Bean Validation, injeção de dependências com CDI e até transformar os managed beans em EJBs para ganhar os serviços do container.

Mesmo com diversas vantagens, o JSF não é unanimidade entre os desenvolvedores. Pelo fato de ser *Component Based*, ainda existem alguns empecilhos para sistemas Web. Entre eles, a falta de controle do HTML é um dos mais sérios. Como o HTML é gerado pelo próprio JSF, qualquer otimização ou modificação que queiramos fazer se torna extremamente custosa e complexa; em casos mais extremos, pode ser que o JSF não dê suporte a determinada funcionalidade. O design e a experiência do usuário são, em geral, adaptados pelas limitações que o JSF apresenta, em vez de uma abordagem mais livre no design, em que o código será adaptado para atingir o resultado desejado.

JSF segue um modelo menos focado na Web e mais voltado para a árvore de componentes. A Web é *apenas* o ambiente do JSF, mas seu objetivo final são os componentes. JSF acaba se tornando uma ferramenta poderosa para criar aplicações com interfaces muito complexas, com diversas funcionalidades, mas que não necessitem de um controle fino do HTML, JavaScript e CSS, ou uma proximidade maior das características do HTTP. Nesses casos, favoreça um framework *Action Based*.

6.3. Domine sua ferramenta de mapeamento objeto relacional

Consultas SQL, gerenciamento de transações, caches e a decisão de quando persistir e buscar dados do banco são preocupações frequentes em grande parte dos projetos e demandam muito tempo e trabalho. Todo código utilizado para resolver esses problemas raramente é escrito de forma a facilitar seu reúso, e o programador é obrigado a reescrever grande parte dele para acessar os dados ao iniciar um novo projeto.

Por este e outros motivos, soluções que visam facilitar o trabalho do programador, diminuindo a necessidade do uso do JDBC e a escrita de complicadas consultas SQL, apareceram há bastante tempo e ganham cada vez mais força.

O que há de errado com o JDBC?

JDBC é uma das APIs mais antigas do Java, existente desde o JDK 1.1 e com constantes atualizações. A versão 4.0, disponível no Java 6, trouxe melhorias significativas e oferece muito poder até mesmo a bancos de dados específicos.[16] O

código para executar uma simples transação com JDBC puro pode ser escrito de maneira bem sucinta.

```
public void insere(String email) throws SQLException {
   Connection con = abreConexao();
   con.setAutoCommit(false);
   con.createStatement().execute(
      "INSERT INTO Destinatario (email) VALUES ('" + email + "')";
   );
   con.commit();
   con.close();
}
```

Pode não estar óbvio, mas este tipo de código ingênuo é a causa dos principais problemas ao acessar bancos de dados com Java: vazamento de memória, transações não finalizadas corretamente, excesso de conexões abertas com o servidor pela ausência de pool, baixa escalabilidade, vazamento de conexões e até mesmo *SQL injection*.

Podemos tentar resolver esses problemas com um código mais completo.

```
public class DestinatarioDao {
   private final DataSource dataSource = new ComboPooledDataSource();

   public void insere (String email) throws DAOException {
      String sql = "INSERT INTO Destinatario (email) VALUES (?)";

      Connection con = null;
      PreparedStatement stmt = null;

      try {
         con = this.dataSource.getConnection();
         con.setAutoCommit(false);
         stmt = con.prepareStatement(sql);
         stmt.setString(1, email);
         stmt.execute();
         con.commit();
      } catch (SQLException e) {
         try {
            if (con != null)
               con.rollback();
            throw new DAOException("Erro no insert", e);
         } catch (SQLException e) {
            throw new DAOException("Erro no insert e rollback", e);
         }
      } finally {
         try {
            if (stmt != null)
               stmt.close();
         } catch (SQLException e) {
```

```
            throw new DAOException("Erro ao fechar statement", e);
         } finally {
            try {
               if (con != null)
                  con.close();
            } catch (SQLException e) {
               throw new DAOException("Erro ao fechar con", e);
            }
         }
      }
   }
}
```

