente o mais próximo do ambiente de produção. O sistema foi mantido com a arquitetura inicial. Foi criado um servidor responsável por emular a bolsa da forma mais real possível. Ou seja, responder com as mensagens exatamente como faz a bolsa e ainda com a mesma proporção em que as coisas acontecem no ambiente real. Isto é, usar a mesma proporção em que os eventos acontecem no ambiente de produção. Estes eventos disparados pela Bovespa são basicamente "*order response*", "*error response*", "*order cancel*", "*cancel reject*" e "*trade response*", ou seja, mensagens de recebimento de oferta, de ocorrência de erro, de oferta cancelada, ,de cancelamento rejeitado e de execução de negócio, respectivamente.

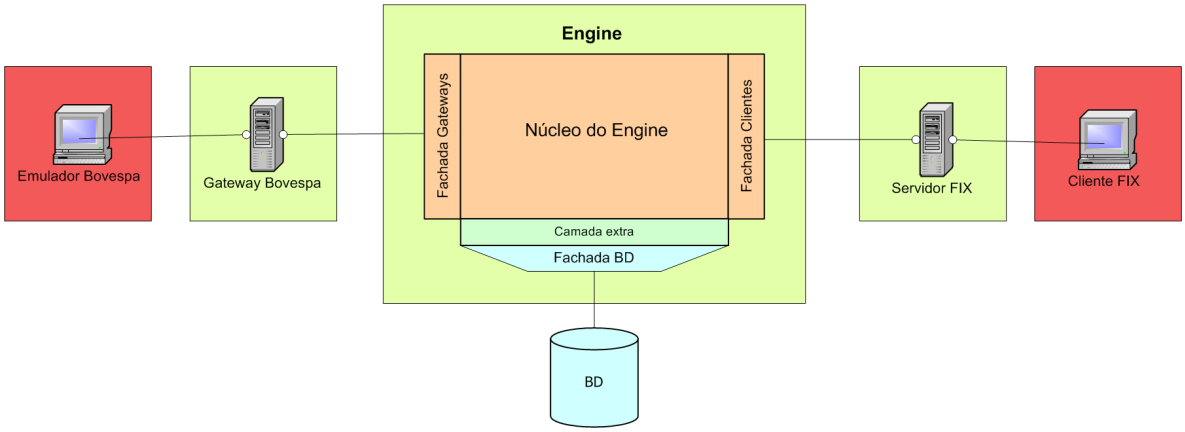

Figura 3.1. Ambiente de testes

Foi criado, também, um cliente da conexão FIX. Este cliente envia as mensagens do mesmo modo como fazem os clientes da corretora – enviando muitas mensagens de novas ofertas e cancelamentos.

Desta forma já era possível emular o ambiente de produção com ótimo grau de precisão.

3.2 Coleta dos dados

Para fazer a coleta dos dados foi utilizado o seguinte método. Cada função da classe `PersistenciaHibernate` foi modificado de forma que o instante do relógio fosse marcado no início do método e marcado novamente no fim do método. Além disso, no final de cada função, era gravado um *LOG* com a diferença dos tempos, indicando quanto tempo

decorreu na execução do método. Os métodos todos foram transformados para terem uma estrutura semelhante à seguinte.

```
class PersistencianHibernate : IPersistencia
{
    Conta getConta(long idConta);
    {
      DateTime inicio = DateTime.Now;
      /* (...) */

      Conta conta = (Conta)session.Load(typeof(Engine.Conta),
                idConta);

      /* (...) */
      DateTime fim = DateTime.Now;
      long ticks = ((TimeSpan)fim.Subtract(inicio)).Ticks;
      saveLOG();
      return conta;
    }

    /* (...) */

    void saveOferta(Oferta oferta)
    {
      DateTime inicio = DateTime.Now;
      /* (...) */
      session.Save(oferta);
      /* (...) */
      DateTime fim = DateTime.Now;
      long ticks = ((TimeSpan)fim.Subtract(inicio)).Ticks;
      saveLOG();
    }

    /* (...) */
}
```

Além disso, os principais métodos do *Engine* também foram adaptados de forma semelhante, como por exemplo, o seguinte método.

```
void enviarOfertaParaBolsa(Oferta oferta)
{
      DateTime inicio = DateTime.Now;

      /* (...) */

      DateTime fim = DateTime.Now;
      long ticks = ((TimeSpan)fim.Subtract(inicio)).Ticks;
      saveLOG();
}
```

Com essas alterações foi possível criar estatísticas e concluir quais métodos causavam a maior degradação no desempenho geral do sistema.

A estratégia para a comparação dos paradigmas **Hibernate** e acesso direto ao banco de dados foi dividida em duas etapas. A primeira foi fazer a substituição dos métodos **da**

camada de persistência que causavam mais impactos por métodos equivalentes que fizessem o acesso direto ao banco de dados por meio de *stored procedures*. A segunda etapa foi analisar os métodos mais demorados do *Engine* tentando migrar o processamento do servidor de aplicação para o banco de dados. Sem dúvidas é nesta mudança que as diferenças mais significativas podem aparecer. Isto porque é nesta mudança que se pode tirar o maior proveito da funcionalidade das *stored procedures*, que já ficam armazenadas e pré-compiladas no banco de dados e utilizam o poder dos muito aprimorados algoritmos que os gerenciadores de banco de dados relacionais possuem atualmente.

Entretanto, existem alguns grandes problemas na mudança de parte da lógica do sistema para o banco de dados. Muitos arquitetos de software adeptos da programação orientada a objetos defendem que esta estratégia é contra alguns dos principais conceitos da orientação a objetos. Eu acredito que é possível desenvolver uma solução fortemente orientada a objetos mesmo delegando parte da lógica dos sistema ao gerenciador de banco de dados. Isto pode ser feito com o padrão *Strategy*[10]. Outro importante problema no escopo deste trabalho é a grande dificuldade em mensurar o ganho de desempenho fazendo mudanças como as citadas anteriormente. É simples mensurar a diferença de desempenho num caso específico, mas é muito difícil generalizar a idéia e estabelecer uma métrica.

3.3 Resultados

Como já era esperado, a análise dos tempos coletados nos métodos da camada de persistência indicou que poucos métodos eram acessados **muito** frequentemente e eram responsáveis por grande parte do tempo gasto na persistência dos dados. O resultado não surpreende, já que poucas atividades se repetem muito frequentemente e as demais são realizadas muito raramente, em relação às execuções totais.

