Carolina Fernanda da Silva

R.A.: 0402013

ANÁLISE E AVALIAÇÃO DO FRAMEWORK HIBERNATE EM UMA APLICAÇÃO CLIENTE/SERVIDOR.

Jaguariúna

2007

Carolina Fernanda da Silva

R.A.: 0402013

ANÁLISE E AVALIAÇÃO DO FRAMEWORK HIBERNATE EM UMA APLICAÇÃO CLIENTE/SERVIDOR.

Monografia apresentada à disciplina Trabalho de Conclusão de Ciência da Computação da Faculdade de Jaguariúna, sob a orientação do Professor Ms. Ricardo Menezes Salgado, como exigência parcial para obtenção do grau de Bacharel em Ciência da Computação.

Jaguariúna

2007

SILVA, Carolina Fernanda. **Análise e Avaliação do Framework Hibernate em uma Aplicação Cliente/Servidor**. Monografia defendida e aprovada na FAJ em 11 de Dezembro de 2007 pela banca examinadora constituída pelos professores:

Prof. Ms. Ricardo Menezes Salgado
FAJ - Orientador

Prof. Ms. Fernando Augusto Zancheta
FAJ

Prof. Paula Saretta
FAJ

SILVA, Carolina Fernanda. **Análise e Avaliação do Framework Hibernate em uma Aplicação Cliente/Servidor. 2007**. Monografia (Bacharelado em Ciência da Computação) - Ciência da Computação, Jaguariúna.

RESUMO

O modelo de programação orientado a objetos e o modelo relacional de banco de dados se destacam no mercado atual, porém é necessária uma compatibilidade entre esses modelos, um processo de conversão chamado mapeamento objeto-relacional. Realizar esse mapeamento de forma manual é muito trabalhoso, por outro lado existem ferramentas que automatizam essa tarefa. O sistema *Hibernate* promove um estreitamento entre a tecnologia relacional e orientada a objetos garantindo interdependência de bancos de dados relacionais e proporcionando uma redução de esforço do desenvolvedor em suas tarefas de persistência de dados.

Nesse trabalho são abordadas técnicas de mapeamento, assim como as funcionalidades do *framework Hibernate*, apresentando suas vantagens e desvantagens no seu uso na camada de persistência. O *Hibernate* cria uma camada independente, não sendo necessário alterar a aplicação em casos de eventuais migrações ou mudança de banco de dados. A sua utilização é transparente, dispensando o desenvolvedor do aprendizado de sintaxe e comandos SQL, ocorrendo a redução significativa de esforço, prazo, custo e manutenção, aumentando assim a produtividade.

Palavras-chave: *Hibernate*, Banco de dados Relacionais, Mapeamento objeto/relacional.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

SQL Structured Query Language

JDBC Java Database Connectivity

MOR Mapeamento objeto/relacional

DBA Administrador de Banco de Dados

SGDB Sistema Gerenciador de Banco de Dados

XML Extensible Markup Language

BDOO Banco de Dados Orientado a Objetos

SEQUEL Strutured English Query Language

ANSI American National Standards Institute

ISSO International Standards Organization

DDL Data Definition Language

DML Data Manipulation Language

DQL Data Query Language

DCL Data Control Language

1FN Primeira Forma Normal

2FN Segunda Forma Normal

3FN Terceira Forma Normal

4FN Quarta Forma Normal

5FN Quinta Forma Normal

E-R entidade-relacionamento

UML Unified Modeling Language

API Application Programming Interface

HQL Hibernate Query Language

J2EE Java 2 Enterprise Edition

JDK Java Development Kit

JPA Java Persistence API

DAO Data Access Object

EJB Enterprise JavaBean

JNDI Java Naming and Directory Interface

JTA Java Transaction API (JTA)

1 INTRODUÇÃO

Na última década, o modelo de programação, projeto e modelagem orientado a objetos tornou-se o mais avançado modelo de desenvolvimento de software, oferecendo aumento da produtividade, segurança e facilidade de manutenção.

A orientação a objetos é um padrão de sistemas que abstrai a realidade apontando diversas entidades com atributos e operações comuns a ser automatizado por objetos que abrangem dados e métodos. Um objeto é um conceito, uma abstração, algo com limites nítidos e significados no contexto do problema em causa (PINHEIRO *apud* RUMBAUGH,1994).

Em paralelo, os bancos de dados relacionais são utilizados para o gerenciamento de grandes volumes de dados e oferece pesquisas rápidas, integridade referencial, compartilhamento de informações de forma segura e o gerenciamento de acessos. Atualmente, dominam o mercado, pois apresentam uma visão única, não redundante e resumida dos dados de uma aplicação.

A medida que os desenvolvedores dominam os sistemas de gerenciamento de bancos de dados relacionais, compreendem *Structured Query Language* (SQL), trabalham com tabelas e chaves estrangeiras, os mesmos fazem uso da técnica mais comum para persistência de dados em *Java*, ou seja, trabalham diretamente com SQL e *Java Database Connectivity* (JDBC).

. Contudo, o processo envolvendo o código de persistência manual é trabalhoso, principalmente quando se trata de diversos dialetos SQL, exigindo maior esforço de desenvolvimento. Sendo assim, se forem modificadas as exigências, será necessário uma maior dedicação e esforço para manutenção.

Como banco de dados relacionais caracteriza-se por possuir uma tecnologia mais difundida no gerenciamento de dados apresentando melhor desempenho e segurança, houve o crescimento da demanda por essa tecnologia, pois há a necessidade de ferramentas de suporte a decisões, que estão mais presentes nas grandes organizações. Já o sistema de banco de dados orientado a objetos possui desvantagens no âmbito de não atender transações de sistemas legados, além de aplicações baseadas em SQL.

> Dessa maneira, de acordo com Rumbaugh (1994) os sistemas de bancos de dados baseados em objetos prometem melhor desempenho e maior facilidade de uso em longo prazo, porém, eles ainda não estão maduros como os sistemas de bancos de dados relacionais convencionais causando problemas de integração com as aplicações convencionais existentes.
>
> Sendo assim, existe uma forte tendência de utilização de bancos de dados relacionais para armazenamento dos objetos de aplicações orientadas a objetos. Contudo, existem diferenças significativas do paradigma orientado a objetos para o modelo relacional. Para que seja possível obter os benefícios de ambos, é necessária uma compatibilidade entre esses modelos.

A aplicação passa a necessitar de um processo intermediário de conversão. O mapeamento objeto-relacional é uma técnica de desenvolvimento para reduzir as diferenças potenciais entre as tecnologias orientadas a objetos utilizando bancos de dados relacionais. Realizar este mapeamento de forma manual é muito custoso e dedica muito tempo na construção e manutenção. Por outro lado, existem ferramentas que automatizam essa tarefa, promovendo um estreitamento entre as tecnologias.

Utilizando-se como meio de armazenamento o banco de dados relacional dentro de uma arquitetura adotando a persistência de objetos, a mesma terá de possuir uma interligação para traduzir o modelo orientado a objeto para um modelo relacional em razão das divergências de representação desses modelos.

Já persistência é a habilidade de um objeto sobreviver ao ciclo de vida do processo no qual ele reside (DAIBERT *apud* KELLER, 2004). A persistência do objeto pode ser considerada transparente, pois o desenvolvedor produz operações de manipulação de dados de objetos em sua programação sem notar que há uma arquitetura promovendo funcionalidades no decorrer dessas manipulações. Esse processo permite o código limpo, livre de mapeamentos e funções que não pertencem ao seu escopo. O conjunto de funcionalidades da persistência deve se restringir apenas a operações de inclusão, atualização, consulta e exclusão.

O modelo relacional é o mais utilizado na representação e compreensão dos dados que integram um sistema de informações. No desenvolvimento de aplicações em banco de dados a modelagem dos mesmos proporciona uma visão concisa e compacta em uma aplicação.

O mapeamento é o ato de se determinar como os objetos e seus relacionamentos são persistidos em um mecanismo de armazenamento de dados permanente seguindo um modelo relacional (PINHEIRO *apud* AMBLER, 2000). O mapeamento objeto/relacional (MOR) constrói uma ponte entre o objeto e *schemas* relacionais, permitindo que a

aplicação persista objetos diretamente fazendo a conversão de objetos em um formato relacional, ou seja, trabalha por reversibilidade transformando dados de uma representação para outra.

De acordo com Bauer e King (2005, p. 39), “MOR não é uma bala prateada para toda a tarefa de persistência; seu trabalho é aliviar o desenvolvedor de 95 por cento de trabalho de persistência de objeto”. Segundo Ambler 2007, “o mapeamento objeto/relacional permite que os desenvolvedores de sistemas se concentrem no que fazem de melhor, ou seja, desenvolver aplicações sem ter a preocupação sobre como os objetos serão armazenados”. Além disso, o MOR deve permitir que os administradores de dados (DBAs) administrem bancos de dados desconsiderando a introdução de erros em aplicações existentes.

1.1 Motivação

> Atualmente, os sistemas gerenciadores de bancos de dados (SGBDs) possuem um lugar de destaque se comparados com as tecnologias de armazenamento promovendo confiabilidade e robustez, porém não possuem características para armazenar objetos.

Desse modo, surge a necessidade do mapeamento dos dados onde os mesmos são persistidos de forma correta e íntegra.

Esse mapeamento, quando usado no contexto de orientação a objetos, adiciona complexidade extra, necessitando de um processo intermediário de conversão.

Segundo Bauer e King (2005), o projeto *Hibernate* é um *framework*[1] que aplica as metodologias de padrões de projeto, promovendo um estreitamento entre a tecnologia relacional e orientada a objetos. Além de possuir código aberto e ter uma comunidade bastante ativa, promovendo interdependência de bancos de dados relacionais e proporcionando uma redução de esforço do desenvolvedor em suas tarefas de persistência de dados.

1 Framework: Conjunto de vários elementos de programação, com foco numa área específica, para resolver problemas computacionais dessa e outras áreas correlatas.

2 OBJETIVO

Descrever os objetivos, vantagens e desvantagens do *Hibernate*, realizando a persistência de objetos em diferentes SGBDs, apresentando os aspectos de sua arquitetura a fim de conhecer seu papel dentro de aplicações desenvolvidas no paradigma orientado a objetos.

A meta é unir tecnologia de desenvolvimento de softwares baseados na orientação a objetos com a tecnologia de armazenamento de dados relacional proporcionando transparência aos desenvolvedores na manipulação de informações de objetos em um mecanismo de persistência.

Foram implementados dois softwares de leitura, inserção e atualização de dados de carga elétrica ativa em base horária (MW/h) medidos em vários pontos de uma empresa distribuidora da região nordeste do Brasil.

3 METODOLOGIA

Para realizar os testes de desempenho nos bancos de dados, foi utilizada a ferramenta *Apache JMeter*, permitindo a análise e visualização das estatísticas por meio de gráficos.

Segundo Fonseca, "o *JMeter* permite simular uma carga pesada de usuários em uma rede, com o objeto dos testes é de analisar o desempenho total sob tipos diferentes da carga". Tem como objetivo realizar testes de caixa cinza e preta, executando testes de stress, validando requisitos não funcionais do software.

3.1 Plano de testes

Primeiro plano de teste

- Simulação de dez usuários conectados ao banco MySQL;
- A operação SQL relacionada foi executada dentro de um *loop* em 100 vezes;
- Cada usuário realizou 5 requisições, citada abaixo:

```
SELECT * FROM Cargas;
```

A tabela foi populada com a seguinte quantidade de dados:

Cargas 500 linhas

Segundo plano de teste

- Simulação de dez usuários conectados ao banco PostgreSQL;
- A operação SQL relacionada foi executada dentro de um *loop* em 100 vezes;
- Cada usuário realizou 5 requisições, citada abaixo:

SELECT * FROM Cargas;

A tabela foi populada com a seguinte quantidade de dados:

Cargas 500 linhas

Terceiro plano de teste

A simulação foi realizada em ambos os SGBDs;
A consulta foi executada dentro de um *loop* de 100 vezes;

Foram realizadas as seguintes consultas:

Consulta uma tabela
SELECT * FROM Sin_Regional;

Consulta duas tabelas
SELECT * FROM Sin_Regional;
SELECT * FROM Sist_Transmissao;

Consulta quatro tabelas
SELECT * FROM Sin_Regional;

SELECT * FROM Sist_Transmissao;
SELECT * FROM Barramentos;
SELECT * FROM Cargas;

As tabelas foram populadas com a seguinte quantidade de dados:

Sin_Regional	15 linhas
Sist_Transmissao	15 linhas
Barramentos	15 linhas
Cargas	500 linhas

Quarto plano de teste

A simulação foi realizada em ambos os SGBDs;
A consulta foi executada dentro de um *loop* de 100 vezes;

Foram realizadas as seguintes consultas:

Consulta uma tabela
SELECT * FROM Sin_Regional;

Consulta duas tabelas
SELECT * FROM Sin_Regional;
SELECT * FROM Sist_Transmissao;

Consulta quatro tabelas
SELECT * FROM Sin_Regional;
SELECT * FROM Sist_Transmissao;
SELECT * FROM Barramentos;
SELECT * FROM Cargas;

As tabelas foram populadas com a seguinte quantidade de dados:

Sin_Regional	150 linhas
Sist_Transmissao	150 linhas
Barramentos	150 linhas
Cargas	1000 linhas

Quinto plano de testes

A simulação foi realizada em ambos os SGBDs;
A consulta foi executada dentro de um *loop* de 100 vezes;
Foi realizada a seguinte inclusão:

INSERT INTO Cargas VALUES (, 1, 2, 3, 4, 555, 555, 555, 555, 555, 555, 555, 555, 555, 555, 555, 555, 555, 555, 555, 555, 555, 555, 555, 555, 555, 555, 555, 555, 555, 555, 555, 555, 555, 555, 555, 555);

Sexto plano de testes

A simulação foi realizada em ambos os SGBDs;
A consulta foi executada dentro de um *loop* de 100 vezes;
Foi realizada a seguinte exclusão:
DELETE FROM Cargas WHERE carg_hor1 = 555

4 ARQUITETURA EM CAMADAS

No decorrer dos anos até atualmente, se busca o aprimoramento de padrões e técnicas para o desenvolvimento de sistemas de maior qualidade e manutenção simplificada.

Coesão e acoplamento são as técnicas inicialmente criadas durante o período em que se utilizavam sistemas estruturados. Segundo Calçado 2006, a coesão agrupa elementos de acordo com sua similaridade. O acoplamento caracteriza-se na independência de componentes, onde se pode alterar determinado componente, sem que haja danos em outros. De acordo com Calçado 2006, o uso de camadas se desenvolve:

> A aplicação dessas métricas não está restrita a objetos, trechos de código ou atributos, elas também são aplicadas à arquitetura de uma aplicação. O uso de Camadas (*Layers*) é um Padrão Arquitetural que ajuda na tarefa de separar responsabilidades, promovendo baixo acoplamento e alta coesão em um sistema (Calçado, 2006, sem p.).

A camada será responsável por agrupar classes, pacotes e componentes com características em comum. As classes que possuem função similar serão reunidas em determinada camada, promovendo a coesão e evitando o acoplamento, pois é controlada a forma que as camadas se comunicam. De acordo com Calçado (2006), o uso de camadas permite uma série de vantagens, sendo elas:

- Reduzem complexidade: agrupam componentes e simplificam a comunicação entre eles;
- Reduzem dependência/acoplamento: a regra de comunicação evita dependências diretas entre componentes de camadas diferentes;
- Favorecem a coesão: componentes de responsabilidades relacionadas são agrupados;
- Promovem reusabilidade: camadas podem ser reutilizadas em outros sistemas ou podem ser substituídas;
- É um Padrão Arquitetural conhecido: facilita a comunicação e entendimento entre desenvolvedores.

Entretanto, encontram-se também as desvantagens:

- Limitadas pela tecnologia: algumas regras precisam ser quebradas por limitações tecnológicas;
- Apenas complicam um sistema muito simples: não é qualquer sistema que exige o uso de camadas;
- Possibilidade de overdose: muitos arquitetos acabam criando camadas demais e tornando a aplicação extremamente complexa.