O código mais robusto utiliza `PreparedStatement` para evitar SQL injection e tirar proveito de alguns bancos que pré-compilam as queries; toma cuidado com `try`, `catch`, `finally` para não deixar transações pendentes e conexões abertas; usa um pool de conexões através de um `DataSource` para evitar o vazamento destas e não utilizar conexões que possivelmente possam ter sido fechadas pelo servidor.[17]

E este código ainda pode ter melhorias. O esforço em lançar exceptions com boas mensagens e não deixar vazar `SQLException` poderia ser maior com mais blocos `try/catch` para tratar os erros separadamente, como no caso de falhar a abertura da conexão. É válido reparar que a utilização de inversão de controle ajudaria bastante o trabalho de evitar esse código repetitivo (o *boilerplate code*). Mesmo tomando todo este cuidado, há aqui um trabalho ainda de design, já que o programador deve decidir onde colocar o código SQL, isolando-o em DAOs, arquivos de *properties*, XMLs ou, ainda, anotações do JDBC 4.0.

Com a evolução da plataforma Java, surgiram diversas soluções para minimizar o risco e facilitar a implementação dos padrões que aparecem no código anterior.

A primeira opção seria abstrair o acesso direto ao JDBC tomando as devidas precauções e usando as melhores práticas. Ainda assim, teríamos muito trabalho e código repetido, e, por isso, ferramentas conhecidas como *data mappers* surgiram. Essas ferramentas liberam o programador da árdua tarefa de executar a consulta SQL, mapear as linhas da tabela nas entidades do seu modelo, gerenciar transações, etc.

Um framework bastante utilizado é o **iBatis**, considerado por alguns como um *data mapper* menos poderoso, que já encapsula as necessidades comuns do desenvolvedor, quando utiliza o JDBC, e chega até a organizar as queries SQLs em arquivos XML.[18] Apesar de não ser um *data mapper*, o **Spring JDBC template** é mais uma alternativa de uso. Esses tipos de *wrappers*, porém, são pouco usados.

Um dos reflexos desta perda de popularidade é que o iBatis parou sua evolução. Um fork foi criado, o **MyBatis**, mas a comunidade Java caminha cada vez mais para as ferramentas de mapeamento objeto relacional.

Mapeamento objeto relacional

O próximo passo seria utilizar uma ferramenta que facilitasse ainda mais o trabalho do desenvolvedor, fazendo com que ele deixasse até mesmo de escrever consultas SQL. Essas ferramentas são conhecidas como **ORM** (*object-relational mapping)*. Elas são capazes de fazer a ponte entre o paradigma entidade relacional e o orientado a objetos de forma a minimizar o abismo entre os conceitos dessas duas abordagens, conhecido como a impedância objeto-relacional (*object relational impedance mismatch*).[19]

Um exemplo são as tabelas compridas que aparecem com certa frequência no mundo relacional e, diferente de classes com muitos atributos na orientação a objetos, não são vistas com tanta repulsa. Ferramentas de mapeamento devem possibilitar que mais de uma classe seja utilizada para representar uma única tabela. O contrário também pode acontecer, como uma classe ser mapeada em mais de uma tabela.

Outro caso são as tabelas associativas simples. Elas são a forma de fazer o relacionamento *muitos para muitos* entre entidades no modelo relacional. Ao mapear isso, sua ferramenta deve arrumar uma forma elegante de esconder esses aglomerados de pares de chaves, provavelmente sem criar uma classe para tal e usando apenas coleções e arrays de objetos.

Já a herança é um conceito que não bate com modelagem relacional. Em alguns frameworks ORM, o uso da herança pode formar um sério gargalo de acordo com a forma de representação adotada no banco de dados (as conhecidas estratégias da JPA e do Hibernate), dado que diversas tabelas precisam ser consultadas em algumas queries polimórficas, em especial quando é impossível usar alguns recursos do SQL, como *unions.*[20] Muitos vão desaconselhar a herança, preferindo usar composição, conforme visto no tópico *Evite herança, favoreça composição*[21,20] (Figura 6.9).

Com a força que as especificações Java EE têm, é comum que a escolha por uma ferramenta ORM acabe caindo sobre uma das implementações da JPA. Dentre elas, os principais concorrentes são o Oracle TopLink, a OpenJPA, a BEA Kodo, o EclipseLink e o Hibernate, certamente a opção mais comum. Seja qual for sua escolha, há alguns cuidados que devem ser tomados.