Pude perceber que apenas quatro métodos eram responsáveis por 84,96% de todo o tempo gasto na camada de persistência. Além disso, de todas as execuções de métodos que ocorreram na camada de persistência, 56,56% foram execuções destes quatro. Isto indica duas coisas. A primeira é que estes quatro métodos sofrem mais execuções do que a soma de todos os demais. A segunda é que nitidamente quase todo o tempo gasto com persistência é gasto com estas quatro funções.

[10] Mais informações sobre padrões de programação orientada a objetos podem ser encontradas nesta página da Wikipédia [7]

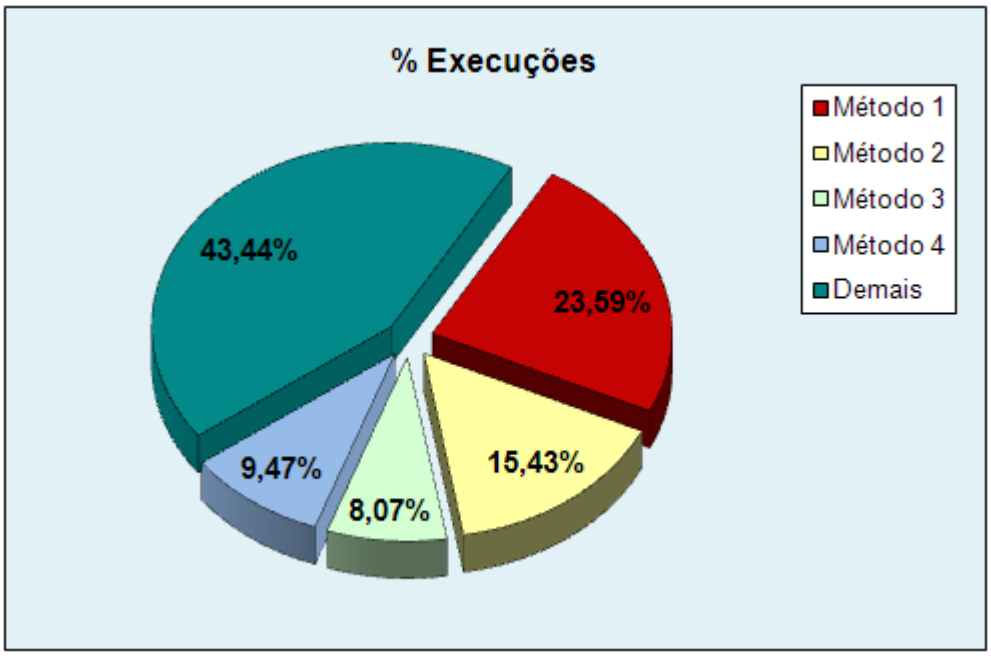

Figura 3.2. Execuções na camada de persistência

O gráfico da figura 3.2 mostra a proporção das chamadas de função entre todas as funções da camada de persistência. Dos 48 métodos existentes na camada de persistência[11], quatro deles são responsáveis por mais da metade das execuções. O da figura 3.3 mostra a divisão do tempo gasto na camada de persistência. A concentração é ainda mais clara. Enquanto 44 métodos são responsáveis por aproximadamente 15% do tempo, os outros 4 acumulam quase 85% de tempo de execução.

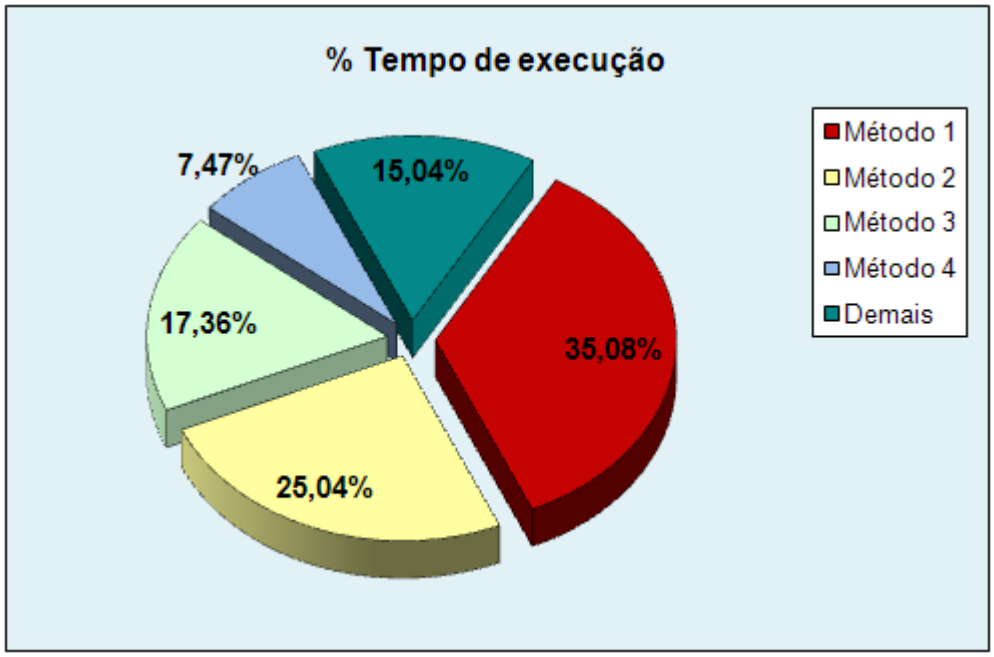

Figura 3.3. Tempo na camada de persistência

[11] Os dados mencionados referem-se à versão da aplicação utilizada para os testes e análises. No ambiente real a aplicação evoluiu e possui, por exemplo, mais do que os 48 métodos mencionados.

O gráfico da figura 3.4 mostra a comparação lado a lado dos dados mencionados acima.