As camadas permitem diversos modos de arranjos, sendo que a quantidade varia de acordo com o tamanho da aplicação e da arquitetura estipulada. Entretanto, com um menor número de camadas, o desempenho da aplicação será melhor, sendo necessário o equilíbrio entre o número de camadas utilizadas e sua arquitetura para obter maior manutenibilidade e menores impactos dentro do sistema. Segundo King (2005, p. 23), "uma arquitetura de aplicativo típica, provada, alto-nivelada usa três camadas, uma para cada apresentação, lógica de negocio e persistência", como mostra a figura 1.

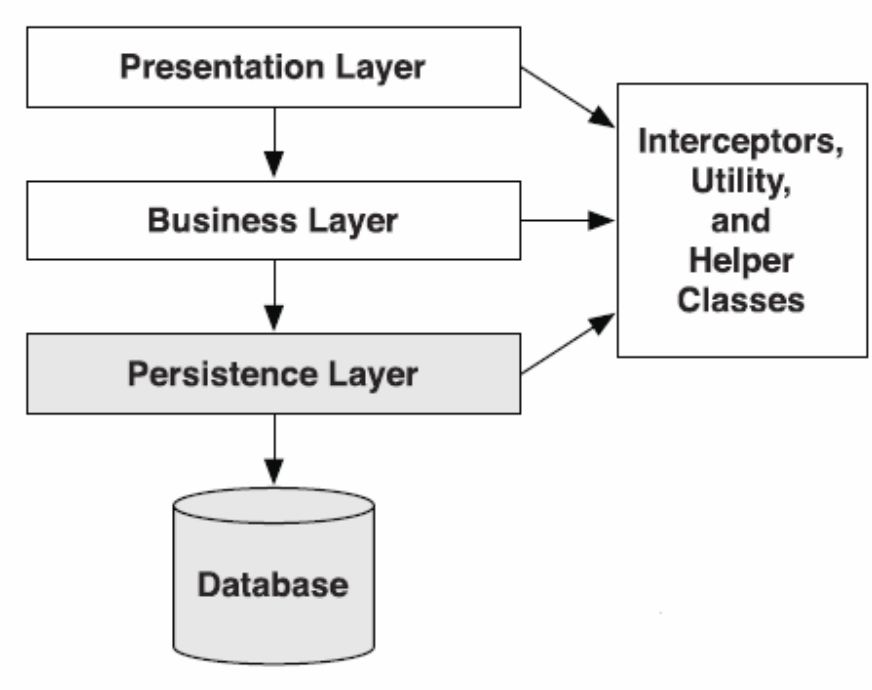

Figura 1 – Arquitetura em camadas (Bauer e King, 2005)

Camada de apresentação (*Presentation Layer*): a lógica de interface do usuário é o ponto mais alto. O código responsável para a apresentação e controle de página e tela de navegação forma a camada de apresentação.

Camada de negocio (*Bussiness Layer*): é geralmente acordado que esta camada de negócio seja responsável pela implementação de qualquer regra de negócio ou requerimentos do sistema que seriam entendidos por usuários como parte do problema de domínio.

Camada de persistência (*Persistence Layer*): é caracterizada por um grupo de classes e componentes responsáveis pela memória de dados e sua recuperação, um ou mais armazenamento de dados. Esta camada necessariamente inclui um modelo das entidades de domínio de negocio.

Banco de dados (*Database*): o banco de dados existe fora do aplicativo *Java*. É a atual representação persistente do estado de sistema. Se um banco de dados de SQL for usado, o banco de dados incluirá o esquema relacional e procedimentos possivelmente armazenados.

Assistente/classes de utilidade (*Utility and Helper Classes*): todo o aplicativo tem um set de assistente de infra-estrutura ou classes de utilidade que é usado em toda camada do aplicativo (por exemplo, classes *Exception* para tratamento de erros). Esses elementos de infra-estrutura não formam uma camada, desde que eles não obedeçam às regras para dependência entre camadas na arquitetura de camadas.

4.1 Evolução dos modelos de persistência de objetos

Buscando rapidez na implementação, os sistemas adotam a inclusão de código SQL dentro da lógica, dificultando a manutenção de código e um acoplamento do sistema ao SGDB utilizado. Em caso de qualquer mudança na estrutura das tabelas, todo o código poderá ser afetado, sendo assim reescrito.

Outra opção é colocar o código SQL em classes de acesso a dados, ou seja, separadas das classes da aplicação, como meio de restringir o impacto das mudanças no sistema como um todo trazendo maior controle ao escopo dos possíveis erros. A camada de persistência permite a abstração de dados, promovendo a interdependência entre o modelo de objetos e o banco de dados. Permite alterações na estrutura ou banco de dados sem causar impactos na aplicação, além de permitir o armazenamento em SGBDs orientados a objetos, objetos relacionais e arquivos *Extensible Markup Language* (XML).

Segundo Júnior, uma camada de persistência de objetos funciona como uma biblioteca permitindo a realização do processo de persistência, de forma transparente. É possível controlar de forma simples o gerenciamento de um modelo de objetos em vários

tipos de repositórios. Fazendo uso dessa concepção, métodos para incluir, alterar e remover objetos, permite ao desenvolvedor trabalhar como se estivesse em um sistema completamente orientado a objetos. Desse modo de acordo com Júnior 2003, a utilização de camada de persistência envolve:

> As vantagens decorrentes do uso de uma camada de persistência no desenvolvimento de aplicações são evidentes: a sua utilização isola os acessos realizados diretamente ao banco de dados na aplicação, bem como centraliza os processos de construção de consultas (*queries*) e operações de manipulação de dados (*insert*, *update* e *delete*) em uma camada de objetos inacessível ao programador. Este encapsulamento de responsabilidades garante maior confiabilidade às aplicações e permite que, em alguns casos, o próprio SGBD ou a estrutura de suas tabelas possam ser modificados, sem trazer impacto à aplicação nem forçar a revisão e recompilação de códigos (Júnior, 2003, sem p.).

4.1.1 Requisitos de uma Camada de Persistência

Uma camada de persistência real deve implementar as seguintes características, (JÚNIOR *apud* AMBLER 2003):

- Dar suporte a diversos tipos de mecanismos de persistência: um mecanismo de persistência pode ser definido como a estrutura que armazenará os dados - seja ela um SGBD relacional, um arquivo XML ou um banco de dados orientado a objetos (BDOO), por exemplo. Uma camada de persistência deve suportar a substituição deste mecanismo livremente e permitir a gravação de estado de objetos em qualquer um destes meios.

- Encapsulamento completo da camada de dados: o usuário do sistema de persistência de dados deve utilizar-se, no máximo, de mensagens de alto nível como *save* ou *delete* para lidar com a persistência dos objetos, deixando o tratamento destas mensagens para a camada de persistência em si.

- Ações com multi-objetos: suportar listas de objetos sendo instanciadas e retornadas da base de dados deve ser um item comum para qualquer implementação, tendo em vista a freqüência desta situação.

- Transações: ao utilizar-se da camada de persistência, o programador deve ser capaz de controlar o fluxo da transação - ou ter garantias sobre o mesmo, caso a própria camada de persistência preste este controle.

- Extensibilidade: a camada de persistência deve permitir a adição de novas classes ao esquema e a modificação fácil do mecanismo de persistência.

- Identificadores de objetos: a implementação de algoritmos de geração de chaves de identificação garante que a aplicação trabalhará com objetos com identidade única e sincronizada entre o banco de dados e a aplicação.

- Cursores e *Proxies*: as implementações de serviços de persistência devem ter ciência de que, em muitos casos, os objetos armazenados são muito grandes - e recuperá-los por completo a cada consulta não é uma boa idéia. Técnicas como o *lazy loading* (carregamento tardio) utilizam-se dos *proxies* para garantir que atributos só serão carregados à medida que forem importantes para o cliente e do conceito de cursores para manter registro da posição dos objetos no banco de dados (e em suas tabelas específicas).

- Registros: apesar da idéia de trabalhar-se apenas com objetos, as camadas de persistência devem, no geral, dispor de um mecanismo de recuperação de registros - conjuntos de colunas não encapsuladas na forma de objetos, como resultado de suas consultas. Isto permite integrar as camadas de persistências a mecanismos de geração de relatórios que não trabalham com objetos, por exemplo, além de permitir a recuperação de atributos de diversos objetos relacionados com uma só consulta.

- Arquiteturas Múltiplas: o suporte a ambientes de programas *stand-alone*, cenários onde o banco de dados encontra-se em um servidor central e mesmo arquiteturas mais complexas (em várias camadas) deve ser inerente à camada de persistência,

já que a mesma deve visar a reusabilidade e fácil adaptação a arquiteturas distintas.

- Diversas versões de banco de dados e fabricantes: a camada de persistência deve tratar de reconhecer diferenças de recursos, sintaxe e outras minúcias existentes no acesso aos bancos de dados suportados, isolando isto do usuário do mecanismo e garantindo portabilidade entre plataformas.

- Múltiplas conexões: um gerenciamento de conexões (usualmente utilizando-se de *pooling*) é uma técnica que garante que vários usuários utilizarão o sistema simultaneamente sem quedas de performance.

- *Queries* SQL: apesar do poder trazido pela abstração em objetos, este mecanismo não é funcional em cem por cento dos casos. Para os casos extremos, a camada de persistência deve prover um mecanismo de *queries* que permita o acesso direto aos dados - ou então algum tipo de linguagem de consulta simular à SQL, de forma a permitir consultas com um grau de complexidade maior que o comum.

- Controle de Concorrência: acesso concorrente a dados pode levar a inconsistências. Para prever e evitar problemas decorrentes do acesso simultâneo, a camada de persistência deve prover algum tipo de mecanismo de controle de acesso. Este controle geralmente é feito utilizando-se dois níveis - com o travamento pessimístico (*pessimistic locking*), as linhas no banco de dados relativas ao objeto acessado por um usuário são travadas e tornam-se inacessíveis a outros usuários até o mesmo liberar o objeto. No mecanismo otimístico (*optimistic locking*), toda a edição é feita em memória, permitindo que outros usuários venham a modificar o objeto.

5 SQL

Desde o princípio do uso do computador é de conhecimento geral que o mesmo tem como função receber dados, processá-los e gerar a saída dos mesmos. Posteriormente surge

a necessidade de armazenar esses dados produzidos por computadores, sendo que esse armazenamento e recuperação de informações têm papel primordial na informática.

A criação da linguagem de banco de dados relacionais, a SQL, foi criada em 1970 com E. F. Codd. Segundo Oliveira (2002 p. 17), "Codd estabeleceu princípios sobre gerência de banco de dados, denominando-os com o termo relacional. Essa foi a base utilizada na criação de uma linguagem-padrão para manipular informações em banco de dados relacionais".

Primeiramente denominada *Strutured English Query Language* (SEQUEL), a SQL fora desenvolvida através da IBM, que fez uso dos princípios de Codd. No ano de 1979 surgiu a versão inicial comercial da linguagem SQL.

A SQL atualmente encontra-se no patamar de padrão para manipulação de dados em banco de dados. A linguagem SQL vem sendo padronizada por duas entidades, a *American National Standards Institute* (ANSI) e também a *International Standards Organization* (ISSO). Após várias modificações, em 1989 surge a versão que é utilizada pelos bancos de dados atuais. Em 1999, surge a SQL-99 que define um padrão de banco de dados objeto relacional. A mesma não é necessariamente utilizada como padrão pelos fabricantes.

Grande parte dos bancos de dados usados atualmente atingem somente até o ponto básico. Ainda que despontem inovações, a maioria dos bancos de dados ainda se utilizam da versão anterior. Entretanto, há a existência de comandos de nível médio e completo. De acordo com Oliveira, o padrão SQL-92 contempla as seguintes modalidades:

> Divide-se o padrão SQL-92 em quatro níveis: *Entry* (básico), *Transational* (em evolução), *Intermediate* (intermediário) e *Full* (completo). A maior parte dos bancos de dados utilizados atualmente atende ao nível básico. Mesmo existindo uma versão mais nova do padrão, a maior parte dos bancos de dados ainda utiliza, de forma básica, o padrão anterior. Alguns comandos, contudo, atingem os níveis intermediários e completo (Oliveira, 2002, p.18).

No modelo relacional a tabela consiste no único tipo de estrutura de dados existente, sendo possível a criação de outras tabelas com a junção ou combinação de

várias delas, assim como pesquisar, atualizar ou excluir linhas de tabelas utilizando apenas comandos SQL.

Sendo que o SQL não é considerado uma linguagem procedural, logo é possível especificar o que deve ser feito e não como deve ser feito. Assim, o comando atinge um conjunto de linhas e não cada uma das linhas somente. Dessa maneira, com o comportamento procedural não é necessário compreender o funcionamento e armazenamento físico dos dados dentro do banco de dados.

Cabe ressaltar que não é possível transferir os comandos SQL de um banco de dados para outro, entretanto, com as devidas adaptações, a maioria dos comandos poderá ser aproveitada, de acordo com o banco de dados utilizado.

O SQL pode ser utilizado de duas maneiras, interativamente ou de forma embutida:

Na maneira interativa executam-se comandos diretamente no banco de dados, sendo que a maioria deles possuem ferramentas que permitem a execução interativa. O usuário digita o comando desejado e imediatamente mostram-se os resultados após a execução. Podemos citar o *QMF for Windows*, ferramenta do banco de dados DB2.

Na segunda forma o comando SQL é embutido, em linguagens que suportam o SQL, como *Cobol, Pascal, Basic*, *C* e outras. Essas linguagens podem ser denominadas de "rotinas SQL", que enviam ou recebem parâmetros através de variáveis. Por sua vez, o banco de dados recebe esses parâmetros em comandos, executando-os e retornando a informação desejada.

De acordo com Oliveira (2002), tanto no modo interativo ou embutido a linguagem é dividida em quatro grupos de comandos:

- ***Data Definition Language*** **(DDL)** : permite a criação de comandos responsáveis pela criação de objetos do banco de dados, como tabelas, visões, índices. Dentre eles:

CREATE TABLE
ALTER TABLE
DROP TABLE
CREATE VIEW
CREATE INDEX

- ***Data Manipulation Language* (DML):** permite a execução de comandos para a manipulação dos dados como consultas e alterações dos dados armazenados no banco de dados.

  INSERT
  DELETE
  UPDATE

- ***Data Query Language* (DQL):** permite extrair dados do banco de dados.

  SELECT

- ***Data Control Language* (DCL):** são comandos que provem a segurança interna do banco de dados, atribuindo privilégios para usuários acessarem determinados objetos do banco de dados.

  GRANT
  REVOKE
  CREATE USER
  ALTER USER

Com o advento do SQL-99, a linguagem SQL passou a incorporar comandos procedurais (*BEGIN, IF, FUNCION, PROCEDURES*), ou seja, extensões da linguagem. Essas extensões são especificas de cada banco de dados, podemos citar *PL/SQL*, que por sua vez é uma linguagem procedural do *Oracle* e o *Transact-SQL*, do banco de dados *SQL Server*.

6 BANCO DE DADOS

Segundo Oliveira (2002, p. 22), “um banco de dados é um conjunto coerente e lógico de dados relacionados que possuem significância intrínseca. Esses dados representam aspectos do mundo real e devem ser mantidos para atender aos requisitos da empresa”.

Temos outra definição com Date (2000, p. 9), “um banco de dados é uma coleção de dados persistentes utilizada pelos sistemas de aplicação de uma determinada organização”.

De acordo com Oliveira (2002), há cinco tipos de banco de dados:

- **Hierárquico:** um gerenciador desse tipo representa dados como uma estrutura de arvore, composto de uma hierarquia de registro de dados.

- **Rede:** representa os dados como registros vinculados uns aos outros, formando conjuntos comuns de dados.

- **Relacional:** representa os dados como uma simples coleção de linhas e colunas em tabelas bidimensionais.

- **Objeto-relacional:** combina o modelo orientado a objetos (união de propriedades e métodos) com o modelo relacional (linhas e colunas de tabelas).

- **Objeto*:*** representa os dados e processos em um único objeto.