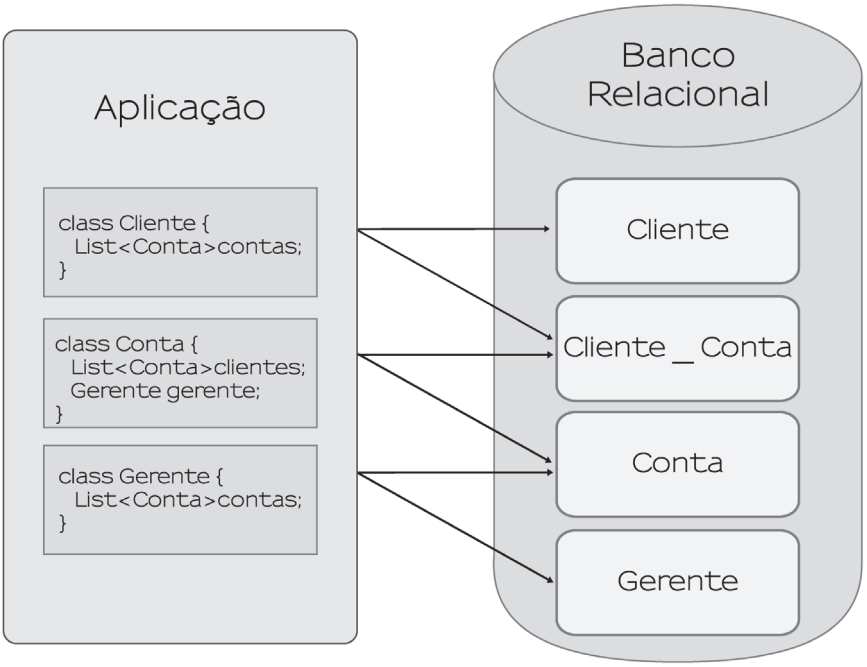

Figura 6.9 – Mapeamento de relacionamentos muitos para muitos e muitos para um.

Um cuidado importante é entender e usar corretamente os modos *eager* e *lazy* de se obter objetos relacionados. Sempre que temos relacionamentos com `@ManyToOne` ou `@OneToOne`, o padrão é trazer o objeto relacionado (*eager*); e, quando temos `@OneToMany` ou `@ManyToMany`, a coleção de objetos relacionados só é recuperada quando for acessada (*lazy*). Para boa parte dos casos, o comportamento padrão é suficiente, mas é preciso identificar casos em que precisamos configurar um `FetchType` diferente (por exemplo, quando um objeto relacionado é muito pesado e queremos comportamento *lazy* em vez de *eager*). É preciso estar atento ao uso indiscrimidado de lazy, que pode gerar o problema dos *n+1* selects, como veremos, assim como o uso indiscriminado de *eager* pode ocasionar carregamento desnecessário de dados em excesso.

Há também a possibilidade de se ajustar o *fetch type* diretamente nas consultas; por exemplo, fazendo uma busca específica como *eager* em que o relacionamento está configurado como *lazy* – usando *join fetch* nas HQL e o `setFetchMode` das Criterias.[22] Algumas implementações de JPA, como a OpenJPA e o Hibernate,

ainda permitem que campos básicos sejam *lazy* sem a necessidade de uma pré-compilação, possibilitando que uma entidade seja parcialmente carregada e que campos chaves sejam inicializados somente quando e se necessários.

Outro cuidado a ser tomado é na integração com o restante da aplicação. Na Web, acessamos entidades do nosso modelo na *View* e, se elas possuem relacionamentos *lazy*, precisamos que o `EntityManager` que as carregou esteja aberto até o fim da renderização da página. Caso esteja fechado, a maioria das implementações de JPA não vai se reconectar ao banco de dados e lançará uma exception (`LazyInitializaionException`, no caso do Hibernate).

Para que isso não ocorra, devemos manter o `EntityManager` aberto, seja através de um filtro, de um interceptador ou algum outro mecanismo. Este é o conhecido pattern *Open EntityManager in View* (derivado do *Open Session in View*, termo cunhado pelo Hibernate) (Figura 6.10).