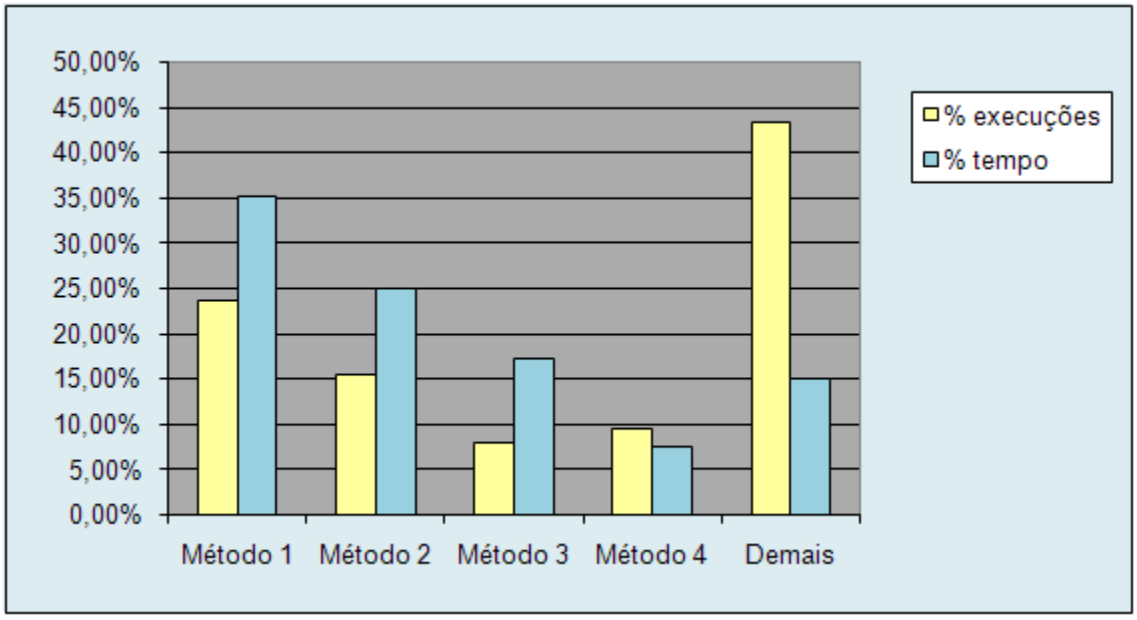

Figura 3.4. Execuções/tempo na camada de persistência

A análise dos dados obtidos torna duas coisas bastante claras. A primeira é que os esforços em mudanças na camada de persistência deve se concentrar em poucos métodos – apenas quatro deles. A segunda, uma conclusão já bastante interessante, é que, *a priori*, não compensa gastar muito tempo e dedicação com a otimização dos outros 44 métodos. Para tornar esse quadro mais claro, ainda que seja possível uma otimização que reduza os tempos de 44 métodos pela metade, o tempo total gasto em persistência é reduzido em apenas 7,52%. Contudo, se o tempo dos 4 métodos mais custosos for reduzido em 15%, o mesmo tempo total é reduzido em 12,74%[12].

Como não se espera que uma redução de 50% no tempo gasto com a persistência de algumas entidades seja simples de se obter, dificilmente é justificado um esforço direcionado a todas as funções da camada de persistência. Na minha aplicação resolvi realizar alterações apenas nos 4 métodos mais custosos. Se os resultados obtidos com as mudanças fossem alarmantes, talvez uma atenção aos demais métodos fosse justificada.

A criação de uma nova implementação da camada de persistência com a alteração dos 4 métodos mais custosos para acesso direto ao gerenciador através de *stored procedures* se mostrou muito mais difícil do que previsto. Cada um dos métodos era responsável pela recuperação ou armazenamento de uma entidade no banco de dados. No entanto, cada entidade está vinculada a outras entidades direta ou indiretamente. Ou seja, a mudança de quatro métodos resultou na criação de 21 métodos. A criação desses métodos se mostrou uma tarefa **muito** repetitiva, cansativa e com grande margem para erros. Isso era esperado, mas numa intensidade muito menor.

O resultado da troca destes métodos foi um ganho de desempenho de 13,85%.

[12] É importante lembrar que este **não** é o tempo total gasto pela aplicação. Trata-se apenas do tempo total gasto na camada de persistência criada com o **Hibernate**.

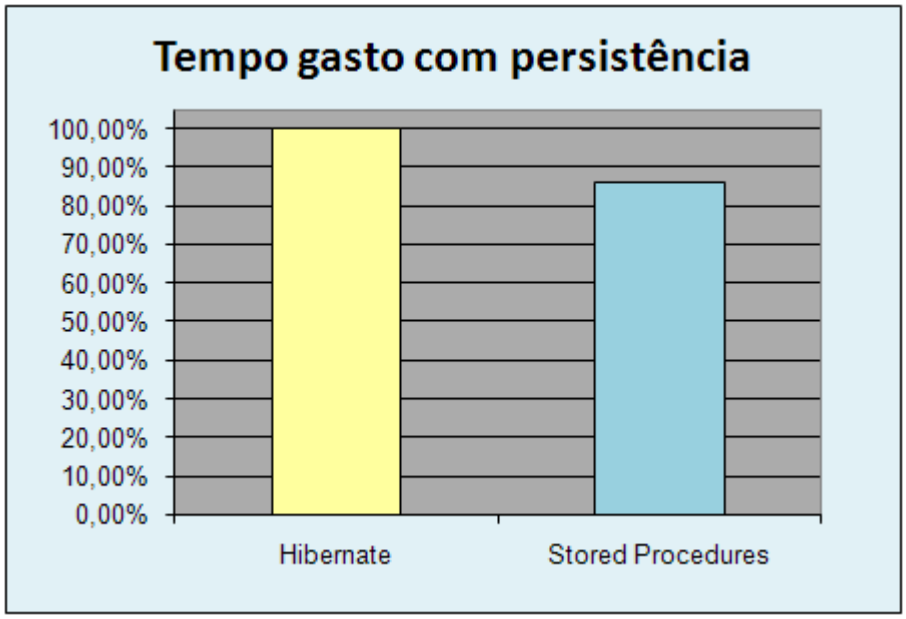

Figura 3.5. Ganhos na camada de persistência

O ganho de desempenho em cada método foi bastante semelhante. A mudança do método 1 – o método mais custoso – rendeu um ganho de 17,48%. O método 2 – 13.58%. O método 3 foi o de maior ganho – 20,76. O método 4, o de menor ganho - 10,75%. Os quatro métodos continuaram representando a maior parte do tempo gasto com persistência. A proporção do tempo gasto com eles passou de 84,96% para 82,68%. Uma mudança bastante pequena, o que parece indicar que, de fato, não é válida uma atenção aos outros métodos que não os quatro mais custosos. A mudança na proporção dos tempos gastos com os métodos é mostrada no gráfico da figura 3.6.