6.1 Sistema de Gerenciamento de Banco de dados

Um gerenciador de banco de dados é uma coleção de programas que permite criar estruturas, manter dados e gerenciar as transações efetuadas em tabelas, além de permitir a extração das informações de maneira rápida e segura.

Segundo Silberschatz (1999, p. 01), “um sistema gerenciador de banco de dados (SGBD) é constituído por um conjunto de dados associados a um conjunto de programas que fornece o acesso a esses dados”.

O principal objetivo de um SGBD é proporcionar um ambiente conveniente e igualmente eficiente para a recuperação e armazenamento das informações do banco de dados. Os sistemas de banco de dados são projetados para gerir grandes volumes de informações.

O gerenciamento de informações implica a definição das estruturas de armazenamento e a definição dos mecanismos para a manipulação dessas informações armazenadas contra eventuais problemas com o sistema, além de impedir tentativas de acesso não autorizadas. Se os dados são compartilhados por diversos usuários, o sistema deve evitar a ocorrência de resultados anômalos.

Algumas das principais características de um gerenciador de banco de dados, de acordo com Oliveira (2002):

- **Controle de redundância:** informações devem possuir um mínimo de redundância visando estabelecer à estabilidade do modelo.

- **Compartilhamento de dados:** as informações devem estar disponíveis para qualquer numero de usuários de forma concomitante e segura.

- **Controle de acesso:** necessidade de saber quem pode realizar qual função dentro do banco de dados.

- **Esquematização:** os relacionamentos devem estar armazenados no banco de dados para garantir a facilidade de entendimento e aplicação do modelo. A integridade das informações deve ser garantida pelo banco de dados.

- **Backup ou cópias de segurança:** deve haver rotinas específicas para realizar a cópia de segurança dos dados armazenados.

6.2 Banco de Dados Relacionais

Segundo Silberschatz (1999, p. 61), “um banco de dados relacional consiste em uma coleção de tabelas, cada uma das quais com um único nome. Uma linha em uma tabela representa um relacionamento entre um conjunto de valores”. Sendo uma tabela um conjunto de relacionamentos, há um estreitamento entre a abordagem tabela e o conceito matemático de relação, originando assim a nomenclatura desse modelo de dados.

O objetivo de um banco de dados relacional é armazenar um grupo de objetos em um dicionário de dados, de forma a tornar rápida e segura a manipulação das informações contidas nos mesmos. Como objetos, podemos entender tabelas, visões, índices e até mesmo procedimentos e funções que estejam armazenadas no banco de dados.

O modelo relacional contempla três aspectos:

- **Estruturas**

As estruturas são objetos bem definidos (tabelas, visões, índices e assim por diante) que armazenam ou acessam os dados de um banco de dados. Essas estruturas e as informações podem ser manipuladas por operações.

- **Operações**

As operações são ações definidas claramente que permitem aos usuários manipular os dados e as estruturas de um banco de dados. Elas devem obedecer a um conjunto predefinido de regras de integridade.

- **Regras de integridade**

As regras de integridade são as leis que governam as operações permitidas nos dados e nas estruturas de um banco de dados. Dessa maneira as mesmas protegem os dados e as estruturas de um banco de dados, garantindo integridade e segurança a estrutura.

6.2.1 Tabela

Uma tabela pode ser entendida como um conjunto de linhas e colunas. As colunas de uma tabela qualificam cada elemento (no caso, a linha com informações relacionadas ao objeto).

6.2.2 Atributo

Os atributos são as informações básicas que qualificam uma entidade e descrevem seus elementos ou características. Quando transpostos ao modelo físico (ao banco de dados), os atributos são denominados de campos ou colunas.

6.2.3 Chaves

As chaves são um conjunto de um ou mais atributos que identificam linhas e estabelecem relações entra linhas e tabelas de um banco de dados relacional.

- **Chave primária:** é representada por uma ou mais colunas que são usadas para distinguir das demais linhas de uma tabela.

- **Chave estrangeira:** é representada por uma ou mais colunas, que por sua vez, são chaves primárias de uma tabela, permitindo a implementação de relacionamentos em um banco de dados relacional.

6.2.4 Visões

Uma visão é a representação de uma ou mais tabelas, que também pode ser definida como uma “consulta armazenada”. As visões admitem as operações básicas como consultas, inserções, atualizações e exclusão, mas com as devidas restrições, ou seja, qualquer das operações executadas em uma visão, afetará as tabelas bases desta visão.

6.2.5 Seqüências

Uma seqüência gera automaticamente valores numéricos para as linhas de uma ou mais tabelas, tornando de maneira simplificada a programação do aplicativo.

6.2.6 Unidades de Programa

Funções, *triggers, procedures*, e blocos anônimos se referem à unidade de programa. Os procedimentos e funções combinam a facilidade e flexibilidade do SQL à funcionalidade de procedimento de uma linguagem de programação estruturada.

6.2.7 Índices

Os índices podem fazer referência a uma ou mais colunas e são criados independentemente dos dados. Fazem parte da lógica, portanto, podem ser excluídos e criados a qualquer momento. O seu objetivo é aumentar o desempenho da recuperação dos dados.

6.3 Normalização

Segundo Heuser (2004, p. 149), “uma forma normal é uma regra que deve ser obedecida por uma tabela para que esta seja considerada "bem projetada””. Seguem abaixo as definições de formas normais:

Primeira forma normal (1FN) - diz-se que uma tabela está na primeira forma normal, quando a mesma não contém tabelas aninhadas, ou seja, nenhum de seus atributos possuem repetições.

Segunda forma normal (2FN) - uma tabela encontra-se na segunda forma normal, quando, além de estar na 1FN, todos os atributos não chave da tabela dependem unicamente da chave, ou seja, não contém dependências parciais.

Entende-se por dependência parcial (funcional), quando uma coluna depende apenas de parte de uma chave primária composta.

Terceira forma normal (3FN) - uma tabela encontra-se na terceira forma normal, quando, além de estar na 2FN, todos os seus atributos não chave não dependem de nenhum outro atributo não chave, ou seja, não contém dependências transitivas.

Entende-se por dependências transitivas, quando uma coluna, além de depender da chave primária da tabela, depende de outra coluna ou conjunto de colunas da tabela.

Quarta forma normal (4FN) - uma tabela encontra-se na quarta forma normal, quando, além de estar na 3FN, não há repetição de dois ou mais atributos não chave, ou seja, não contem dependências multivaloradas.

Entende-se por dependências multivaloradas, quando uma coluna ou conjunto de colunas depende multivaloradamente de uma coluna (determinante) da mesma tabela quando um valor do atributo determinante identifica repetidas vezes um conjunto de valores da coluna dependente.

Quinta forma normal (5FN) - é muito raro ocorrer, sendo que é utilizada quando uma tabela na quarta forma normal pode ser dividida em duas ou mais tabelas, para evitar redundâncias existentes.

6.4 Modelagem de Dados

Segundo Oliveira (2002, p. 25), “a abordagem relacional é a utilização de conceitos de entidade e relacionamento para criar as estruturas que irão compor o banco de dados, partindo sempre da necessidade do usuário ou grupo de usuários do sistema”.

Dessa maneira a modelagem de dados tem como propósito representar as informações do negócio, desenvolvendo um modelo contendo entidades e relacionamentos. De acordo com Pinheiro (2005, p. 36), “no desenvolvimento de aplicações em banco de dados, o modelo relacional é o mais utilizado para a representação e entendimento dos dados que compõem a essência de um sistema de informações”.

O modelo conceitual e lógico é descrito por Heuser (2004, p. 6,7) da seguinte maneira:

Modelo conceitual: “o modelo conceitual consiste em uma descrição do banco de dados de forma independente de implementação em um SGBD. O modelo conceitual registra que dados podem aparecer no banco de dados, mas não registra como os mesmos estão armazenados a nível de SGBD”, onde faz se uso da técnica de abordagem entidade-relacionamento, (E-R) que será abordado posteriormente.

Modelo lógico: “o modelo lógico é uma descrição de um banco de dados no nível de abstração visto pelo usuário do SGBD. Assim, o modelo lógico é dependente do tipo particular de SGBD que está sendo utilizado”.

Modelo físico: o modelo físico é usado por profissionais que procuram otimizar o desempenho não envolvendo mudanças no aspecto funcional. As linguagens e notações não são padronizadas e variam de acordo com cada SGBD.

A técnica mais difundida de modelagem é a abordagem entidade-relacionamento, onde é representada pelo diagrama de entidade-relacionamento. Essa técnica criada em 1976 por Peter Chen é considerada um padrão de modelagem e foi utilizado de modelo para outras técnicas como a orientada a objetos *Unified Modeling Language* (UML).

De acordo com Heuser (2004, p. 14), uma entidade pode ser definida como “um conjunto de objetos da realidade modelada sobre os quais se deseja manter informações no banco de dados”. Segundo Pinheiro (2005, p. 36), “as entidades não estão soltas, isoladas uma das outras, mas relacionadas de forma a mostrar a realidade em um contexto lógico”.

Relacionamento é definido por Heuser (2004, p. 15) como um conjunto de associações entre ocorrências de entidades. É similar ao relacionamento de orientação a objetos, onde obrigatoriedade e cardinalidade fazem parte de suas características.

As vantagens na utilização do Modelo de Entidade x Relacionamento de acordo com Oliveira (2002, p. 26):

- Sintaxe robusta: o modelo documenta as necessidades de informação da empresa de maneira precisa e clara.

- Comunicação com usuário: os usuários podem, com pouco esforço entender o modelo.
- Facilidade de criação: os analistas´podem criar e manter um modelo facilmente.

- Integração com varias aplicações: diversos projetos podem ser inter-relacionados utilizando-se o modelo de dados de cada um deles.

- Utilização universal: o modelo não está vinculado a um banco de dados especifico, mas sim ao modelo da empresa, o que garante a sua interdependência de implementação.

De acordo com Pinheiro,

> No ambiente de banco de dados implementamos as entidades como tabelas, suas instâncias como registros e atributos como colunas. Os relacionamentos são mapeados em regras de integridade implementadas através de *constraints* de chaves estrangeira no banco de dados. A integridade referencial deve garantir que, quando um registro se relaciona, este deve ter uma referência válida para outro registro em outra tabela. Esta referência é inserida por cópias de colunas implementadas na tabela cujo único objetivo é referenciar a chave primária de outro registro em outra tabela (Pinheiro, 2005, p.37).

6.5 Técnicas de mapeamento

Sabe-se que o modelo orientado a objetos e o modelo relacional apresentam diferenças de paradigma, porém são os mais utilizados em projetos de software. Observa-se uma semelhança entre o diagrama de entidade-relacionamento e o diagrama de classes da UML, onde uma classe com seus atributos é mapeada por uma tabela do banco de dados e seus atributos, ou seja, as colunas. Contudo, o modelo relacional não fornece um suporte relacionado à herança, que é utilizado no modelo orientado a objetos.

Outra diferença é a relação entre os objetos e entidades, onde no modelo relacional é utilizado chaves estrangeiras para a as associações entre as tabelas, nesse caso um campo referência outra tabela, já os objetos fazem referência a coleções de outros objetos, mas através de apenas um atributo.

Tendo conhecimento que os bancos de dados relacionais por sua natureza não suportam hierarquia, há três técnicas para mapear uma estrutura hierárquica de herança de classes para um banco de dados relacional: uma tabela por hierarquia completa de classes, uma tabela por classe concreta e uma tabela para cada classe concreta.

Tabela única para toda a Hierarquia

Uma tabela única para toda a hierarquia pode ser desenvolvida segundo Pinheiro, da seguinte maneira:

> Nesta técnica todos os atributos de todas as classes da estrutura hierárquica são mapeados para uma única tabela. Cada linha é um objeto de uma subclasse específica, diferenciada na tabela através de uma coluna extra que indica a qual classe se refere. Esta coluna extra, denominada código de tipo, pode implementar o polimorfismo se vislumbramos a possibilidade de acrescentar códigos que representam combinações de classes (Pinheiro, 2005, p.38).

Como vantagem tem-se um bom desempenho, pois todas as informações estão em uma única tabela, proporcionando assim simplicidade uma vez que para se adicionar classes é necessário acrescentar colunas para as novas informações. As desvantagens são maior acoplamento entre as classes da hierarquia onde há necessidade de uma coluna identificar a classe de cada registro e em caso de mudança de uma classe em determinada tabela pode-se afetar outras classes e com a ausência de normalização dos dados ferindo as regras da teoria de modelagem. Devido as colunas com valores nulos, ou seja, não preenchidas em parte das linhas da tabela, havendo um desperdício de espaço no banco de dados. Além disso, a tabela pode sofrer um grande aumento devido às hierarquias, sendo esta abordagem aconselhável a pequenas hierarquias.

Uma tabela para cada classe concreta

Diferentemente das classes definidas como abstratas as classes concretas podem ter objetos instanciados. Essa abordagem define uma tabela para cada classe concreta não sendo necessário um identificador de classe na tabela. Além disso, a tabela implementa atributos herdados da superclasse assim como os atributos da própria classe. Tem como vantagem melhor performance em relação ao acesso aos dados, pois todas as informações de uma tabela se encontram em uma única classe. Porém a redundância dos atributos, pois o atributo é replicado em todas as classes de cada subclasse, havendo maior manutenção da integridade da informação. Essa abordagem é indicada para aplicações que raramente sofrem mudanças.

Uma tabela para cada classe

É definido um relacionamento de um-para-um entre todas as tabelas, onde classes são tabelas e atributos são colunas. Nessa modalidade, a estrutura de tabelas ficam semelhantes a hierarquia de classes no modelo UML, devido a normalização dos dados. Tem como vantagem uma manutenção simples, já que para a inclusão de novas classes criam-se também novas tabelas, sendo que o aumento do banco de dados é proporcional ao número de objetos persistidos. Tendo em vista o aumento de tabelas no banco de dados, se torna mais complexo as rotinas de busca e gravação de objetos obtendo um menor desempenho.Há também como ponto negativo a qualidade de junções (*joins*) entre tabelas. Essa abordagem é apropriada em casos de mudanças continuas.

	Uma tabela por hierarquia de classes	**Uma tabela por classe concreta**	**Uma tabela por classe**
Ad-hoc reporting	Simples	Médio	Médio/Difícil
Facilidade de implementação	Simples	Médio	Difícil
Facilidade de acesso	Simples	Simples	Médio/Simples
Acoplamento	Muito alto	Alto	Baixo
Velocidade de acesso	Rápido	Rápido	Médio/Rápido
Suporte a polimorfismos	Médio	Baixo	Alto

Tabela 1. Comparativo entre técnicas de mapeamento de classes (Júnior, 2003).

6.5.1 Mapeando objetos para Tabelas

Existem diversas técnicas de mapeamento de objetos, onde cada uma apresenta vantagens e desvantagens sobre as outras. Sendo assim, o desenvolvedor, a partir da escolha do mecanismo de persistência, organiza a estrutura de tabelas em um banco de dados para suportar o esquema de objetos modelado.

6.5.2 Mapeando atributos

Os atributos de um objeto são mapeados em colunas da tabela, onde não necessariamente um atributo deve ter uma coluna que o referencia em uma tabela. Podemos ter um atributo que pode ser obtido através de consultas, ou um atributo mapeado para mais de uma coluna, e vice-versa como vários atributos mapeados para somente uma coluna da tabela. Vale a pena ressaltar que o mapeamento deve considerar fatores, como a tipagem dos dados e comprimento máximo dos campos.

6.5.3 Mapeamento de Relacionamentos

O mapeamento de relacionamentos na associação entre objetos é uma modalidade que tem por característica a simplicidade. O relacionamento pode ser dividido em três tipos:

Um-para-um: onde se faz uso da chave estrangeira (*foreign key*), relacionando duas tabelas.

Um-para-muitos: funciona da mesma forma do relacionamento anterior, utilizando chaves, porém a chave é colocada na tabela que possui objetos múltiplos, ou seja, o lado “muitos” do relacionamento.

Muitos-para-muitos: são modeladas por uma terceira tabela, que armazena as chaves, relacionando as duas tabelas.