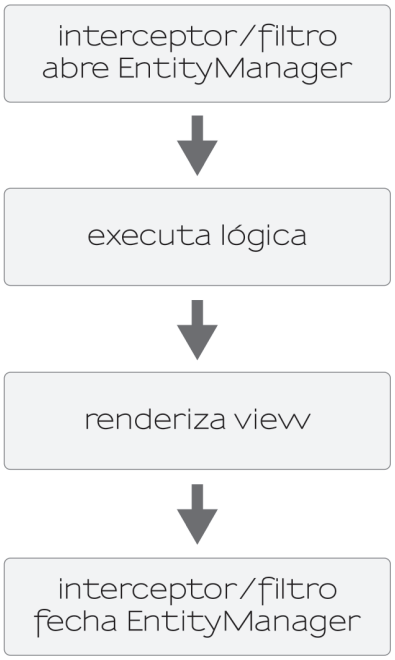

Figura 6.10 – Padrão Open EntityManager in View.

Outra forma de escapar dos problemas de proxies *lazy* é utilizar um container de injeção de dependências. Frameworks, como Spring e VRaptor, já possuem filtros e interceptadores para esses casos, e até mesmo usando a anotação

`@PersistenceContext` do Java EE para injeção de dependências podemos obter o mesmo efeito.[23]

Com o advento do Java EE 5 e algumas formas de injeção de dependências, diminuiu a necessidade de um grande número de patterns que havia para o J2EE, mas algumas preocupações ainda persistem. A principal delas envolve como acessar um `EntityManager`, uma pergunta para a qual existem diversas opções: dentro de um objeto que age como um DAO, em um Domain Model ou em um Model Façade. Ainda é possível, e cada vez mais fácil com o uso da injeção de variáveis membro, acessá-lo diretamente através da lógica de negócios. Apesar de este ainda ser um debate em que muitas opiniões divergem, a grande maioria dos desenvolvedores opta por proteger o acesso ao `EntityManager` através de um DAO, não permitindo que qualquer código o acesse e também ajudando a reaproveitar métodos que encapsulam rotinas de acesso a dados que são mais que apenas a execução de uma query.[24,25]

Por uma questão de ajuste de granularidade, para que não sejam expostos mais dados que o necessário, ou para agrupar diferentes atributos de entidades em um único resultado, é possível que haja a necessidade do *Data Transfer Object*.[26] Outros patterns do Java EE 5 também foram catalogados, mas muitos deles não foram largamente adotados pela comunidade, ou possuem nomenclatura e implementações divergentes.[27,26]

Connection Pool

A quase onipresença do Hibernate, tanto através do uso da JPA quanto diretamente, faz com que o conhecimento profundo das funcionalidades oferecidas por este framework mereça atenção especial. Apesar do esforço da JPA 1.0 em unificar uma grande parte de funcionalidades comuns a todas as ferramentas ORM em uma especificação, muito ficou de fora. A JPA 2.0 tenta diminuir essas grandes diferenças entre os diversos fabricantes existentes, criando novas anotações e estendendo a interface `EntityManager` com novos métodos e recursos.[28]

Entre os recursos importantes que existem no Hibernate e outros frameworks, porém não na JPA, podemos citar alguns que são vitais para a escalabilidade e a performance da aplicação. O desconhecimento dessas funcionalidades pode levar até mesmo um projeto simples ao fracasso, seja pelo consumo excessivo de memória, seja pelo disparo exagerado de queries.[29]

Usar um *pool de conexões*, por exemplo, é praticamente obrigatório em qualquer aplicação que use banco de dados. Um problema comum em aplicações que abrem conexões indiscriminadamente é sobrecarregar o banco com um número muito grande de conexões. O uso do pool permite controlar o máximo de conexões que podem ser abertas. Além disso, abrir e fechar conexões são operações custosas. Não vale a pena estabelecer toda a comunicação com o banco, abrir a conexão, para usá-la rapidamente, e depois logo fechá-la. Usar o pool economiza bastante tempo ao manter as conexões abertas e compartilhá-las.

Há diversos pools de conexão no mercado já implementados e prontos para uso, desde os próprios *data sources* dos servidores até bibliotecas separadas como o C3P0 e o DBCP. Usar um pool de conexões com o Hibernate é extremamente fácil, bastando umas poucas linhas de configuração XML,[30] lembrando-se apenas de evitar o pool padrão do framework, que deve ser utilizado apenas para testes.