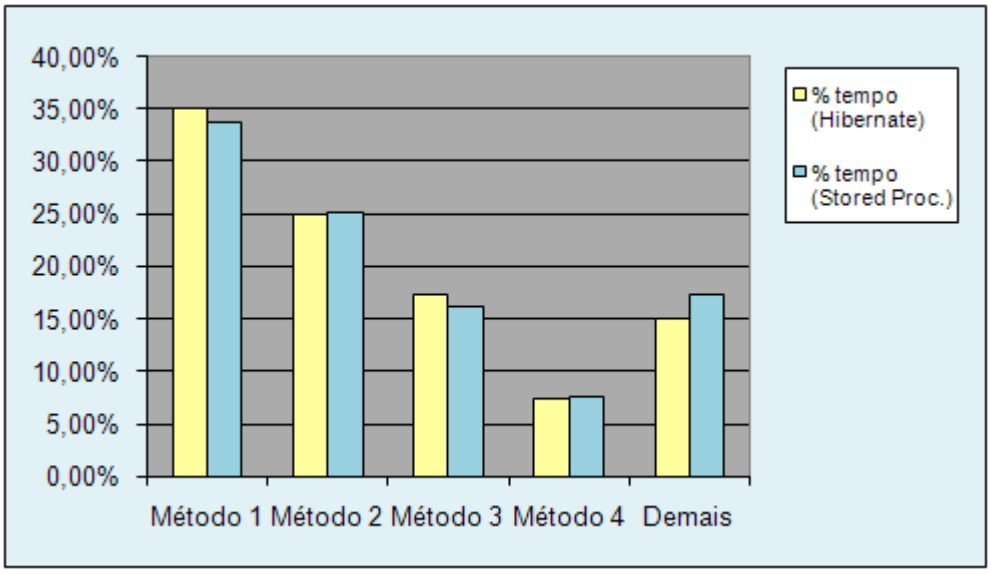

Figura 3.6. Proporção de tempo na camada de persistência

A análise dos métodos do núcleo do *Engine* apresentou resultados um pouco diferentes. Havia uma dispersão um pouco maior nos tempos das funções. Enquanto na camada de persistência quatro métodos eram responsáveis por quase 85% do tempo, no núcleo os 5 métodos mais custosos representavam pouco mais de 75% de todo o tempo gasto. Os demais métodos eram incrivelmente equivalentes entre si no tempo gasto para a execução.

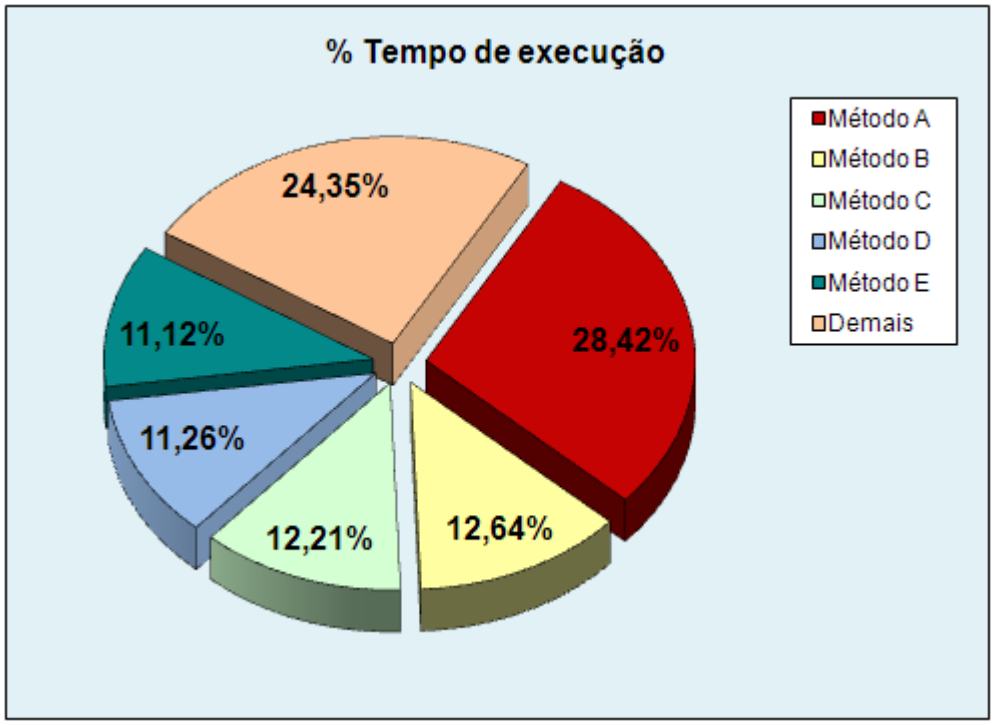

Figura 3.7. Tempo no núcleo do *Engine*

Todavia, ainda pode-se perceber uma concentração muito grande em poucos métodos. Evidentemente cada um destes métodos faz chamadas a diversos outros. A idéia aqui era substituir cada um destes métodos por outro equivalente que realizasse poucas (possivelmente apenas uma) chamadas a *stored procedures*. Para não perder a coerência com a arquitetura inicial, a interface `IPersistencia` foi adaptada de forma a incluir métodos que faziam chamadas às *procedures*. Ou seja, foram criados métodos da seguinte forma.

```
interface IPersistencia
{
    /* (...) */
    bool aprovaOperacao(Oferta oferta);
    /* (...) */
}
```

Portanto criei *stored procedures* no banco de dados de forma que os métodos ficassem muito simples. Neste ponto o problema de ganho de desempenho passa a depender da qualidade das consultas *SQL* criadas nas *procedures*. Para que o ganho de desempenho fosse bom, os planos de execução das *procedures* deveriam ser os melhores possíveis.

Portanto o problema passa a depender da habilidade do desenvolvedor (ou preferencialmente de um DBA) em criar boas consultas *SQL*. Como tenho bastante experiência na criação e otimização ("*tuning*") de consultas *SQL*, tentei aprimorar as *procedures* o máximo possível. No entanto passa a ser extremamente difícil mensurar a diferença de desempenho entre métodos baseados no **Hibernate** e métodos que delegam o processamento a *stored procedures* já que essa modificação depende muito da aplicação e do desenvolvedor. Essa faz parte das conclusões sobre as quais comentarei mais adiante.

Como era esperado, com as mudanças para *procedures*, foi possível obter um ganho excelente em alguns dos métodos. No entanto, a surpresa foi que, em dois deles, o tempo gasto aumentou. Ou seja, o desempenho piorou. Fiz diversas tentativas de alterar as *stored procedures* que estes dois métodos referenciavam de modo a melhorar a performance. Mas ainda assim os resultados foram muito ruins. Possivelmente algum profissional com mais habilidade com consultas *SQL* teria conseguido resultados melhores, mas acredito que ainda seriam piores do que os métodos iniciais.