De acordo com Pinheiro,

> Na metodologia orientada a objetos, além da hierarquia, existem mais três tipos de relacionamentos entre objetos: associação, agregação e composição. Relacionamentos na tecnologia de orientação a objetos são implementados através de referências a objetos e correspondentes operações de manipulação (Pinheiro, 2005, p.40).

São apresentadas abaixo as classificações de relacionamento de objetos que influenciam o mapeamento objeto-relacional.

Cardinalidade do relacionamento entre objetos

- **Relacionamento um-para-um:** refere-se ao relacionamento onde a cardinalidade de cada um dos componentes é no máximo um.

- **Relacionamento um-para-muitos:** refere-se ao relacionamento onde a cardinalidade de um objeto é no máximo um e do outro objeto pode ser um ou mais.

- **Relacionamento muitos-para-muitos:** refere-se ao relacionamento onde a cardinalidade de cada um dos componentes é um ou mais.

Direcionamento do relacionamento de objetos

- **Relacionamentos unilaterais:** este relacionamento existe quando o objeto tem a referencia de outro objeto, mas o mesmo não contém a referencia do primeiro.

- **Relacionamentos bidirecionais:** este relacionamento existe quando os objetos se referenciam.

No mundo relacional existe apenas o relacionamento unilateral, sendo assim uma tabela que contem uma ou mais colunas que faz referencia a chave primária de outra tabela.

Deve-se escolher uma tabela para implementar a chave estrangeira em caso de cardinalidade um para um. Já na cardinalidade um-para-muitos, a chave deve ser implementada na tabela de extremidade muitos. Sendo que na cardinalidade muitos-para-muitos, é criada uma nova tabela chamada de associativa, que contém as associações entre as tabelas, ou seja, as duas chaves estrangeiras correspondentes às chaves primarias das tabelas envolvidas.

6.6 Banco de dados orientado a objetos

Nos últimos anos, as aplicações de banco de dados não tem se encaixado nas soluções propostas pelo modelo relacional e o modelo entidade relacionamento. Dessa maneira o modelo orientado a objetos nasceu para atender as exigências de novas aplicações.

Segundo Silberschatz (1999, p. 269), o modelo de dados orientado a objeto é uma adaptação a sistemas de banco de dados do paradigma de programação orientado a objeto. Está baseado no conceito de encapsular os dados em um objeto e o código opera nesses dados.

Podemos citar como características essenciais do banco de dados orientado a objetos:

Objeto: corresponde a uma entidade do modelo E-R. Segundo Silberschatz (1999, p. 251), o objeto tem associado a ele:

- Um conjunto de variáveis que contem os dados para o objeto; as variáveis correspondem aos atributos no modelo E-R.

- Um conjunto de mensagens ao qual o objeto responde; cada mensagem pode ter zero, um ou mais parâmetros.

- Um conjunto de métodos cada qual sendo um corpo de código para implementar a mensagem: um método retorna um valor como resposta à mensagem.

Classes de objeto: existem muitas entidades similares em um banco de dados, ou seja, usam os mesmos métodos e possuem variáveis de mesmo nome e tipo. Exceto nos valores designados as variáveis, normalmente agrupa-se objetos similares formando uma classe, compartilhando assim uma definição comum.

Herança: no esquema de banco de dados orientado a objeto, normalmente temos muitas classes similares, entretanto temos classes que fazem uso de variáveis especificas. Surge o conceito de herança que permite representar as similaridades entre as classes, colocando essas classes em uma hierarquia de especialização. A abordagem de uma hierarquia de classe é correspondente ao conceito de especialização no modelo entidade-relacionamento.

6.6.1 Linguagem orientada a objeto

A linguagem orientada a objeto podem ser classificadas de duas formas:

Na sua primeira forma são utilizados conceitos de orientação a objeto no projeto e codificados no banco de dados relacional. São usados modelos de entidade-relacionamento e posteriormente convertidos em conjuntos de relações manualmente.

Outra opção é incorporar os conceitos de orientação a objeto na linguagem de banco de dados. Assim temos mais duas opções:

- Uma opção é estender a linguagem de manipulação, o SQL, são os denominados sistemas relacionais objeto.

- A outra maneira é estender a linguagem orientada a objetos, são denominadas linguagens de programação persistente.

6.6.2 Linguagens de programação persistentes

Ao contrário das linguagens de programação tradicionais, as linguagens de programação persistentes manipulam diretamente os dados persistentes.

Segundo Silberschatz (1999, p. 261), uma linguagem de programação persistente é uma linguagem de programação estendida com estruturas para tratar dados persistentes. Linguagens de programação persistentes podem ser diferenciadas de linguagens com SQL embutida em duas formas:

Na linguagem embutida, a linguagem *host* é diferente da linguagem de manipulação, sendo o programador responsável por qualquer tipo de conversão entre as linguagens.

Já na linguagem de programação persistente, a linguagem de manipulação é totalmente integrada com a linguagem *host* e ambas compartilham o mesmo sistema. Os objetos podem ser criados e armazenados no banco de dados sem qualquer tipo explicito ou mudanças de formato.

Na linguagem embutida, o programador deve explicitar no código para efetuar mudanças de dados do banco de dados para a memória e no caso de atualização, armazenar esse dado para dentro do banco.

Já na linguagem de programação persistente, o programador não precisa explicitar no código para efetuar manipulações de dados persistentes tanto como buscar na memória como armazenar no banco.

Vale a pena ressaltar com o uso das linguagens de programação torna-se fácil cometer erros que por sua vez danifiquem o banco de dados. Sendo de extrema importância um suporte a linguagem declarativa, porem atualmente não é bem aceita pelas linguagens de programação persistentes.

6.6.3 Persistência de objetos

Persistência por classe

Abordagem simples, ao se declarar a classe persistente, todos seus objetos são persistentes. Caso contrário, objetos de classes não persistentes são transientes. Com isso, pode ser considerada uma abordagem não flexível.

Persistência por criação

Nessa abordagem é introduzida uma extensão para a criação de objetos transientes. De acordo com a maneira que os objetos são criados os mesmos podem ser persistentes ou transientes.

Persistência por marcação

Nessa abordagem todos os objetos são criados como transientes, e marcado explicitamente como persistente antes do término do programa. Assim, após a criação do objeto é decidido se o mesmo vai ser transiente ou persistente.

Persistência por referência

Nessa abordagem, são declarados objetos como persistentes (raízes) e todos os objetos referenciados pela raiz, também são persistentes, formando estruturas de dados persistentes.

6.7 Bancos de dados relacionais-objeto

O modelo relacional aninhado é uma extensão do modelo relacional em que os domínios podem ser definidos como atômicos ou como relações. O conjunto de inteiros é um domínio atômico, mas o conjunto de todos os conjuntos de inteiros é um domínio não atômico.

Relações aninhadas são apenas um exemplo de extensões ao modelo relacional básico. Outros tipos de dados não atômicos, como registros aninhados também têm se mostrado úteis. O modelo relacional é aninhado, violando a 1NF, para permitir tipos não atômicos.

Sistemas relacionais-objeto combinam dados complexos baseados em um modelo relacional aninhado com conceitos orientado a objeto como classe de objeto e herança. O modelo de dados orientado a objeto tem provocado uma necessidade por características como herança e referências a objetos. Sistemas com os tipos complexos e orientação a objetos permitem que os conceitos do modelo E-R, como identidade de entidades, atributos multivalorados, generalizações e especializações sejam representadas diretamente sem uma tradução complexa para o modelo relacional. O SQL e outras

linguagens de consulta tem sido estendidas para tratar tipos complexos e orientação a objetos.

Tanto os bancos de dados orientados a objeto construídos com base em linguagens persistentes, como os bancos de dados relacionais-objeto construídos a partir do modelo relacional, é de responsabilidade do projetista de banco de dados escolher o tipo apropriado as necessidades da aplicação, além disso, são direcionados a diferentes mercados. O SQL fornece boa proteção de dados contra erros de programação e facilita as otimizações de alto nível, promete desempenho para aplicações que executam na memória principal e que realizam grande número de acesso ao banco de dados. As linguagens de programação persistentes visam as aplicações que exigem grande desempenho, entretanto susceptíveis a erros de programação e sem poder de consulta. Já os sistemas relacionais objeto visam tornar a modelagem de dados e a consulta mais fácil pelo uso de dados complexos, incluindo dados multimídia.

Segundo Silberschatz (1999, p. 287), para efeitos de comparação:

	Tipos de dados	**Linguagens de consulta**	**Característica**
Sistemas relacionais	Simples	Poderosa	Alta proteção
Linguagem de programação simples baseada em BDOOs	Complexo	Integração com linguagem de programação	Alto desempenho
Sistemas relacionais-objeto	Complexo	Poderosa	Alta proteção

Tabela 2: Comparativo entre sistemas de bancos de dados (Silberschatz, 1999).

7 HIBERNATE

Hibernate é uma ferramenta de mapeamento objeto/relacional completa que provê todos os benefícios do MOR. Selecionando os vários módulos do software de *Hibernate* e combiná-los dependendo do tipo de projeto técnico e de exigências do negócio.

De acordo com Fernandes e Lima, 2007.

> O *Hibernate* é um framework de mapeamento objeto relacional para aplicações Java, ou seja, é uma ferramenta para mapear classes Java em tabelas do banco de dados e vice-versa. É bastante poderoso e dá suporte ao mapeamento de associações entre objetos, herança,

> polimorfismo, composição e coleções. O *Hibernate* não apresenta apenas a função de realizar o mapeamento objeto relacional. Também disponibiliza um poderoso mecanismo de consulta de dados, permitindo uma redução considerável no tempo de desenvolvimento da aplicação (Fernandes e Lima, 2007, sem p).

Hibernate Core

O *Hibernate Core*, também conhecido como *Hibernate* 3.2.x, ou *Hibernate*, é o serviço base para a persistência com seu *Application Programming Interface* (API) nativo e seu mapeamento de dados em arquivos XML. Possui linguagem *Hibernate Query Language* (HQL) que se caracteriza por ser similar ao SQL. O *Hibernate Core* serviu de base para outros módulos, podendo ser utilizado de modo isolado, sendo independente de outros *frameworks* e ambientes de desenvolvimento como *Java 2 Enterprise Edition* (J2EE), aplicações *Swing*, dentre outras. A aplicação fará uso de um mapeamento escrito em arquivos XML O módulo Core é responsável pela geração do código SQL, encapsulando as operações JDBC, eliminando a necessidade de escritas de sentenças SQL e de manipulações de *ResultSets* JDBC, com exceção de consultas complexas ou dependentes de recursos nativos do banco de dados.

Hibernate Annotations

Uma nova maneira de se construir um mapeamento tornou-se possível com o Java Development Kit 5.0 (JDK), se utilizando de anotações diretamente no código de fonte de Java. Com o pacote *Hibernate Annotations* trabalhando acima do pacote *Hibernate Core* pode-se substituir ou adicionar as anotações ao mapeamento XML. Empregando o *Hibernate Core* e o H*ibernate Annotations* para reduzir linhas de código no mapeamento objeto/relacional, em comparação com os arquivos XML.

Hibernate EntityManager

É responsável pelo controle do ciclo de vida e a persistência dos objetos. O *Hibernate EntityManager*, cujo é outro módulo opcional que caracteriza-se em um pequeno envoltório em torno no *Hibernate Core*, fornecendo compatibilidade de JPA e

implementando as operações para inserção, atualização, remoção e consulta de objetos anotados.

7.1 Incompatibilidade de paradigma

De acordo com Bauer e King (2005, p. 8), “sistemas de gerenciamento de bancos de dados relacionais são a única tecnologia de gerenciamento de dados provada e quase sempre são uma exigência em qualquer projeto *Java*”.

Os paradigmas modelo de objeto e modelo relacional são alvos de discussão nos últimos anos devido o problema de incompatibilidade de paradigma. Esse problema comumente é gerador de preocupação e esforço adicional em qualquer projeto no ambiente coorporativo.

Segundo Bauer e King (2005, p. 8), “em um aplicativo orientado para objeto, a persistência permite que o objeto sobreviva ao processo que o criou. O estado do objeto pode ser armazenado em um disco e um objeto com o mesmo estado pode ser recriado em algum ponto no futuro”.

Uma solução ao problema de incompatibilidade que vem ganhando bastante aceitação nos últimos anos e já citado anteriormente é o mapeamento objeto relacional. O MOR realiza a persistência automática (transparente) de objetos de uma aplicação orientada a objetos para tabelas de um banco de dados relacional, por meio de metadados (descrevem o mapeamento entre o modelo de objetos e o modelo relacional). Dessa maneira ao invés de obter os dados dos objetos e combina-los a uma *string* de consulta SQL que será enviada ao banco de dados relacional, o desenvolvedor apenas invoca os métodos responsáveis por salvar, excluir, alterar os objetos e o *framework Hibernate* transforma os objetos e seus atributos em tabelas e colunas.

Problema de granularidade

Segundo Bauer e King (2005, p. 12), “a granularidade refere-se ao tamanho relativo dos objetos”. Objetos persistentes podem ter diferentes tipos de granularidade para tabelas e colunas de granularidade inerentemente limitada.

No modelo de objetos é possível alcançar vários níveis de granularidade, ao contrario do modelo relacional que possuem apenas dois níveis, tabelas e colunas.

Tendo como solução a persistência de objetos de baixa granularidade em tabelas de grande granularidade ou vice-versa. Em outras palavras, uma tabela armazena diversos tipos de objetos ou um objeto é armazenado em diversas tabelas. Porém Bauer e King (2005) propõem como objetivo do *Hibernate* suportar modelos de objeto de granularidade fina, ou seja, "mais classes que tabelas". Portanto esse modelo oferece maior coesão e reutilização de código implementando segurança de tipo e comportamento.

Problema de subtipos

Conforme Bauer e King (2005, p. 15), "a incompatibilidade de subtipos é uma na qual a estrutura de herança em seu modelo de *Java* deve ser persistida em um banco de dados SQL que não oferece uma estratégia de herança". As primeiras soluções MOR não suportavam esse tipo de mapeamento. As diferentes maneiras de mapeamento de herança são apresentadas como solução a esse problema e foram abordadas no capitulo anterior.

Problema de identidade

O problema de identidade acontece quando temos dois objetos e os mesmos forem idênticos. No caso o *Java* propõe duas soluções para o problema, identidade e igualdade de objeto, já o banco de dados relacional uma solução, a chave primária.

Bauer e King (2005, p. 116), descrevem os três métodos para identificação de objetos:

Identidade de objeto: objetos são idênticos se ocuparem o mesmo local na memória na *Java Virtual Machine* (JVM). Podendo ser verificada utilizando o operador == .

Igualdade de objeto: objetos são iguais se possuem o mesmo valor (igualdade de valor), como definido pelo método *equals (Object o)*.

Identidade do banco de dados: objetos armazenados em um banco de dados relacional são idênticos se representarem a mesma linha ou, equivalentemente, compartilharem a mesma tabela e valor de chave primária.

É proposta como solução o uso de identificadores de banco de dados, ou seja, o próprio *Hibernate* trataria a identidade do banco de dados internamente, essa solução é fácil e rápida, porém é preciso escolher uma boa chave primária.

Pode ser considerada uma boa chave primária, aquela que a coluna ou colunas, possuírem sempre valores preenchidos (nunca ser nula), cada linha possui um único valor (única) e o valor dessa linha nunca sofrerá mudanças (constante). Na tabela abaixo são apresentados vários mecanismos internos do *Hibernate* para a geração de chaves primárias:

Mecanismo	Descrição
Identidy	Mapeado para colunas *identity* no DB2, MySQL, MSSQL, Sybase, HSQLDM, Informix.
Sequence	Mapeado em sequências no DB2, PostgreSQL, Oracle, SAP DB, Firebird (ou *generator* no Interbase).
Increment	Lê o valor máximo da chave primária e incrementa um. Deve ser usado quando a aplicação é a única a acessar o banco e de forma não concorrente.
Hilo	Usa algoritmo *high/low* para geração de chaves únicas
uudi.hex	Usa uma combinação do IP com um *timestamp* para gerar um identificador único na rede.