Um problema tradicional dos pools está ligado à detecção de conexões quebradas, que ainda estão marcadas como disponíveis, causando a famosa `java.net.SocketException: Broken pipe` ao tentar utilizá-la. É necessário checar de alguma forma se a conexão de um pool ainda está válida, aberta, seja através de uma verificação, ao requisitar uma conexão, seja executando verificações periódicas.[17]

Cache e otimizando a performance

Outro ponto importante para uma boa performance e escalabilidade é evitar consultas repetidas em pequenos intervalos de tempo: o sistema responde rápido e o banco de dados não é acionado. Dentro de uma mesma `Session`, as entidades gerenciadas por ela ficam em um cache, o de primeiro nível (*first level cache*), até que essa sessão seja fechada ou a entidade explicitamente removida (através do `clear` ou `evict`). Esse primeiro nível de cache existe para garantir a integridade dos dados na memória e no banco, evitando que no contexto de uma `Session` possa existir mais de uma representação do mesmo registro ao mesmo tempo. É o que Fowler chama de *Identity Map* em seu POEAA.[1]

Pensando em um contexto Web, o cache de primeiro nível existe durante o ciclo de vida de uma requisição. É possível ir além, já que muitas entidades sofrem poucas alterações e podem ter seus dados cacheados por uma região maior que a delimitada por uma única requisição. O **cache de segundo nível** (*second level cache*) tem este papel. Através de simples anotações, é possível tirar proveito de robustos mecanismos de cache, como *Ehcache* ou *JBoss Cache*. É possível ainda configurar

detalhes como a política do ciclo de vida das entidades no cache (LRU, LFU, FIFO...), seu respectivo tamanho, os tempos de expiração, entre outros.[31]

Quando uma determinada entidade for buscada através de sua chave primária e ainda não estiver no cache de primeiro nível, o Hibernate passará pelo cache de segundo nível para verificar sua presença. Caso seu tempo de expiração não tenha passado, uma cópia desse objeto cacheado será devolvida sem necessidade de acesso ao banco de dados. Se as tabelas em questão são apenas acessadas por sua aplicação, é possível configurar o cache de tal maneira que nunca expire enquanto não houver uma atualização na entidade. O código a seguir mostra a utilização transparente de um cache de segundo de nível, uma vez que a segunda busca não atingirá o banco de dados:

```
EntityManager manager1 = jpa.getEntityManager();
Conta primeiroRetorno = manager1.find(Conta.class, 15);

EntityManager manager2 = jpa.getEntityManager();
Conta segundoRetorno = manager2.find(Conta.class, 15);

System.out.println("Titular da primeira conta: " +
  primeiroRetorno.getTitular());

System.out.println("Titular da segunda conta: " +
  segundoRetorno.getTitular());
```

Na Figura 6.11 é possível verificar como os dados são armazenados e encontrados no cache de primeiro e segundo níveis:

Fazer um cache eficiente, que evita vazamento de memória e problemas de concorrência sem perder escalabilidade, é uma tarefa difícil; ao adotar ferramentas de ORM que suportam tais bibliotecas, toda essa infraestrutura já está disponível. Mas toda estratégia de cache é perigosa porque, para evitar ficar com dados obsoletos, é preciso uma boa política de invalidação em caso de atualizações. O Hibernate faz tudo isso de forma transparente se as atualizações são feitas através dele. Mas, se nossos objetos tiverem muitas atualizações em comparação com o número de consultas, o efeito pode ser o inverso do desejado: o Hibernate gastará muito tempo nas invalidações de cache e a performance provavelmente será prejudicada. **Use cache apenas em entidades que não mudam nunca ou nas que mudam muito pouco em relação às consultas**.