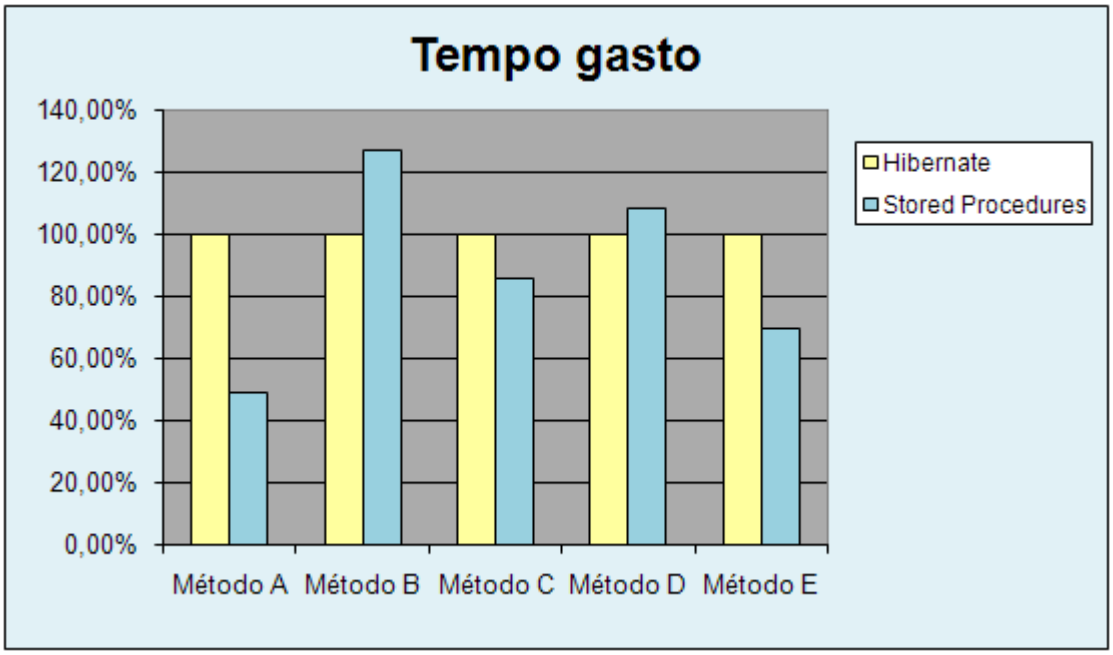

Figura 3.8. Alteração nos tempos no núcleo do *Engine*

O maior ganho (um resultado impressionante) foi no método A – um ganho de 51,03%. Os métodos C e E tiveram ganhos de 14,02% e 30,53%, respectivamente. As perdas ocorreram nos métodos B e D. Mesmo após diversas tentativas, o melhor resultado que consegui foi uma grande perda de 27,32% no método B e de 8,34 no método D. Acredito que um dos maiores motivos para que não fosse possível melhorar o desempenho nestes dois métodos seja a criação inicial da aplicação com o paradigma do **Hibernate**. Em outras palavras, a aplicação "não foi desenhada" para tirar maiores proveitos das *stored procedures*.

Com as mudanças de todos os métodos, o tempo total gasto no núcleo do *Engine* diminuiu em 15,22%. E voltando os métodos que sofreram perdas aos originais, o ganho de desempenho foi de 19,61%[13].

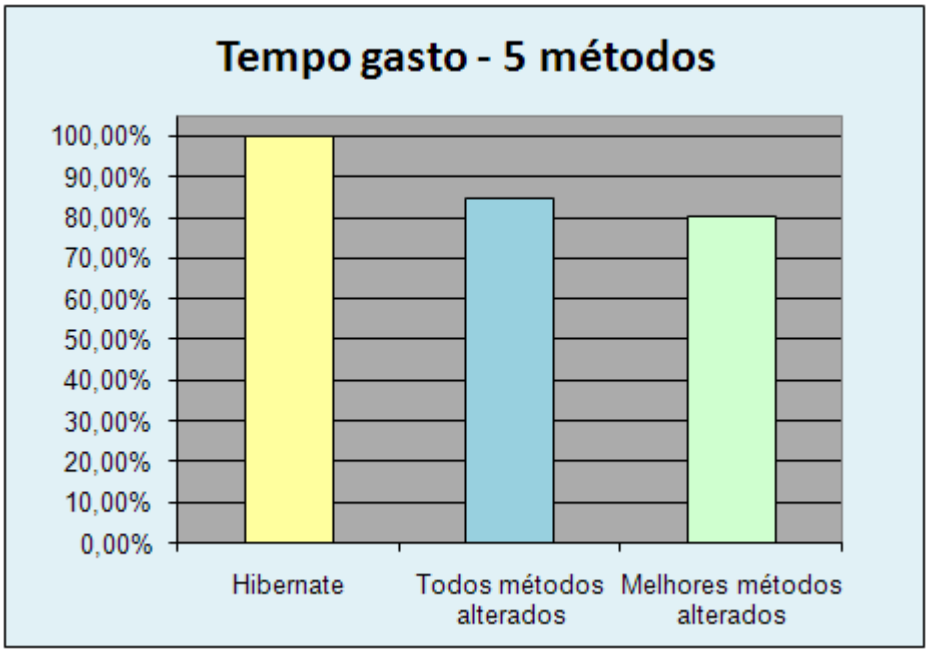

Figura 3.9. Ganhos no núcleo do *Engine*

Desta forma, a participação de cada um destes cinco métodos no tempo total gasto no *Engine* mudou bastante. A maior diferença foi no método A. Enquanto, antes da mudança, ele era responsável por 28,42% do tempo total, depois da alteração ele passou a ser responsável por 16,41%. Sem dúvida foi com essa alteração que obtive a maior parte de todo o ganho de desempenho. Os métodos C e E, apesar de terem sofrido uma melhora significativa, tiveram sua participação mantida razoavelmente estável. Isso porque a grande diminuição do método mais custoso elevou a participação de todos os outros. A participação do método C mudou de 12,21% para 12,38% e a do método E de 11,12% para 9,11%. Os métodos B e D tiveram suas participações aumentadas de 12,64% para 18,98% e 11,26% para 14,39%, respectivamente. Os outros métodos todos somados saltaram de 24,35% para 28,72%.

[13] Vale lembrar que os testes de mudança dos métodos do *Engine* foram feitos **mantendo** as alterações na camada de persistência. Ou seja, comparando com o cenário inicial, o resultado final é ainda pouco melhor que 19,61%. No entanto a diferença é pequena já que com a criação das *stored procedures* as chamadas a métodos na camada de persistência diminuíram muito.