Tabela 3: Mecanismo de geração de chaves primárias (Júnior, 2003).

Problema de associação

A associação é referente a relação de uma entidade. Na linguagem orientada a objetos a associação é representada por meio de referencia, e no modelo relacional a chave estrangeira representa a associação.

No caso do modelo de objeto a associação é navegável em ambas as direções, ou seja, possuem um relacionamento bidirecional. Já o modelo relacional desconhece a

navegação, portanto possuem um relacionamento unilateral. São propostas como soluções:

Associações um-para-um: uso de chave estrangeira.

Associações muitos-para-muitos: uso de uma tabela link, ou seja, uma terceira tabela.

Associações um-para-muitos: uso de chave, porém do lado muitos.

Cabe ressaltar que esse conceito já foi abordado e detalhado no capítulo anterior.

Problema de navegação de gráfico de objeto

Segundo Bauer e King (2005, p. 20), "a coisa mais importante para melhorar desempenho de acesso a dados é minimizar o número de pedidos ao banco de dados. O modo mais óbvio para fazer isso é minimizar o número de consultas de SQL". O *Hibernate* oferece como solução quatro estratégias de busca: busca imediata, busca preguiçosa, busca ávida e busca em lotes. Que por sua vez, serão abordados posteriormente.

Estima-se que 30 por cento é gasto de esforço e tempo para codificar o SQL/JDBC de um aplicativo *Java*. Apesar de todo esse esforço, o resultado não parece ser o melhor. Conforme Lemos 2007, os principais problemas que geram o custo da incompatibilidade:

- É necessário escrever muito código para (tentar) contornar o problema;
- Código se torna repetitivo e de difícil manutenção;
- A escrita de código SQL pode tornar a aplicação dependente do banco de dados;
- Modelagem dos objetos fica prejudicada;
- Outras camadas ficam fortemente acopladas à camada de persistência;
- Produtividade pode ser fortemente afetada.

7.2 Mapeamento Objeto-relacional

De acordo com Bauer e King,

> A evolução dos sistemas de hoje em verdadeiro sistemas de banco de dados relacional com integração orientada para objeto sem costura permanece pura especulação [...]. MOR é atualmente a melhor solução disponível e é um salvador de tempo para os desenvolvedores que enfrentam a incompatibilidade de objeto/relacional diariamente (Bauer e King, 2005, p.40).

A solução MOR consiste em quatro partes de acordo com Bauer e King (2005, p. 32):

- Um API por executar operações *Create, Read, Update, Delete* (CRUD) básicas em objetos de classes persistentes;
- A linguagem ou API para especificar consultas que recorrem a classes e propriedades de classes;
- Uma facilidade para especificar mapeamento de metadados;
- Uma técnica para a implementação de MOR para interagir com objetos transacionais a fim de executar verificação suja (*dirty*), ir buscar associações lentas, e outras funções de otimização.

Transparência é o que caracteriza a qualidade de implementação da camada de persistência como silenciosa, pois o desenvolvedor realiza operações relacionadas a manipulação dos dados, sem perceber que o framework *Hibernate* está por trás dessas manipulações. Além disso, proporciona um código mais limpo, tratando apenas de funções pertinentes ao seu escopo. Os níveis de qualidade do MOR são divididos em quatro modalidades (BAUER E KING *apud* FUSSEL).

Puro relacional

Todo o aplicativo, inclusive a interface do usuário é usado o conceito de modelo relacional. Podendo ser útil para aplicações simples, pois os processos de negócio ficam a cargo do banco de dados. Porém essa abordagem apresenta deficiências de portabilidade e manutenção.

Mapeamento de objeto leve

Neste caso as entidades são representadas como classes e mapeadas manualmente para tabelas do banco de dados relacional. Essa abordagem é bastante difundida, porém é bem sucedida em aplicações com um pequeno número de entidades.

Mapeamento de objeto médio

Os aplicativos fazem uso do modelo de objetos, o SQL é realizado em tempo de construção utilizando ferramentas que geram código, as consultas são efetivadas usando uma linguagem OO. Essa abordagem é adotada por aplicações de nível médio e oferecendo portabilidade, pois não se encontra atrelada ao banco de dados.

Mapeamento de objeto completo

Esse tipo se mapeamento atende os conceitos de herança, polimorfismo e composição, oferecendo um mapeamento transparente.

Fazendo uso da implementação do mapeamento objeto/relacional, além da redução de esforço na construção de aplicações, se faz necessário uma documentação padrão e compreensível a outras tecnologias. Os desenvolvedores são impostos a documentar o modelo de objetos e o modelo relacional de forma acessível. Conforme Bauer e King (2005), o mapeamento objeto/relacional e o *Hibernate* apresentam como principais vantagens:

Produtividade

O *Hibernate* elimina grande parte do trabalho relacionado a persistência, permitindo que o desenvolvedor se concentre na implementação da lógica da aplicação.

Manutenção

Com menos código, é enfatizado a lógica de negócio promovendo um maior entendimento do sistema. Quando o sistema faz uso da persistência manual há sempre uma tensão, pois se o modelo de objeto sofrer alguma mudança, o modelo relacional será impactado ou vice-versa. De acordo com Bauer e King (2005, p. 37), "MOR funciona como um pára-choque entre os dois modelos: permite o uso mais elegante de orientação de objeto no lado de *Java* e isola cada modelo de mudanças menores para o outro".

Desempenho

Tarefas manuais nem sempre tem melhor desempenho do que as tarefas automatizadas, considerando limitações de tempo e custo. Além disso, com o potencial aumento da produtividade o desenvolvedor poderá gastar mais tempo em possíveis gargalos.

Interdependência de fabricante

De acordo com Bauer e King (2005, p. 38), "interdependência de banco de dados ajuda nos cenários de desenvolvimento onde desenvolvedores usam um banco de dados local de peso leve, mas desdobram para produção em um banco de dados diferente".

Assegura-se quando a base de dados não possui uma modelagem e normalizações adequadas, sente-se uma dificuldade de reduzir as diferenças entre os dois mundos, o orientado a objetos e o relacional. Isso acontece principalmente em bancos legados onde não existia uma modelagem e a base de dados era utilizada em sistemas com uso de linguagens estruturadas.

Com a idéia do *Hibernate* de encapsular todo o código SQL, alguns desenvolvedores são se sentem confortáveis com o não controle de suas implementações relacionadas a persistência. Apesar do *Hibernate* oferecer o HQL como um recurso poderoso, em alguns casos é preciso recorrer ao SQL nativo devido ao desempenho ou *queries* complexas.

Pode ser considerado como um ponto negativo a inclusão de diversos *jars* a aplicação, porém uma vez efetivada sua configuração não é necessário reconfigurá-lo. Abaixo estão listadas os principais *jars* utilizados:

- *ehcache-1.1.jar*
- *jta.jar*
- *xml-apis.jar*
- *commons-logging-1.0.4.jar*
- *c3p0-0.8.5.2.jar*
- *asm-attrs.jar*
- *log4j-1.2.9.jar*
- *dom4j-1.6.jar*
- *antlr-2.7.5H3.jar*
- *cglib-2.1.jar*
- *asm.jar*
- *jdbc2_0-stdext.jar*
- *xerces-2.6.2.jar*
- *commons-collections-2.1.1.jar*
- *ejb3-persistence.jar*
- *hibernate-annotations.jar*
- *lucene-1.4.3.jar*
- *hibernate3.jar*

7.3 Interfaces

Interfaces de programação são conceitos interessantes no *Hibernate*. Conforme Bauer e King (2005, p. 50), “o principal objetivo do projeto da API é manter as interfaces entre componentes de software em espaço mais estreito possível”. Sendo que APIs MOR não são tão pequenas, as interfaces do *Hibernate* podem ser vista na figura 2:

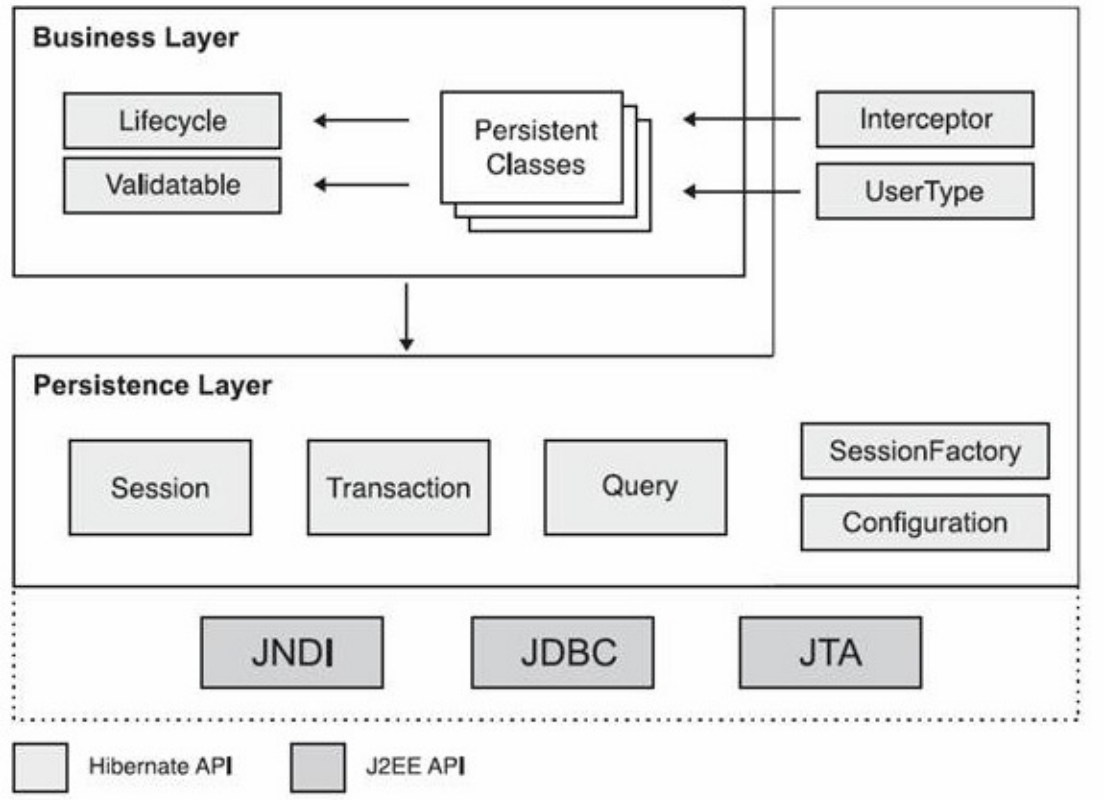

Figura 2: Arquitetura do *Hibernate* (Bauer e King, 2005)

Interfaces do *Hibernate* podem ser classificadas:

Interfaces chamadas por aplicativos para executar CRUD básico e operações de consulta. Essas interfaces são o ponto principal de dependência do negócio do aplicativo/lógica de controle do *Hibernate*. São inclusas Sessão (*Session*), Transação (*Transaction*) e Consulta (*Query*).

- *Session* é uma interface primária, fácil de criar, remover, atualizar e recuperar objetos persistentes. Possibilita a comunicação entre a aplicação e a persistência através de conexão JDBC.
- *SessionFactory* é a fabrica de sessões, criado na inicialização do aplicativo, mantém o mapeamento objeto relacional na memória e armazena declarações SQL. Normalmente existe um único *SessionFactory* para o aplicativo, no entanto se esse aplicativo acessar mais de um banco de dados deverá ser criado outro *SessionFactory*.

- *Transation* é uma interface opcional, se não utilizado o gerenciamento de transações fica dentro do código.
- *Query* permite controlar e executar consultas tanto HQL quanto SQL nativo.

Interfaces chamadas por código de infra-estrutura do aplicativo para configurar o *Hibernate*, o mais importante é a classe de Configuração. A interface de configuração é utilizada para especificar o local e propriedades do *Hibernate*, para ser criado o *SessionFactory*. Uma vez criado o *SessionFactory* o arquivo de configuração pode ser descartado. Vale a pena ressaltar que o local do arquivo de mapeamento é relativo a raiz do *classpath* do aplicativo.

- Interfaces *Callback* permitem ao aplicativo reagir a eventos que aconteçam dentro do *Hibernate*, como Interceptor, *Lifecycle* e *Validatable*.

Estas interfaces permitem extensão da funcionalidade de mapeamento poderoso do *Hibernate*, como *userType*, *CompositeUserType* e *IdentifierGenerator*. Estas interfaces suportam "definição de usuário" permitem somar seus próprios tipos e são implementadas por meio do código de infra-estrutura do aplicativo (se necessário).

De acordo com Bauer e King,

> Um fundamento e elemento muito poderoso da arquitetura é a noção do *Hibernate* de um Tipo (*type*). Um *Hibernate tipo objeto* mapeia um tipo *Java* para um tipo de coluna de banco de dados (de fato, o tipo pode atravessar colunas múltiplas). Todas as propriedades persistentes de classes persistentes, inclusive associações, têm um tipo de *Hibernate* correspondente. Este projeto torna o *Hibernate* extremamente flexível e extensível (Bauer e King, 2005, p.54).

Como o *Hibernate* faz uso de APIs do Java. O JDBC apresenta um nível elementar de abstração de funcionalidade relacionado a banco de dados, dessa maneira qualquer banco que possui *driver* JDBC é suportado pelo *Hibernate*. Em relação aos aplicativos de servidores J2EE são integrados por meio do Java Naming and Directory Interface (JNDI) e Java Transaction API (JTA).

Interfaces de extensão

De acordo com Bauer e King (2005, p. 55), o *Hibernate* oferece interfaces de extensão quando as estratégias embutidas já não forem suficientes.

- Geração de chave primária (interface *IdentifierGenerator*);
- Suporte dialeto de SQL (classe abstrata de *Dialect*);
- Estratégias de armazenamento (interfaces Cachê e *CacheProvider*);
- Gerenciamento de conexão JDBC (interface *ConnectionProvider*);
- Gerenciamento de transação (interfaces *TransactionFactory*, *Transaction* e *TransactionManagerLookup*);
- Estratégias de MOR (interface hierárquica *ClassPersister*);
- Estratégias de acesso de propriedade (interface *PropertyAccessor*);
- Criação de representação (interface *ProxyFactory*).

As aplicações precisam ter conhecimento dos estados dos objetos no seu ciclo de vida da persistência, pois sempre que propaga o estado de um objeto que está em memória para o banco de dados ou vice versa, a aplicação se comunica com a camada de persistência, fazendo uso do gerenciador de persistência e as interfaces de consulta do *Hibernate*. Conforme Fernandes e Lima 2007, “em aplicações orientadas a objetos, a persistência permite que um objeto continue a existir mesmo após a destruição do processo que o criou. Na verdade, o que continua a existir é seu estado, já que pode ser armazenado em disco e então, no futuro, ser recriado em um novo objeto”.