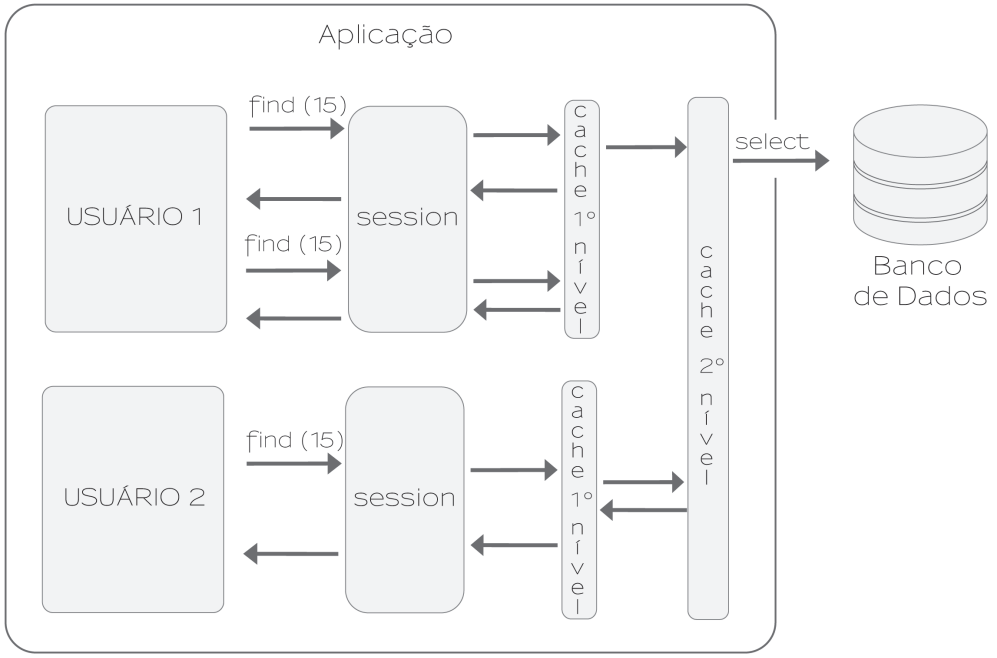

Figura 6.11 – Cache de primeiro e segundo níveis em ação.

A adoção de caches favorece performance e escalabilidade em troca de maior consumo de memória. Caches são ainda mais eficazes para aplicações nas quais dados desatualizados (*stale*) podem ser mostrados por determinado período de tempo sem prejuízo para o cliente. Por exemplo, uma query que retorna o produto mais vendido no dia pode ser cacheada por alguns minutos ou até horas.

O que foi apresentado até agora é o chamado cache de entidades. Mas frequentemente são executadas pesquisas que retornam o mesmo resultado até que as tabelas envolvidas sofram alterações. O Hibernate pode cachear até mesmo o resultado de queries, através do *query cache*. Guardar todo o resultado de uma query consumiria muita memória e dados repetidos, e é por isso que a ferramenta vai apenas armazenar as chaves primárias das entidades devolvidas. Quando esta for executada mais uma vez, o Hibernate pegará esse conjunto de chaves primárias e as buscará, sem a necessidade de executar a query de novo. Por isso, é importante também colocar essas entidades no cache de segundo nível, e não apenas a query, para que a nova execução não precise fazer nenhum hit no banco de dados, utilizando apenas os dados da memória para devolver o resultado.

Esse resultado de query cacheado será invalidado ao menor sinal de modificação nas tabelas envolvidas, pois isso pode alterar o que esta query retornaria. Queries que envolvam tabelas modificadas constantemente não são boas candidatas ao *query cache*.

É possível que o desenvolvedor não queira invalidar todo o cache de uma query a cada modificação nas respectivas tabelas, pois diversas vezes é aceitável que a resposta para uma query não seja consistente com os dados mais atuais, ou seja sabido que a atualização de determinados campos não influenciam o resultado das queries. Esse comportamento de invalidação do cache de queries pode ser modificado para um ajuste mais fino, através da substituição da `StandardQueryCache`, implementando diretamente a `QueryCache` ou estendendo o padrão, para que seja tolerante a alguns updates.[32]

Em queries muito pesadas e frequentemente acessadas, uma implementação simples, que verifica a existência da informação no cache e então executa a query, pode possuir gargalos. Se, por exemplo, no intervalo de 10 segundos, 200 usuários acessarem essa informação e ainda não havia dados no cache, a query será executada uma vez para cada um deles, já que o resultado só é colocado no cache após o retorno da primeira requisição. Esse problema, quando muitas requisições se acumulam em cima de um dado, que pode ou não estar cacheado, é chamado de *dog pile effect*.[33] O *double check locking* e outras boas práticas previnem situações como essas, e é importante conhecê-las.[34,35,36] A integração dos frameworks com as bibliotecas de cache já se previnem dessas situações, porém, ao utilizar a biblioteca de cache diretamente, e em alguns casos isto se faz necessário, você mesmo precisa tomar essas medidas.