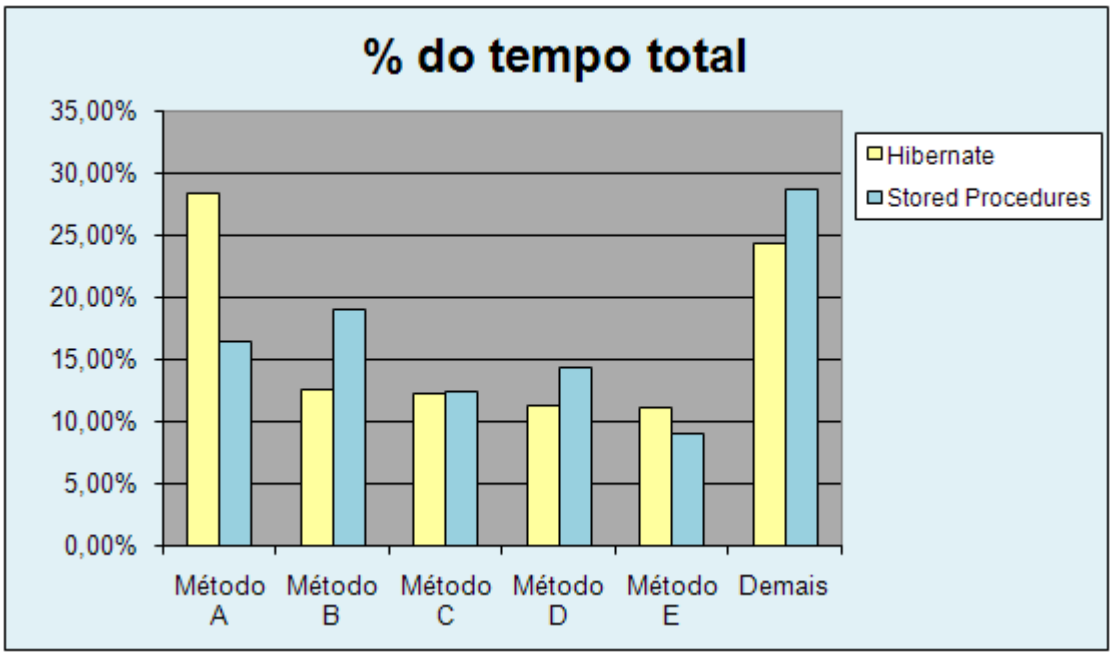

Figura 3.10. Proporção de tempo no núcleo do *Engine*

Conclusões

4.1 Desempenho

A primeira e, talvez, mais importante conclusão na realização deste trabalho é que é impossível estabelecer uma métrica capaz de comparar – em todos os sentidos – os ganhos de desempenho entre a persistência através de uma ferramenta como o **Hibernate** e a persistência direta por *stored procedures*. É possível medir o ganho de performance da substituição direta (como fiz na minha camada de persistência), mas essa medida não tira proveito das mais importantes funcionalidades existentes nas *stored procedures*.

Os ganhos na substituição direta foram, todavia, já bastante significativos. Dependendo da aplicação essa substituição direta já pode ser justificada. Mas, claramente, a maior possibilidade de ganhos de desempenho existe nos núcleos das aplicações, nos quais reside toda a lógica de negócio. No caso da aplicação descrita neste trabalho, obtive ganhos extraordinários. Porém, como pode-se perceber nos resultados, não é possível saber *a priori* o que esperar. Infelizmente a maior conclusão no quesito desempenho é o que já se podia esperar. Isto é, que ***é possível*** obter ganhos **muito** substanciais na substituição de uma solução baseada em **Hibernate** para uma solução baseada no acesso direto através de *stored procedures*. Entretanto, este trabalho mostrou que um caminho possível é implementar um sistema usando um arcabouço tal qual o **Hibernate** e em seguida fazer algumas substituições de métodos por chamadas utilizando *stored procedures*.

4.2 Implementação e manutenção

Outra grande constatação foi na diferença entre as soluções no que diz respeito à facilidade e no tempo de desenvolvimento para implementação. Evidentemente soluções como as de *Mapeamento Objeto/Relacional* exitem para facilitar na espinhenta tarefa da criação da persistência. Como são largamente utilizadas pode-se concluir que o auxilio que trazem é vantajoso e significativo. No entanto, na minha tarefa de replicar uma solução existente sem o auxílio da ferramenta, percebi o quão grande é a economia de tempo e esforço quando não se faz todo o trabalho sozinho. Em outras palavras, percebi a incrível facilidade e rapidez com que pode-se implementar a persistência numa aplicação utilizando uma ferramenta como o **Hibernate**. Aparentemente, em situações comuns, é mais vantajoso fazer uso de utilitários como este e obter maiores ganhos de desempenho de outras formas - como investimentos em equipamentos, ou otimizações em métodos pontuais críticos.

No que diz respeito à manutenção tirei conclusões semelhantes. Quase todos os projetos crescem ou precisam ser alterados de alguma forma. O tempo e esforço que se

gasta se a aplicação for desenvolvida utilizando-se de uma ferramenta como o **Hibernate** é, também, muito menor.

4.3 Conclusões finais

As aplicações mais comuns têm muita motivação para utilizar o **Hibernate**, ou outra ferramenta igualmente poderosa. Os ganhos de tempo e esforço compensam as perdas de desempenho. Em algumas aplicações que têm maior demanda por desempenho é justificada uma solução híbrida entre **Hibernate** e acesso direto por *stored procedures*. Neste caso pode-se tentar utilizar o **Hibernate** para quase toda a tarefa de persistência e *procedures* para remover a lógica da aplicação. Embora não se possa afirmar que existirá ganhos com este tipo de arquitetura, este trabalho mostrou que a possibilidade de ganhos muito significativos é imensa. Como é muito comum que poucas áreas dos sistemas sejam responsáveis por quase todo o tempo de processamento gasto, pode ser válida uma tentativa deste tipo com foco nas poucas partes mais custosas e críticas.

Outra conclusão importante obtida é que uma mudança na solução de persistência pode ser incrivelmente complicada. A arquitetura das aplicações é fortemente baseada no paradigma escolhido para persistência. Embora seja muito difícil tentar prever o futuro das aplicações, é válido um esforço inicial antes de escolher a solução de persistência a ser adotada.

Bibliografia

[1] Bovespa - http://www.bovespa.com.br

[2] The FIX Protocol Organization - http://www.fixprotocol.org

[3] Hibernate - http://www.hibernate.org

[4] NHibernate - http://www.nhibernate.org

[5] Microsoft SQL Server - http://msdn2.microsoft.com/sql

[6] Microsoft C# - http://msdn.microsoft.com/vcsharp

[7] Padrões de Orientação a Objetos (Wikipédia) - http://en.wikipedia.org/wiki/Design_pattern_(computer_science)

[8] Arbitragem (Wikipédia) - http://en.wikipedia.org/wiki/Arbitrage

[9] ADRs (Wikipédia) - http://en.wikipedia.org/wiki/American_Depositary_Receipt

II – Parte Subjetiva

Experiência Pessoal

1.1 Experiência no desenvolvimento num ambiente profissional

Quando li a respeito da parte subjetiva no roteiro sugerido para preparação das monografias, na página da disciplina, a coisa que achei mais importante de escrever sobre foi como é o desenvolvimento num ambiente profissional. Muitas vezes percebi no IME que os alunos gostam bastante de desenvolver EPs mas não têm muita idéia de como é o desenvolvimento na "vida real". Como sou um dos (poucos) alunos que trabalhou durante a maior parte do tempo em que cursei a faculdade (e, no meu caso, foi **bastante** tempo!**)**, achei uma boa idéia tentar mostrar um pouco este lado. Além disso achei uma boa idéia dar a minha opinião sobre o polêmico assunto *trabalho durante o BCC*.