No ciclo de vida de persistência, o objeto pode transacionar de um objeto transiente para um objeto persistente para um objeto *detached*, como pode ser visto na figura 3:

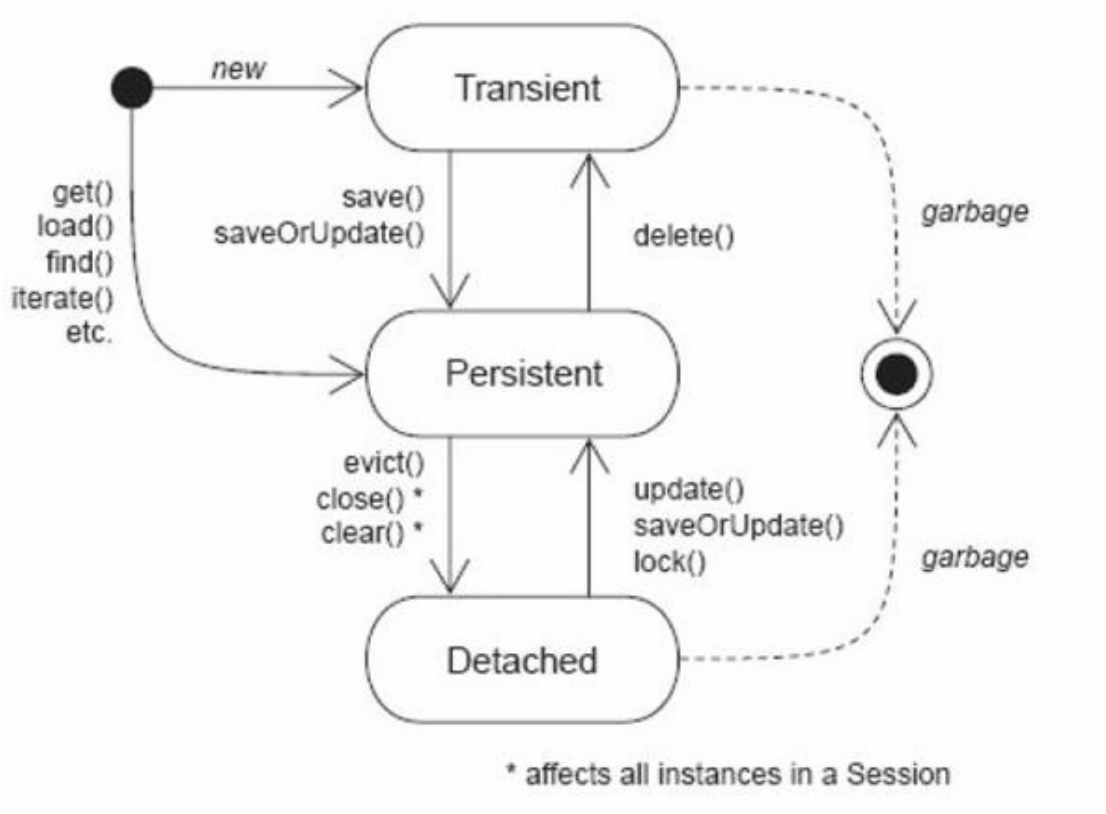

Figura 3: Ciclo de vida (Bauer e King, 2005).

Objeto transiente: quando os objetos são instanciados, ou seja, fazem uso do operador new, são transientes e não estão associados a nenhuma linha do banco de dados. O *Hibernate* não considera instancias transientes como transacionais, isso significa que não existe *rollback* para objetos transientes.

Objeto persistente: é qualquer objeto que possui uma identidade de banco de dados, ou seja, possui um valor de chave primária. Ao contrário do objeto transiente, instancias persistentes são transacionais e estão sempre associadas a uma sessão.

Objetos *detached*: quando o objeto tinha uma instancia associada ao contexto persistente e deixou de ser associada, por fechamento ou finalização de sessão. Sendo objetos intermediários não são transientes nem persistentes.

7.4 HQL

Segundo Bauer e King,

> HQL não é uma linguagem de manipulação de dados como SQL. Só é usado para obtenção de objetos, não para atualizar, inserir ou deletar dados. A sincronização de estado do objeto é o trabalho do gerenciador de persistência, não do desenvolvedor (Bauer e King, 2005, p.185).

Sendo compatível com as seguintes habilidades:

- A habilidade para aplicar restrições a propriedades de objetos associados, relacionados por referencia, ou contidos nas coleções;
- A habilidade para obter somente propriedades de uma entidade ou entidades, sem o custo indireto de carregar a própria entidade em um escopo transacional;
- A habilidade para ordenar os resultados da consulta;
- A habilidade para paginar os resultados;
- Agregação com *group by*, *having* e funções agregadas como *sum*, *min*, *max*;
- Associações externas ao obter múltiplos objetos por linha;
- A habilidade para chamar funções SQL definidas pelo usuário;
- Subconsultas.

Há quatro métodos de busca para qualquer associação conforme Bauer e King (2005, p. 188):

Busca imediata: o objeto associado é buscado imediatamente usando uma leitura no banco de dados seqüencial (ou pesquisa de cachê).

Busca preguiçosa: o objeto associado ou coleção é buscado “preguiçosamente”, quando é acessado pela primeira vez. Isso resulta em uma nova requisição ao banco de dados (exceto se o objeto associado estiver armazenado em cachê).

Busca ávida: o objeto ou coleção é buscado junto com o objeto proprietário, usando uma junção externa SQL, e nenhuma requisição adicional de banco de dados é exigida.

Busca em Lotes: esta técnica pode ser usada para melhorar o desempenho da busca preguiçosa obtendo um lote de objetos ou coleções quando uma associação preguiçosa é acessada (busca em lotes também pode ser usada para melhorar o desempenho da busca imediata).

De acordo com Bauer e King,

> Consideramos HQL o método mais poderoso. As consultas HQL são fáceis de entender e elas utilizam classe persistente e nomes de propriedade no lugar de nomes de tabela e de colunas. HQL é polimórfica: você pode obter todos os objetos com uma determinada interface consultando essa interface. Com HQL, você tem todo o poder de restrições arbitrarias e projeção de resultados, com operadores lógicos e chamadas de função exatamente como em SQL, mas sempre em nível de objeto usando nomes de classe e de propriedade. Você pode usar parâmetros para ligar os argumentos de consulta de uma maneira segura de tipo. Consultas com estilo de relatórios também são suportadas, e é uma área importante onde geralmente faltam recursos para outras soluções MOR (Bauer e King, 2005, p.381).

Com os poderosos recursos HQL, não só podemos recuperar dados do banco de dados como popular esses dados, ou seja, realizamos qualquer tipo de consulta de forma flexível, mas em raras ocasiões o Hibernate pode não oferecer uma solução para uma consulta complexa. Dessa maneira precisamos recorrer ao uso do SQL, realizando consultas “SQL nativas”.

8 ESTUDO DE CASOS

Na figura abaixo está representada a modelagem do banco de dados, no estudo de caso proposto.

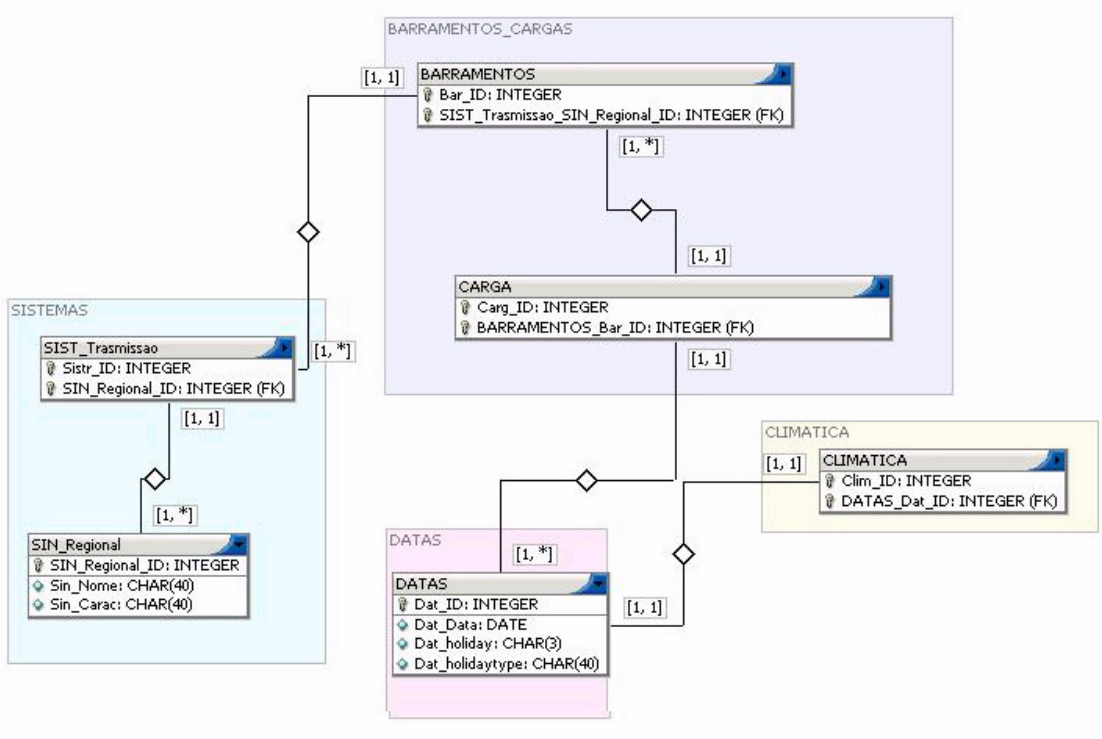

Figura 4: Modelagem do banco de dados

A figura 4 apresenta a modelagem do banco de dados, sendo representada todas às estruturas de tabelas do projeto. Foi utilizado a ferramenta *DBDesigner* para efetuar a representação gráfica.

A figura 5 mostra a estrutura do projeto no *NetBeans 5.0*, usando o framework *Hibernate.*

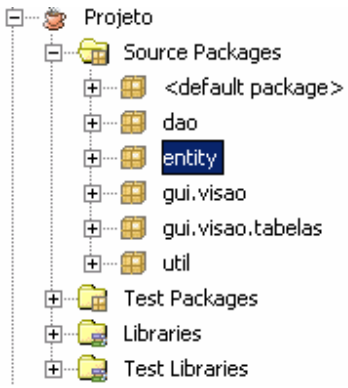

Figura 5: Estrutura do projeto

Abaixo estão representadas as classes persistentes que implementam as entidades de domínio de negócio. Como citado anteriormente o mapeamento objeto/relacional pode ser realizado por meio de anotações. Sendo que qualquer símbolo que comece com @ (arroba) é uma anotação que aparece no código fonte, mas são ignoradas pelo compilador.

A anotação foi anexada a linguagem *Java* após a versão 5.0 do JDK simplificando a maneira de escrever classes persistentes que eram realizadas por meio de arquivos XML.

```
package entity;

import java.io.Serializable;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "sin_regional")

public class SinRegional implements Serializable {

    @Id
    @Column(name = "SIN_Regional_ID", nullable = false)
    private Integer sinRegionalID;

    @Column(name = "Sin_Nome")
    private String sinNome;

    @Column(name = "Sin_Carac")
    private String sinCarac;


    public SinRegional() {
    }

      //métodos getters e setters

    @Override
    public int hashCode() {
        int hash = 0;
        hash += (this.sinRegionalID != null ?
             this.sinRegionalID.hashCode() : 0);
        return hash;
    }

    @Override
    public boolean equals(Object object) {
```

```
        if (!(object instanceof SinRegional)) {
            return false;
        }
        SinRegional other = (SinRegional)object;
        if (this.sinRegionalID != other.sinRegionalID &&
           (this.sinRegionalID == null ||
        !this.sinRegionalID.equals(other.sinRegionalID))) return false;
        return true;
    }
}
```

Quadro 1 – Classe SinRegional utilizando anotações.

No quadro 1 é representada uma classe de mapeamento utilizando anotações, onde o *@Entity* indica que a classe é uma entidade e a mesma é mapeada através da anotação *@Table*, ou seja, o nome da tabela correspondente. Cada coluna da tabela refere-se a um atributo da classe, mapeadas por *@Column* e a sua chave primária recebe a anotação *@Id*. Também são representados os métodos *hashCode* e *equals*, que garantem a igualdade do objeto mapeadas por *@Override* que sobrepõe o método da classe base ou superclasse.

```
package entity;

import java.io.Serializable;
import javax.persistence.Column;
import javax.persistence.EmbeddedId;
import javax.persistence.Entity;
import javax.persistence.Table;

@Entity
@Table(name = "sist_transmissao")

public class SistTransmissao implements Serializable {

    @EmbeddedId
    protected SistTransmissaoPK sistTransmissaoPK;

    @Column(name = "Sistr_Nome")
    private String sistrNome;

    @Column(name = "Sistr_Num_Barras")
    private Integer sistrNumBarras;

    @Column(name = "Sistr_Caract")
    private String sistrCaract;

    @Column(name = "Sistr_Inf_Extras")
    private String sistrInfExtras;
```

```
    public SistTransmissao() {
    }

    public SistTransmissao(SistTransmissaoPK sistTransmissaoPK) {
        this.sistTransmissaoPK = sistTransmissaoPK;
    }


    public SistTransmissao(int sinRegionalID, int sistrID) {
        this.sistTransmissaoPK = new SistTransmissaoPK(sinRegionalID,
        sistrID);
    }

    //métodos getters e setters


    @Override
    public int hashCode() {
        int hash = 0;
        hash += (this.sistTransmissaoPK != null ?
                 this.sistTransmissaoPK.hashCode() : 0);
        return hash;
    }


    @Override
    public boolean equals(Object object) {
        if (!(object instanceof SistTransmissao)) {
            return false;
        }
        SistTransmissao other = (SistTransmissao)object;
        if (this.sistTransmissaoPK != other.sistTransmissaoPK &&
           (this.sistTransmissaoPK == null ||
           !this.sistTransmissaoPK.equals(other.sistTransmissaoPK)))
             return false;
        return true;
    }

}
```

Quadro 2 – Classe SistTransmissao utilizando anotações.

No quadro 2 observamos uma nova anotação o @EmbeddedId, que por sua vez, define a chave composta. Abaixo é representada a classe que compõe a chave composta.

```
package entity;

import java.io.Serializable;
import javax.persistence.Column;
import javax.persistence.Embeddable;


@Embeddable
```

```
public class SistTransmissaoPK implements Serializable {

    @Column(name = "Sistr_ID", nullable = false)
    private int sistrID;

    @Column(name = "SIN_Regional_ID", nullable = false)
    private int sinRegionalID;


    public SistTransmissaoPK() {
    }


    public SistTransmissaoPK(int sinRegionalID, int sistrID) {
        this.sinRegionalID = sinRegionalID;
        this.sistrID = sistrID;
    }

    //métodos getters e setters

    @Override
    public int hashCode() {
        int hash = 0;
        hash += (int)sinRegionalID;
        hash += (int)sistrID;
        return hash;
    }

    @Override
    public boolean equals(Object object) {

        if (!(object instanceof SistTransmissaoPK)) {
            return false;
        }
        SistTransmissaoPK other = (SistTransmissaoPK)object;
        if (this.sinRegionalID != other.sinRegionalID) return false;
        if (this.sistrID != other.sistrID) return false;
        return true;
    }
```

Quadro 3 – Classe SistTransmissaoPK

No quadro 3 a classe é identificada com a anotação @Embeddable, indicando que se trata de uma classe de chave composta. Lembrando que todas as classes persistentes implementam a interface *Serializable*, permitindo que o objeto possa ser armazenado em disco ou percorrer pela rede. A figura abaixo mostra todas as classes persistentes dentro do pacote *entity*.

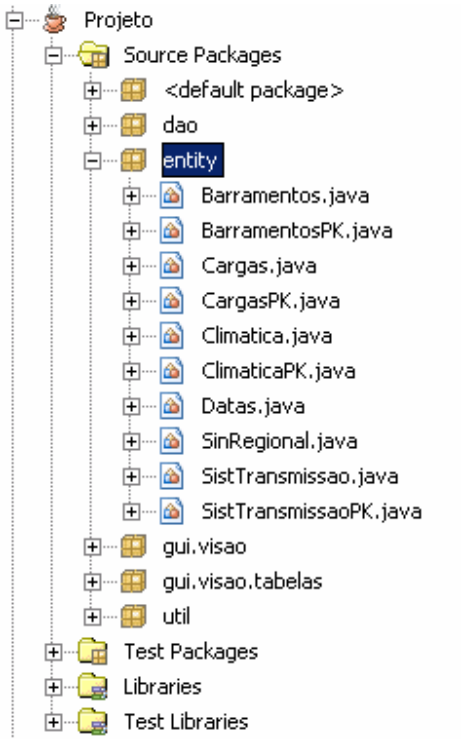

Figura 6: Classes entidades - *Hibernate*

Fazendo uso da mesma modelagem, porém, sem a utilização do *Hibernate*, ou seja, a mesma aplicação com a implementação de JDBC/SQL. Abaixo está representada a estrutura do projeto.