Quando o volume de dados é muito grande, uma máquina pode ser pouco para guardar todo o cache. É possível utilizar um cluster para armazená-lo, e o Hibernate já trabalha com *Infinispan*, *Coherence* e *Memcache* de forma a diminuir a necessidade de configuração. Com eles, é possível fazer o *fine tuning* para minimizar o tráfego intracluster, além de desabilitar a sincronização imediata de estado com o banco de dados, usando o *write-behind*, uma abordagem assíncrona muito mais escalável, porém com menor confiabilidade, pois permite a visualização de dados desatualizados (*stale data*).[37,38]

Mais otimizações e recursos

Além das estratégias de cache, o Hibernate ainda disponibiliza outros mecanismos de otimização. Ao executar uma query, é comum iterar sobre todas as entidades retornadas. Se o resultado de uma query é um `List<Livro>`, e `Livro`

possui como atributo uma `List<Autor>` que será carregada de maneira *lazy*, existe então um potencial problema: ao fazer o laço para mostrar todos os autores desses livros, para cada invocação `livro.getAutores()` uma nova query será executada, por exemplo:

```
select * from Autor where livro_id = 1
select * from Autor where livro_id = 3
select * from Autor where livro_id = 7
select * from Autor where livro_id = 12
select * from Autor where livro_id = 15
select * from Autor where livro_id = 16
```

Este problema é conhecido como `n+1`, pois, ao executar uma única query, tem-se como efeito colateral a execução de mais `n` queries, onde `n` é o número de entidades resultante da query original e que pode ser bem grande.

É possível configurar o Hibernate para que ele inicialize a coleção *lazy* de autores dos livros de uma maneira mais agressiva, sem fazer cada carregamento separamente, mas sim de 5 em 5, valor que pode ser definido através da anotação `@BatchSize(size=5)`, que diminui bastante as queries de inicialização:

```
select * from Autor where livro_id in {1, 3, 7, 12, 15}
select * from Autor where livro_id in {16}
```

Assim, o número de queries executadas é minimizado, unindo diversas queries em uma única, diminuindo o número de *roundtrips* até o banco de dados.

Para detectar a presença do problema n+1 e outros, não é viável olhar o tempo todo o log de queries em produção, pois é fácil deixar escapar diversos gargalos. O Hibernate possui a classe `Statistics` que, se habilitada, armazenará dados sobre entidades, relacionamentos, queries e cache. É possível ver com detalhes quantas vezes cada query está sendo executada, os tempos máximos, mínimos e médios; e, ainda, quantas vezes o cache de segundo nível é invalidado em comparação a quantas vezes ele é usado; ou até quantas transações foram abertas e não comitadas, entre outros. Através desta classe, é muito mais fácil identificar gargalos e problemas para sabermos onde colocar um novo cache, onde usar o `@BatchSize`, onde é melhor ser *eager* ou *lazy*, etc.

Ainda falando em otimizações, o Hibernate é muitas vezes criticado em situações que envolvem o processamento de muitos dados de uma vez. Mas existem mecanismos específicos para trabalhar nesses casos.[39] Imagine que seja necessário fazer o processamento de um imenso arquivo TXT e inserir mi-

lhões de linhas no banco de dados de uma vez. A invocação de `session.save` (ou `entityManager.persist`) várias vezes, provavelmente, vai resultar em uma `OutOfMemoryError`.

A noção de contexto persistente atrelado à sessão faz com que todos os objetos inseridos sejam armazenados no cache de primeiro nível até que ela seja fechada. E inserir muitos objetos nesse cache não é uma boa ideia. Soluções possíveis vão desde chamar `session.clear` de vez em quando para limpar esse cache, até usar diretamente a `StatelessSession`, outro tipo de sessão que não guarda estado (e por isso perde vários benefícios do Hibernate, mas resolve problemas de manipulação de muitos objetos).