A principal coisa de que me lembro do início do desenvolvimento deste projeto (e de quase todos) é que ele já nasceu atrasado. Na experiência que tenho dificilmente um projeto nasce com todas as datas bem definidas. Normalmente a data de entrega é o mais breve possível, senão "para ontem". Além disso, é muito comum que existam vários projetos simultâneos que têm urgência. E mais comum ainda é projetos serem interrompidos porque aparece outro ainda mais importante. Este projeto foi um destes em que tudo o mais teve que ser interrompido.

Outra grande diferença entre projetos reais e trabalhos no IME é que, geralmente, não existe uma definição clara nos projetos. Quando se é responsável por um projeto real normalmente o que se tem é o problema esperando por uma solução. Portanto, depois que o projeto é confirmado, um dia você chega ao trabalho e tem apenas um problema. Deve decidir por onde começar, como separar o projeto com os outros membros da equipe e todo o resto para que o problema seja resolvido no menor tempo possível e que a solução seja a melhor possível.

Mas, sem dúvida, a maior diferença entre todos os projetos desenvolvidos na faculdade e em empresas é que o que foi desenvolvido vai, de fato, ser utilizado. E além disso você vai estar lá quando os problemas ocorrerem ou quando houver necessidade de manutenção. E pior – você será responsável. Esses são nitidamente ótimos motivos para realizar o melhor trabalho possível! O problema disso é que o melhor trabalho possível sempre demora mais. Portanto parte do trabalho é achar a solução ideal entre o melhor possível e o que demora menos para ser desenvolvido. E, talvez a coisa mais importante que os alunos saindo da faculdade devam saber é que depois que o projeto estiver em produção e surgirem problemas, não vai resolver dizer "eu avisei!"

Sem sombra de dúvidas o maior motivo para eu ter demorado oito anos para conseguir sair da faculdade foi o trabalho durante o curso. Especificamente no IME, trabalhar durante o curso o torna **muito** mais difícil. Para mim, além de tornar muito mais

difícil cursar as disciplinas, me afastou da minha turma. O que torna ainda mais complicado o tempo na faculdade. Não só a formatura é atrasada em alguns (ou muitos) anos, o aproveitamento nas disciplinas é muito comprometido. No entanto, em muitos casos (como no meu) o trabalho durante a faculdade **não** é uma opção. Quando se tem que trabalhar durante o curso existem basicamente duas opções. Ou arruma-se alguns pequenos trabalhos durante o curso para conseguir a renda necessária tentando atrapalhar o mínimo possível a faculdade, ou consegue-se um emprego que possa trazer para o futuro alguma coisa a mais. Em outras palavras, ou "vai empurrando" até o fim da faculdade ou "começa de vez". Durante o primeiro ano e meio eu arrumei trabalhos apenas para "ir empurrando". No entanto depois disso resolvi (por diversos motivos) entrar uma corretora de valores (esta na qual o projeto de que este texto trata foi desenvolvido.) Gostava do mercado financeiro, e o trabalho me parecia bastante promissor.

Por outro lado, considero muito importante que os alunos arrumem trabalhos na área no máximo no início do último ano. Já vi muitos alunos no IME sem a mais vaga idéia do que é trabalhar. Na minha opinião isso é extremamente ruim para eles. Embora o trabalho durante a faculdade tenha atrapalhado **muito** meu curso, não me arrependo.

1.2 Desafios e frustrações encontrados

Durante o desenvolvimento do trabalho surgiram diversos desafios. Diversas vezes havia tarefas que me pareciam simples e que se tornaram muito mais complexas e trabalhosas. Quando descobri que apenas 4 métodos na camada de persistência do **Hibernate** deveriam ser trocados me pareceu muito fácil fazer a substituição. No entanto, quando comecei a fazer a mudança, não só a criação de cada método era muito mais trabalhosa do que imaginava, como tive que criar 21 novos métodos que faziam o acesso por *stored procedures*! Outro grande desafio (este também uma pequena frustração) foi escrever *stored procedures* que fizessem o trabalho contido nas funções do núcleo do *Engine*. Em duas das cinco funções o ganho foi excelente. Entretanto, em dois métodos o resultado era sempre muito ruim. Em uma das funções, a versão inicial da *stored procedure* rendeu uma piora de desempenho de quase 100%! Ou seja, demorava quase o dobro que que antes de qualquer alteração! Depois de **muitas** mudanças o resultado ainda permanesceu péssimo.

Outro grande desafio foi criar um emulador para a Bovespa no ambiente de testes criado. Durante um bom tempo usei o próprio ambiente de testes da Bovespa para fazer os testes. Mas para conseguir fazer os testes a qualquer hora (e de casa) tive que criar o emulador. Embora a integração com a Bovespa seja extremamente simples, tentei criar uma aplicação que reagisse da mesma forma que a bolsa o faz. Isso se mostrou mais difícil do que aparentava.

O maior desafio (e também a maior frustração) na parte técnica foi tentar criar a métrica capaz de dar qualquer idéia de quanto uma aplicação pode ser melhorada tirando-se parte da lógica da aplicação para *stored procedures*. Não só não consegui a métrica como meus resultados não me apontaram numa direção clara. A única conclusão que obtive é que a melhora ***pode*** ser excelente.

Mas o maior desafio enfrentado foi conseguir arrumar tempo para fazer o desenvolvimento e os testes! O cronograma inicial foi completamente desrespeitado (no início do ano não consegui fazer muita coisa). O trabalho foi quase todo realizado no segundo semestre. Tive que contar com muita tolerância do meu orientador (o Jef! Já ficam aqui agradecimentos que reiterarei na seção apropriada!)

As maiores frustrações ficam por conta dos objetivos iniciais não terem sido atingidos. Todavia, acredito que os resultados alcançados são muito úteis. O que 'compensa' a frustração!