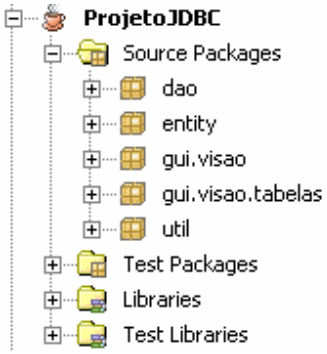

Figura 7: Estrutura do projetoJDBC

```
package entity;

public class SinRegional {

    private Integer sinRegionalID;
```

```
    private String sinNome;
    private String sinCarac;

    public SinRegional() {
    }

      //Métodos getters e setters
}
```

Quadro 4: Classe SinRegional

```
package entity;

public class SistTransmissao {

    private int sistrID;
    private int sinRegionalID;
    private String sistrNome;
    private Integer sistrNumBarras;
    private String sistrCaract;
    private String sistrInfExtras;

        public SistTransmissao() {
        }

      //Métodos getters e setters

}
```

Quadro 5: Classe SistTransmissao

Observa-se no quadro 4 e 5 classes de entidade no projeto JDBC. Diferentemente do projeto *Hibernate*, não fazem uso de nenhuma anotação, bem como não sendo necessário a identificação de chaves estrangeiras. Na figura abaixo mostra todas as classes dentro do pacote *entity*.

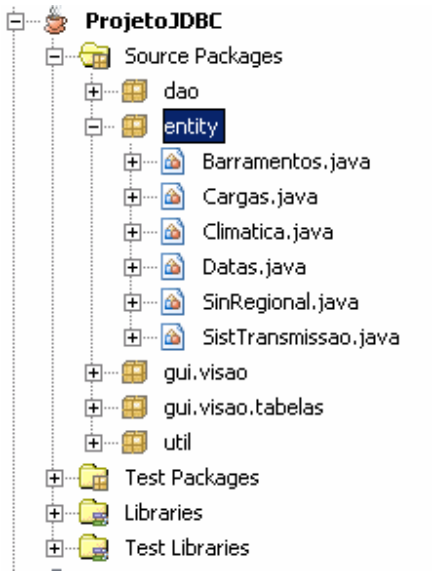

Figura 8: Classes entidades - JDBC

Para dar inicio ao *Hibernate* é preciso configurar seu principal arquivo, *hibernate.cfg.xml*. Através dele é configurado o acesso ao banco de dados e a localização das entidades mapeadas.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property
name="hibernate.connection.driver_class">org.postgreSQL.Driver</property>
        <property name="hibernate.connection.password">senha</property>
        <property
name="hibernate.connection.url">jdbc:postgreSQL://localhost:porta/nomedob
anco</property>
        <property name="hibernate.connection.username">usuario</property>
        <property
name="hibernate.dialect">org.hibernate.dialect.PostgreSQLDialect</propert
y>
        <mapping class="entity.SinRegional" />
        <mapping class="entity.SistTransmissao" />
        <mapping class="entity.Barramentos" />
        <mapping class="entity.Cargas" />
        <mapping class="entity.Climatica" />
        <mapping class="entity.Datas" />
    </session-factory>
</hibernate-configuration>
```

Quadro 6 – Arquivo de configuração utilizando o banco de dados *PostgreSQL*.

No quadro 6 são apresentadas os principais parâmetros de configuração do arquivo *hibernate.cfg.xml*. As descrições desses parâmetros são:

- *hibernate.connection.driver_class:* Classe do *Driver* JDBC utilizado para realizar a conexão.
- *hibernate.connection.password:* Senha do usuário utilizada para a conexão com o banco de dados.
- *hibernate.connection.url:* String de conexão contendo o mecanismo de acesso JDBC, o SGBD, nesse caso o Postgres, seguido do servidor, porta e nome do banco.
- *hibernate.connection.username:* Nome do usuario do banco de dados.
- *hibernate.dialect:* Dialeto contendo características particulares do banco de dados.
- *mapping class:* representa as classes de domínio mapeadas com anotações.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property
name="hibernate.connection.driver_class">org.gjt.mm.mySQL.Driver</propert
y>
        <property name="hibernate.connection.password">senha</property>
        <property
name="hibernate.connection.url">jdbc:mySQL://localhost:porta/nomedobanco<
/property>
        <property name="hibernate.connection.username">usuario</property>
        <property
name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
        <mapping class="entity.SinRegional" />
        <mapping class="entity.SistTransmissao" />
        <mapping class="entity.Barramentos" />
        <mapping class="entity.Cargas" />
        <mapping class="entity.Climatica" />
        <mapping class="entity.Datas" />
    </session-factory>
</hibernate-configuration>
```

Quadro 7 – Arquivo de configuração utilizando o banco de dados *MySQL*.

Observa-se no quadro 7, quando realizada a mudança de banco de dados, são modificadas a classe do *Driver,* a *url* de conexão, usuário, senha e o dialeto do banco de dados.

Na tabela 4 contém alguns dialetos utilizados pelo *Hibernate*.

SGBD	DIALETO
PostgreSQL	org.hibernate.dialect.PostgreSQLDialect
MySQL	org.hibernate.dialect.MySQLDialect
MySQL com InnoDB	org.hibernate.dialect.MySQLInnoDBDialect
MySQL com MyISAM	org.hibernate.dialect.MySQLMyISAMDialect
Oracle 9i/10g	org.hibernate.dialect.Oracle9Dialect
Microsoft SQL Server	org.hibernate.dialect.SQLServerDialect
SAP DB	org.hibernate.dialect.SAPDBDialect
HypersonicSQL	org.hibernate.dialect.HSQLDialect
Ingres	org.hibernate.dialect.IngresDialect
Progress	org.hibernate.dialect.ProgressDialect
Interbase	org.hibernate.dialect.InterbaseDialect
Pointbase	org.hibernate.dialect.PointbaseDialect
FrontBase	org.hibernate.dialect.FrontbaseDialect
Firebird	org.hibernate.dialect.FirebirdDialect
Informix	org.hibernate.dialect.InformixDialect

Tabela 4: Dialetos do *Hibernate*.

```
package util;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.AnnotationConfiguration;
import org.hibernate.cfg.Configuration;

public class ConnectDBMySQL {
```

```
    private static final SessionFactory sessionFactory;
    private static final ThreadLocal<Session> threadLocal = new
           ThreadLocal<Session>();

    static {

        try {
            Configuration configuration = new
                AnnotationConfiguration();
            configuration.configure("hibernate.cfg.xml");
            sessionFactory = configuration.buildSessionFactory();

        } catch (Throwable t) {
            throw new ExceptionInInitializerError(t);
        }
    }

    public static Session getInstance() {
        Session session = (Session) threadLocal.get();
        session = sessionFactory.openSession();
        threadLocal.set(session);
        return session;
    }

}
```

Quadro 8 – Classe de conexão usando *Hibernate*

No quadro 8, é apresentada a classe de conexão que se utiliza do arquivo mostrado anteriormente *hibernate.cfg.xml,* para configurar e fazer uso do método "*buildSessionFactory*". Retornando um "*sessionFactory*" que por sua vez será responsável por obter e retornar objetos.

Nota-se também que é criada uma instância do tipo "*Session*" que associada a um objeto "*threadLocal*"é capaz de abrir um processo concorrente a cada sessão aberta.

Para obter conexões com mais de uma banco de dados é preciso mais dois arquivos o primeiro será o arquivo de configuração representado anteriormente nos quadros 6 e 7, e fazendo uso dessas configurações outro arquivo de conexão representado no quadro 8. Como já citado anteriormente será criado dois "SessionFactory".

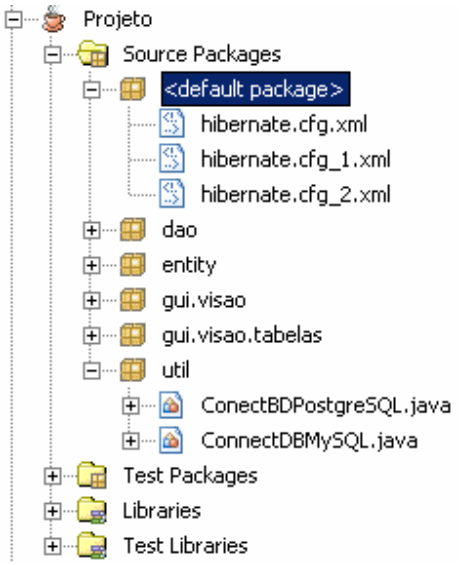

Figura 9: Arquivos para conexões em banco de dados distintos

Como pode ser visto na figura 9, cria-se mais de um arquivo de configuração, tanto para banco de dados distintos como para diferentes SGBDs. Dessa maneira, são criadas diferentes classes de conexão de acordo com os SGBDs utilizados.

```
package util;

import java.SQL.*;
import java.SQL.Connection;
import java.SQL.DriverManager;

public class ConnectBDPostgreSQL {

    String url = "jdbc:postgreSQL://localhost:porta/nomedobanco";
    String username = "usuario";
    String password = "senha";

    private Connection connection;

    public Connection getConexao() throws Exception {
        try {
           Class.forName( "org.postgreSQL.Driver" );

            connection  =  DriverManager.getConnection(  url,  username,
password );
        } catch ( ClassNotFoundException cnfex ) {
            System.err.println("Failed to load JDBC/ODBC driver." );
            cnfex.printStackTrace();
            System.exit( 1 );
        } catch ( SQLException SQLex ) {
            System.err.println( "Unable to connect" );
            SQLex.printStackTrace( );
```

```
        }

        return connection;
    }

}
```

Quadro 9 – Classe de conexão usando JDBC.

O quadro 9 mostra uma classe comum de conexão ao banco de dados sem o uso do *Hibernate*. Onde é criado um objeto tipo "*Connection"* que utilizando a *string url,* nome do usuário e senha, obtem uma conexão. No exemplo acima é utilizado a *string url* do banco de dados *PostgreSQL*.

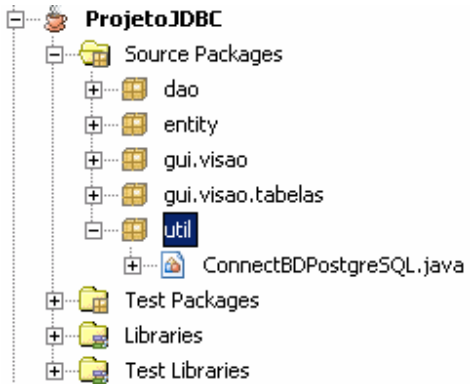

Figura 10: Arquivo para conexões em banco de dados

A figura 10 apresenta a classe de conexão ao banco de dados. Lembrando que para ser efetuada a conexão com outros SGBDs, deve ser criado outra classe com as devidas especificações.

```
package dao;

import entity.*;
import java.util.List;
import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.Transaction;
import util.ConnectBDPostgreSQL;


public class ClasseDao {

    private Session session;
```

```
public void salvar(Object object) {

    session = ConnectDB.getInstance();
    Transaction tx = null;
    try {
        tx = session.beginTransaction();
        session.save(object);
        tx.commit();
    } catch (HibernateException e) {
        e.printStackTrace();
    } finally {
        session.close();
    }
}

public void editar(Object object) {

    session = ConnectDB.getInstance();
    Transaction tx = null;
    try {
        tx = session.beginTransaction();
        session.update(object);
        tx.commit();
    } catch (HibernateException e) {
        e.printStackTrace();
    } finally {
        session.close();
    }
}

public void deletar(Object object) {

    session = ConnectDB.getInstance();
    Transaction tx = null;
    try {
        tx = session.beginTransaction();
        session.delete(object);
        tx.commit();
    } catch (HibernateException e) {
        e.printStackTrace();
    } finally {
        session.close();
    }
}

    public List consultarTabela(String entityName) {
    session = ConnectDB.getInstance();
    List list = session.createQuery("from " + entityName).list();
    return list;
}

public SinRegional consultarSinRegional(int codigo) {
    session = ConnectDB.getInstance();
    SinRegional sinRegional =
   (SinRegional)session.createQuery("from SinRegional where
                  SIN_Regional_ID = " + codigo).uniqueResult();
```

```
        return sinRegional;
    }
```

Quadro 10 – Classe *DAO* utilizando *Hibernate.*

No quadro 10 é utilizada a classe do padrão *Data Access Object* (DAO), sendo responsável por gerenciar e encapsular o acesso aos dados. Observa-se que é criado um objeto *"session"*, garantindo uma única sessão aberta.

Os métodos salvar, editar e deletar, recebem como parâmetro um tipo *"object"* que pode ser qualquer tabela mapeada no arquivo "*hibernate.cfg.xml"*, ou seja, um método salvar pode ser utilizado por todas as classes mapeadas para inserir novos dados em suas respectivas tabelas. O mesmo acontece para os métodos editar e deletar.

O método consultar tabela recebe um tipo *String* com o nome da entidade que se deseja consultar e retorna um "*List*" com todos os dados da tabela. Em seguida o método *consultarSinRegional()* retorna uma linha segundo o código passado por parâmetro.

Nota-se que essas consultas utilizam o *HQL* (Hibernate Query Language), a linguagem de consultas do *Hibernate*, onde é possível escrever sentenças orientadas a objetos, usando herança, polimorfismo e associações recuperando e modificando dados.

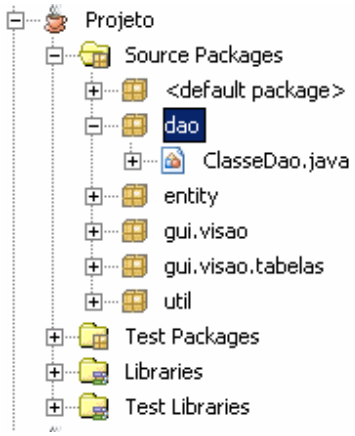

Figura 11: Classe DAO projeto Hibernate

A figura 11 mostra apenas uma classe para realizar a persistência de toda a aplicação.

```
package dao;

import entity.*;
import java.SQL.*;
import util.ConnectBDPostgreSQL;
import java.util.*;
public class SinRegionalDao {
```

```
    ConnectDB con = new ConnectDB();

    public Vector consultarSinRegional() throws Exception{
        Connection c = con.getConexao();
        PreparedStatement ps = null;
        ResultSet rs = null;
        Vector lista = new Vector();
        try {
            ps = c.prepareStatement("SELECT * FROM sin_regional");
            rs = ps.executeQuery();

            while (rs.next()) {
                SinRegional sinReg = new SinRegional();
                sinReg.setSinRegionalID(rs.getInt(1));
                sinReg.setSinNome(rs.getString(2));
                sinReg.setSinCarac(rs.getString(3));
                lista.add(sinReg);
            }
        } finally {
            if (rs != null) {
                rs.close();
                rs = null;
            }
            if (ps != null) {
                ps.close();
                ps = null;
            }
            if (c != null) {
                c.close();
                c = null;
            }
        }
        return lista;
    }

    public SinRegional consultarSinRegionalPorCod(String codigo) throws
Exception{
        Connection c = con.getConexao();
        PreparedStatement ps = null;
        ResultSet rs = null;
        SinRegional sinReg = new SinRegional();
        try {
            ps = c.prepareStatement("SELECT * FROM sin_regional WHERE
SIN_Regional_ID = '" + codigo + "'");
            rs = ps.executeQuery();

            while (rs.next()) {
                sinReg.setSinRegionalID(rs.getInt(1));
                sinReg.setSinNome(rs.getString(2));
                sinReg.setSinCarac(rs.getString(3));
            }
        } finally {
            if (rs != null) {
                rs.close();
                rs = null;
            }
            if (ps != null) {
```

```
                ps.close();
                ps = null;
            }
            if (c != null) {
                c.close();
                c = null;
            }
        }
        return sinReg;
    }

    public   void   salvarSinRegional(SinRegional   SinRegional)   throws
Exception{
        Connection c = con.getConexao();
        PreparedStatement ps = null;
        Vector lista = new Vector();
        int cod = 0;
        try {
            ps = c.prepareStatement("INSERT INTO sin_regional VALUES (?,
?, ?)");
            ps.setString(1,String.valueOf(cod));
            ps.setString(2,SinRegional.getSinNome());
            ps.setString(3,SinRegional.getSinCarac());
            ps.executeUpdate();
        } finally {
            if (ps != null) {
                ps.close();
                ps = null;
            }
            if (c != null) {
                c.close();
                c = null;
            }
        }

    }

    public void deletarSinRegional(String codigo) throws Exception {
        Connection c = con.getConexao();
        PreparedStatement ps = null;
        try {
            ps  =  c.prepareStatement("DELETE  FROM  sin_regional  WHERE
SIN_Regional_ID = '" + codigo + "'");
            ps.executeUpdate();
        } finally {
            if (ps != null) {
                ps.close();
                ps = null;
            }
            if (c != null) {
                c.close();
                c = null;
            }
        }
    }
```

```
    public void editarSinRegional(SinRegional SinRegional) throws
Exception {
        Connection c = con.getConexao();
        PreparedStatement ps = null;
        String cod = Integer.toString(SinRegional.getSinRegionalID());
        try {
            ps = c.prepareStatement("UPDATE sin_regional SET Sin_Nome =
?, Sin_Carac = ? WHERE SIN_Regional_ID = '" + cod + "'");
            ps.setString(1,SinRegional.getSinNome());
            ps.setString(2,SinRegional.getSinCarac());
            ps.executeUpdate();

        }finally {
            if (ps != null) {
                ps.close();
                ps = null;
            }
            if (c != null) {
                c.close();
                c = null;
            }
        }
    }

}
```

Quadro 11 – Classe *DAO* da tabela *sinRegional* utilizando SQL.