Já quando o problema for trazer muitos registros do banco para a memória, melhor que executar uma simples *query* e pegar uma imensa lista de objetos que provavelmente estourará a memória, é estudar estratégias de como pegar poucos objetos por vez. Nesse cenário, os `ScrollableResults` permitem a utilização do cursor diretamente no banco de dados, evitando trazer todos os dados de uma vez para a memória.

Deve-se considerar até se vale a pena trazer os elementos para a memória. Muitos processamentos podem ser feitos diretamente no banco de dados através de *batch updates*. Por exemplo, reajustar em 10% os preços de todos os produtos de uma loja: em vez de trazer todos os produtos para a memória e atualizar chamando o `setPreco`, podemos executar um único *UPDATE* diretamente no banco de dados. O Hibernate suporta esse tipo de processamento através de *UPDATE*, *DELETE* e *INSERT... SELECT* diretamente pela HQL.

É também possível executar queries nativas pelo Hibernate. Embora uma das grandes facilidades do framework seja a geração das queries, existem situações nas quais queremos escrever algumas delas diretamente. Um bom motivo é aproveitar as queries complexas, escritas pelo DBA, com todos os detalhes de otimização do banco, em vez de tentar traduzir tudo para o HQL. Com queries nativas, conseguimos chamar recursos bem específicos do banco de dados que estamos usando, inclusive *stored procedures*, e obter um grau a mais de otimização nas situações em que performance for mais importante que portabilidade de banco de dados.

Há ainda diversas outras funcionalidades, como a existência do modo de *fetch extra lazy*, fazendo com que uma coleção não seja inteiramente inicializada: queries diferentes serão disparadas para chamadas, como `lista.size()`, `lista.get(15)` e até mesmo `lista.add (entidade)`, evitando carregamento de dados desnecessá-

rios. Ou, ainda, as funcionalidades de auditoria do Hibernate, que permitem manter um histórico completo de alterações em cada entidade do sistema.

Além de funcionalidades próprias, o Hiberante possui diversos subprojetos que auxiliam tarefas comumente relacionadas ao banco de dados. O **Hibernate Search**, por exemplo, integra de maneira elegante o Lucene com o Hibernate, permitindo que seu índice de pequisa textual seja atualizado de maneira transparente através de listeners, e possui mecanismos de alta escalabilidade para fazer a atualização destes de maneira assíncrona. A busca textual muitas vezes é deixada de lado por desconhecimento do Lucene e desta integração, gerando queries complexas que geram resultados pobres, já que o SQL não permite buscas com o mesmo poder.

Não usar algumas dessas poderosas funcionalidades pode dificultar o desenvolvimento da aplicação, além de possivelmente criar um enorme gargalo de performance. É importante conhecer a fundo as funcionalidades que afetam a performance do Hibernate,[40] além de todos os outros recursos para aumentar a produtividade do programador. Novamente aqui, **usar um framework sem um conhecimento significativo sobre suas funcionalidades pode causar problemas que seriam facilmente evitados.**

6.4. Distribuição de objetos

Distribuir objetos em diferentes máquinas é uma proposta comum para tentar melhorar a escalabilidade de uma aplicação. Mas as dificuldades e desvantagens que uma arquitetura distribuída apresenta podem ser mais prejudiciais à performance e manutenibilidade do que os ganhos.

Essas vantagens e desvantagens da distribuição dos objetos dependem do design adotado pela implementação, que pode induzir os desenvolvedores a cometer erros com facilidade. Historicamente, por escolhas inadequadas no design de diversas tecnologias de objetos distribuídos, criaram-se design patterns para resolver os problemas inerentes à remotabilidade, como os relacionados à perda de escalabilidade ao fazer invocações remotas em excesso sem necessidade, ou ao transportar um volume alto de dados entre os *tiers*.

Com cada versão nova do EJB, por exemplo, as invocações remotas ficam mais transparentes e se aproximam demais a uma invocação local, quando olhamos apenas o código. O desenvolvedor sem muito conhecimento pode acabar codificando e gerando inúmeras requisições entre tiers, causando perda de performance e escalabilidade, problemas comuns ao empregar tais soluções sem o devido cuidado.