1.3 Partes do projeto não concluídas

Minha proposta inicial do trabalho previa a comparação de custo das soluções. Como não consegui uma métrica, essa comparação perde um pouco o sentido. Mas ainda seria possível fazer as comparações por custo levando-se em conta apenas a camada de persistência da aplicação (que mostraram uma tendência bem mais definida). Essa comparação, que me parecia bastante interessante, não foi feita.

Havia também a proposta de analisar opções de otimização, como cache. Por falta de tempo essa análise foi toda deixada de lado. Essa parte considero extremamente interessante e penso que seria por aí que continuaria este trabalho.

Integração com o curso

2.1 Disciplinas mais relevantes do BCC

Sem dúvida a maior contribuição do IME foi "ensinar a aprender". Essa frase (já bastante gasta!) explica bastante. Um bom exemplo é que em nenhum dos principais projetos que desenvolvi utilizei linguagens de programação estudadas no IME. Mesmo no IME diversas disciplinas exigiam a utilização de uma linguagem ou ferramenta específica que nunca havia sido ensinada. Isso é (a cultura de "independência"), na minha opinião, indispensável.

Todavia, diversas disciplinas contribuíram diretamente para o bom desenvolvimento deste trabalho. Entre elas destacam-se as seguintes.

Laboratórios de programação (MAC211 e MAC242) – Muito úteis para os primeiros contatos com um projeto completo. Também muito úteis (assim como algumas outras disciplinas) para ajudar a aprender a trabalhar em grupo.

Estruturas de Dados (MAC323) – Disciplina essencial. Nesta disciplina aprendi alguns dos conceitos mais importantes de todo o curso. Tive a sorte de ter tido um ótimo professor – o Yoshi. Todos os projetos de que participei foram profundamente afetados pelos conceitos estudados em ED. É uma das disciplinas que distinguem um bom profissional. Alguém sem os conceitos estudados nesta disciplina nunca será um bom desenvolvedor. Independente de quanta experiência obtenha.

Conceitos fundamentais de Linguagens de Programação (MAC316) – Esta disciplina é muito útil porque põe em contato com diversos conceitos muito diferentes em programação. O maior aprendizado agregado com esta disciplina, para mim, foi o conhecimento de linguagens funcionais. No meu caso contribuiu **muito** para criar aplicações mais segmentadas e bem divididas, facilitando muito a manutenção. A disciplina lecionada pelo prof. Alan fez os conceitos serem muito bem absorvidos, dada a exigência do professor!

Algoritmos em Grafos (MAC328) – Em muitas situações surge a idéia de *estado* e *mudança de estados*. No caso deste projeto estado de uma oferta, por exemplo. Esta disciplina é fundamental para conseguir descrever bem as situações e modelar uma solução correta.

Sistemas de Bancos de Dados (MAC426) – Neste projeto é, evidentemente, fundamental. Mas acredito que seja uma das disciplinas mais importantes do IME. Trata de tópicos e conceitos fundamentais para qualquer projeto grande.

Engenharia de Software (MAC332) – Essencial para o desenvolvimento de qualquer grande projeto. As práticas aprendidas não precisam, necessariamente, ser utilizadas. Mas os conceitos são essenciais.

Programação Orientada a Objetos e Tópicos de Programação Orientada a Objetos (MAC441 e MAC413) – A primeira é disciplina **fundamental**. Ao menos ela deveria ser, na minha opinião, obrigatória. O desenvolvimento de aplicações "modernas" é impossível sem os conceitos desta disciplina.

Programação Concorrente ou Introdução à Computação Paralela e Distribuída (MAC438 ou MAC431) – Outras disciplinas essenciais para projetos importantes. São disciplinas sem as quais alguém não se torna um bom desenvolvedor. Ambas são muito semelhantes.

O computador na Sociedade e na Empresa (MAC424) – Sem dúvida a disciplina de que mais gostei no IME. Acredito que seja, também, a disciplina que provocou o maior impacto em mim. Embora seja completamente dispensável para a formação de um desenvolvedor, é incrivelmente agregadora para qualquer um! O professor Setzer é incrível e recomendo fortemente este curso! Acredito que tive grande sorte de ter podido cursá-la.

Informação, Comunicação e a Sociedade do Conhecimento (MAC339) – Assim como MAC424 esta disciplina não é essencial para o curso. Mas, também assim como MAC424, agrega **muito** para quem cursa. O professor Imre Simon é excelente e os textos estudados são ótimos. Mais uma ótima disciplina para um curso tão técnico.

2.2 Conceitos fundamentais estudados no curso

Além do aprendizado da "capacidade de aprender sozinho", os conceitos técnicos mais importantes aprendidos no IME dizem respeito à natureza do desenvolvimento. Desta forma é possível se manter atualizado ou aprender qualquer tecnologia. Muitas disciplinas contribuíram muito fortemente para isso. Mas dentre elas destaco **Conceitos fundamentais de Linguagens de Programação**. Este contato com fundamentos das linguagens ajudam muito no desenvolvimento de aplicações estruturadas, ainda que as linguagens vistas dificilmente serão utilizadas na prática.

Especificamente os conceitos de **Lógica de Programação, Orientação a Objetos, Estruturas de Dados, Bancos de Dados, Grafos e Programação Concorrente** são essenciais na minha opinião. São os conceitos mais exigidos quando, na empresa em que trabalho, contratamos um novo funcionário.

Agradecimentos

Os agradecimentos iniciais ficam para meu orientador, o Prof. Dr. João Eduardo Ferreira (ou só Jef). Além da grande ajuda sobre qual caminho seguir para fazer um trabalho útil, sua paciência e tolerância foram de incomparável importância! Além do agradecimento fica o pedido de desculpas por não ter conseguido realizar o trabalho exatamente da forma como queríamos. Além do Jef a todos os professores que foram tão importantes neste longo tempo em que estou no IME.

Agradeço muito, também, à minha família – minha mãe e irmã. Sempre foram muito importantes em toda minha vida e passamos, juntos, por muitos momentos ótimos e outros muito difíceis.

Tenho a felicidade de ter muitos grandes amigos. São uma parte muito importante da minha vida. Para não encher esse texto de nomes, deixo um agradecimento a todos eles. Dos grandes amigos que persistem desde o colégio, aos amigos do IME (tive novamente a sorte de entrar numa turma incrível), amigos do GP (com quem habitualmente tomo uma cerveja toda sexta feita), amigos do trabalho e amigos que fiz durante toda a vida!

Por último à minha namorada, Júlia. Esta é mais uma das muitas grandes sortes que tive! Meus últimos anos, desde que estamos juntos, foram e têm sido os melhores. Ela deixa, sem dúvida, minha vida muito mais feliz. Agradeço a ela por tudo que é e por tudo que faz por mim. E agradeço, também, à minha grande sorte por tê-la!