No quadro 11 a classe com operações de *insert, update, delete* e dois tipos de consulta *consultarSinRegional()* retorna um tipo Vector com todas as colunas e linhas da tabela, já a *consultarSinRegionalPorCod()* retorna uma linha, segundo o código que é passado por parâmetro.

Nota-se um aumento significativo no número de linhas de código comparado ao quadro 6. Vale ressaltar que no quadro 6 os métodos são utilizados por todas as tabelas do banco de dados, já o quadro 7 apresenta os mesmos métodos fazendo uso da linguagem SQL, para apenas uma tabela do banco de dados (*SinRegional*). Sendo assim necessárias um método de inserção, atualização, exclusão e consultas desses dados armazenados para cada tabela do banco de dados. Como pode ser visto na figura 12:

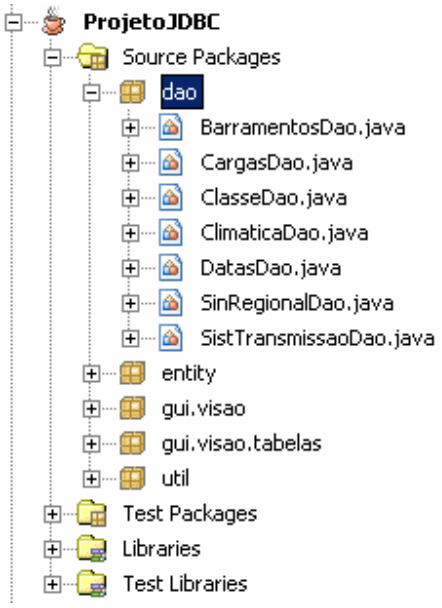

Figura 12: Classe DAO – projeto JDBC

Em ambas as aplicações os pacotes “gui.visão” e “gui.visão.tabelas”, apresentam o mesmo conteúdo, ou seja a interface gráfica das aplicações. Seguem abaixo as representações de cada tabela.

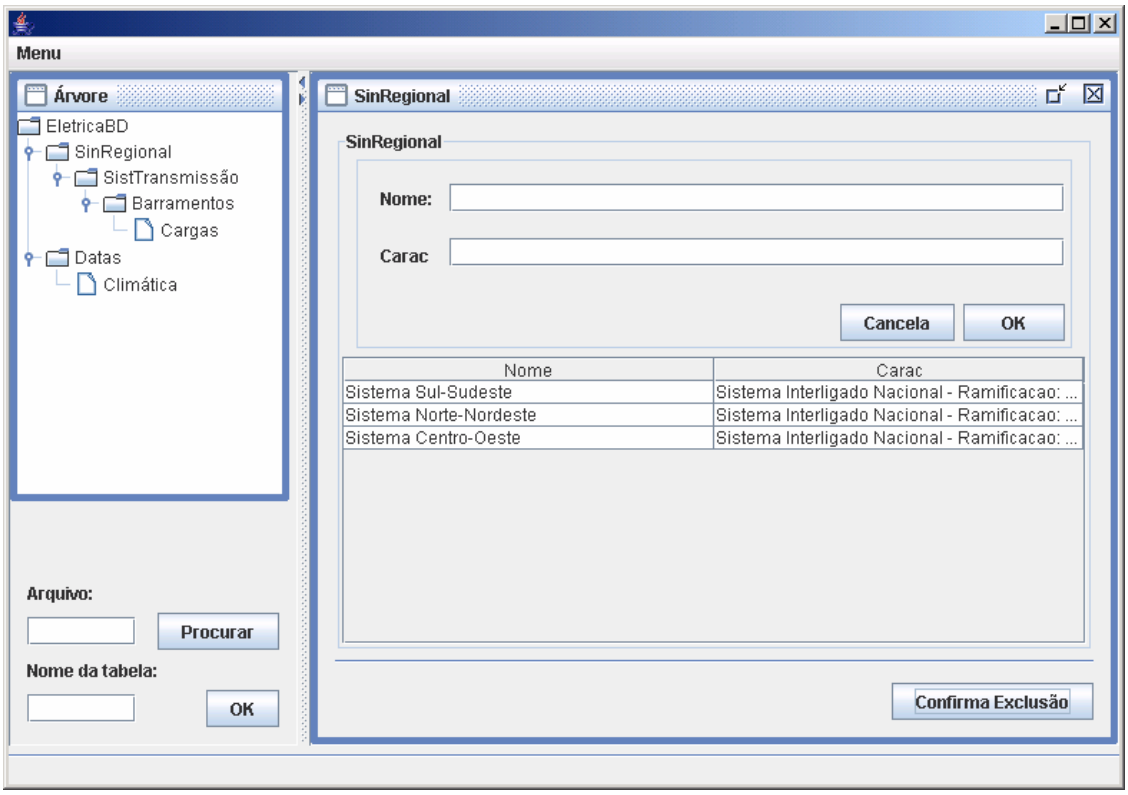

Figura 13: SinRegional

A figura 13 representa a tabela SinRegional, podendo ser efetuada as operações de inclusão, alteração e exclusão das informações. Tendo o resultado dessas operações a representação simultânea na tabela.

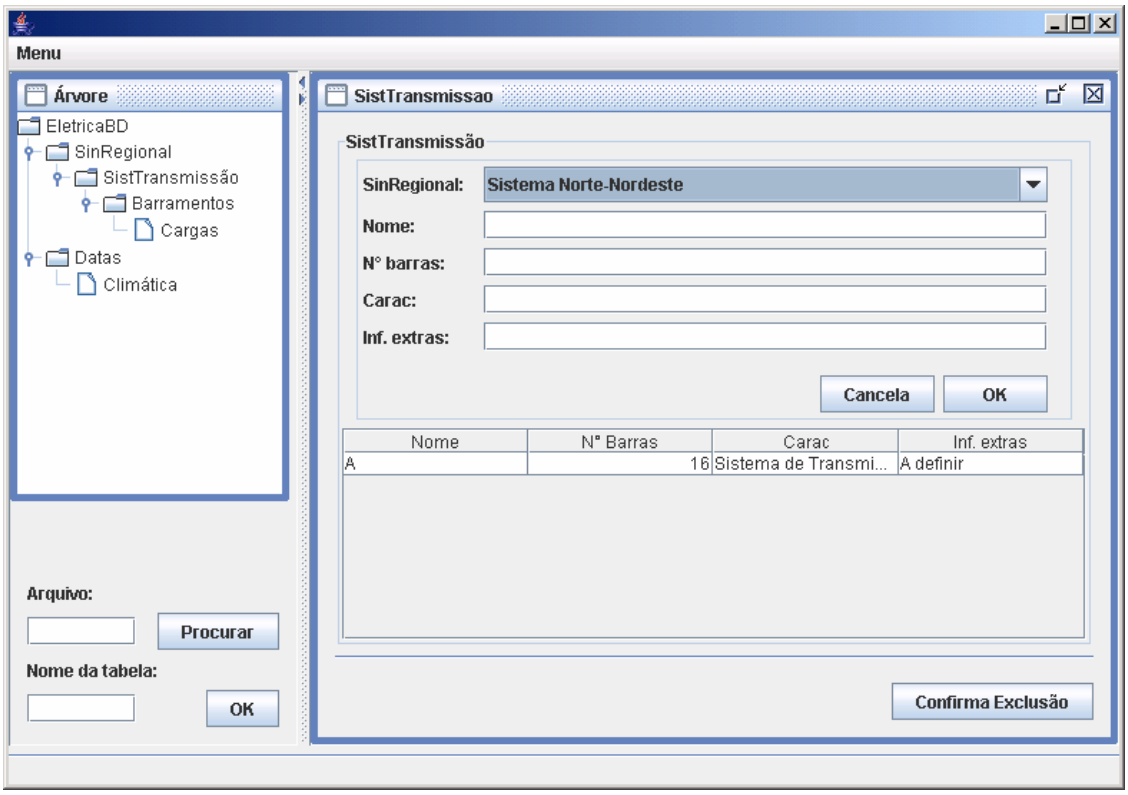

Figura 14: SistTransmissao

A figura 14 representa a tabela SistTransmissao, que contém a tabela SinRegional como chave estrangeira, podendo ser efetuada as operações de inclusão, alteração e exclusão das informações. Tendo o resultado dessas operações a representação simultânea na tabela.

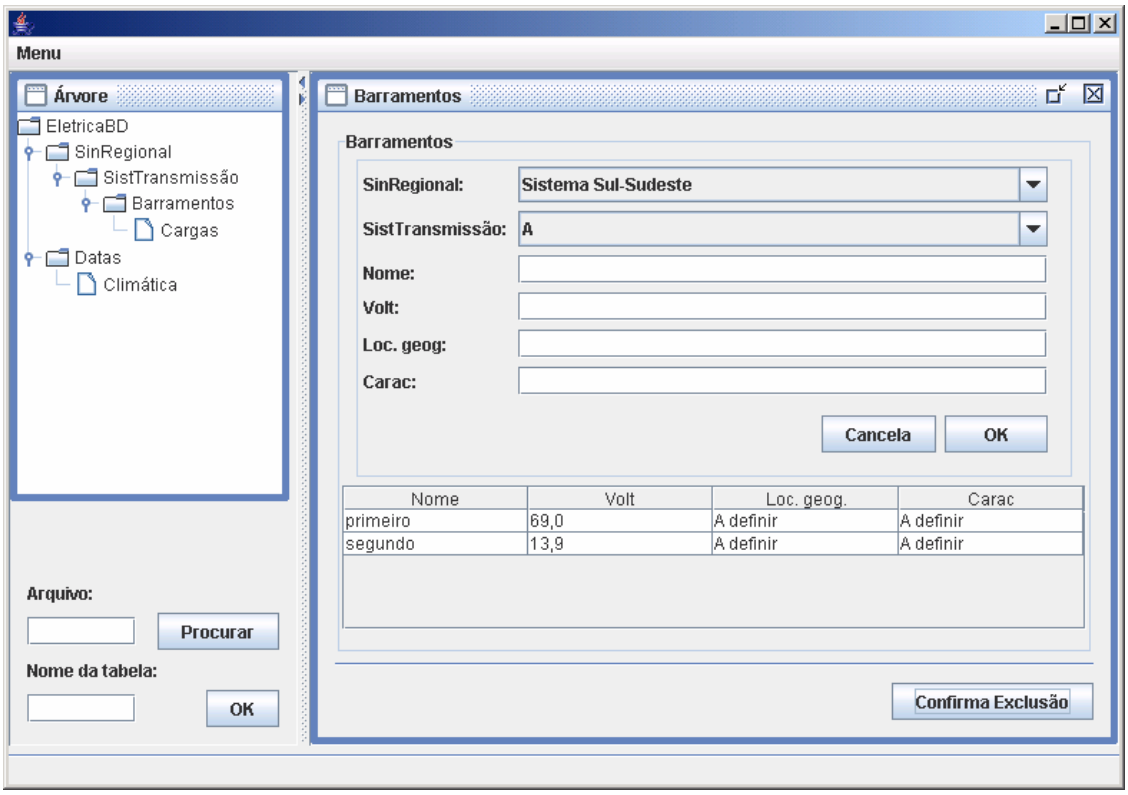

Figura 15: Barramentos

A figura 15 representa a tabela Barramentos, que contém as tabelas SinRegional e SistTransmissao como chave estrangeira, podendo ser efetuada as operações de inclusão, alteração e exclusão das informações. Tendo o resultado dessas operações a representação simultânea na tabela.

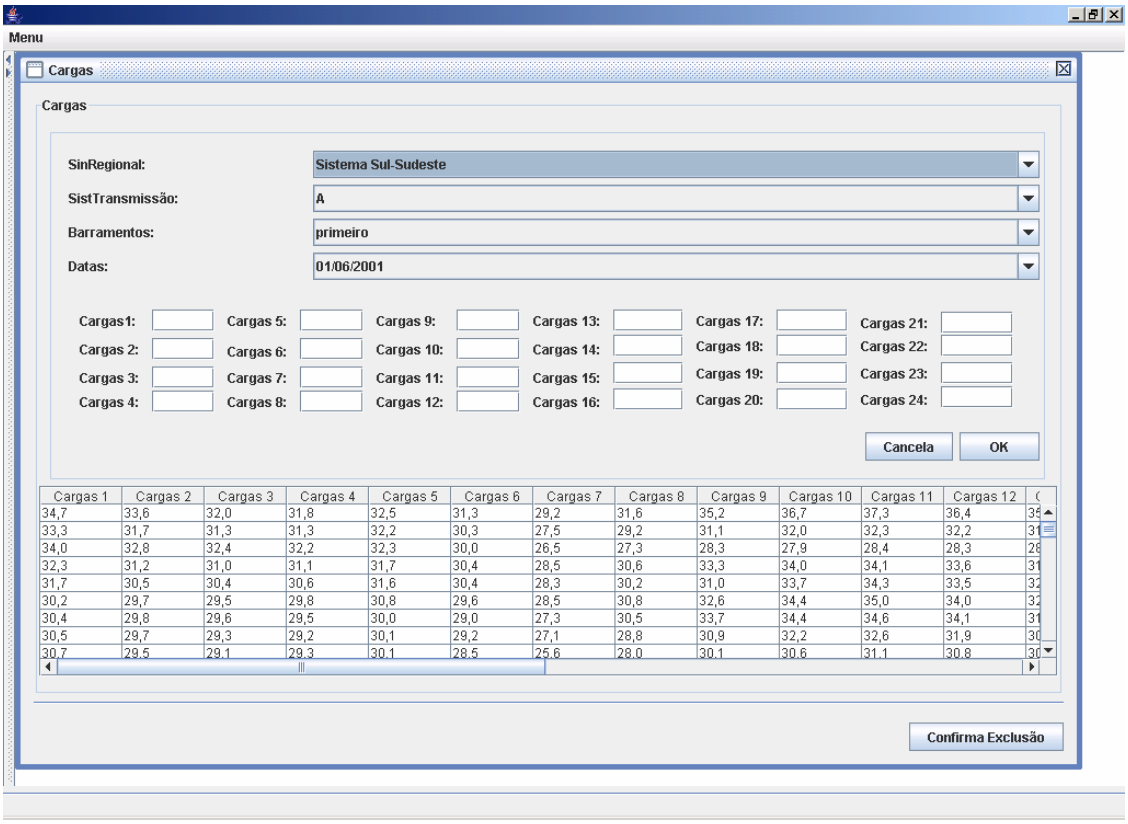

Figura 16: Cargas

A figura 16 representa a tabela Cargas, que contém as tabelas SinRegional, SistTransmissao, Barramentos e Datas como chave estrangeira, podendo ser efetuada as operações de inclusão, alteração e exclusão das informações. Tendo o resultado dessas operações a representação simultânea na tabela.

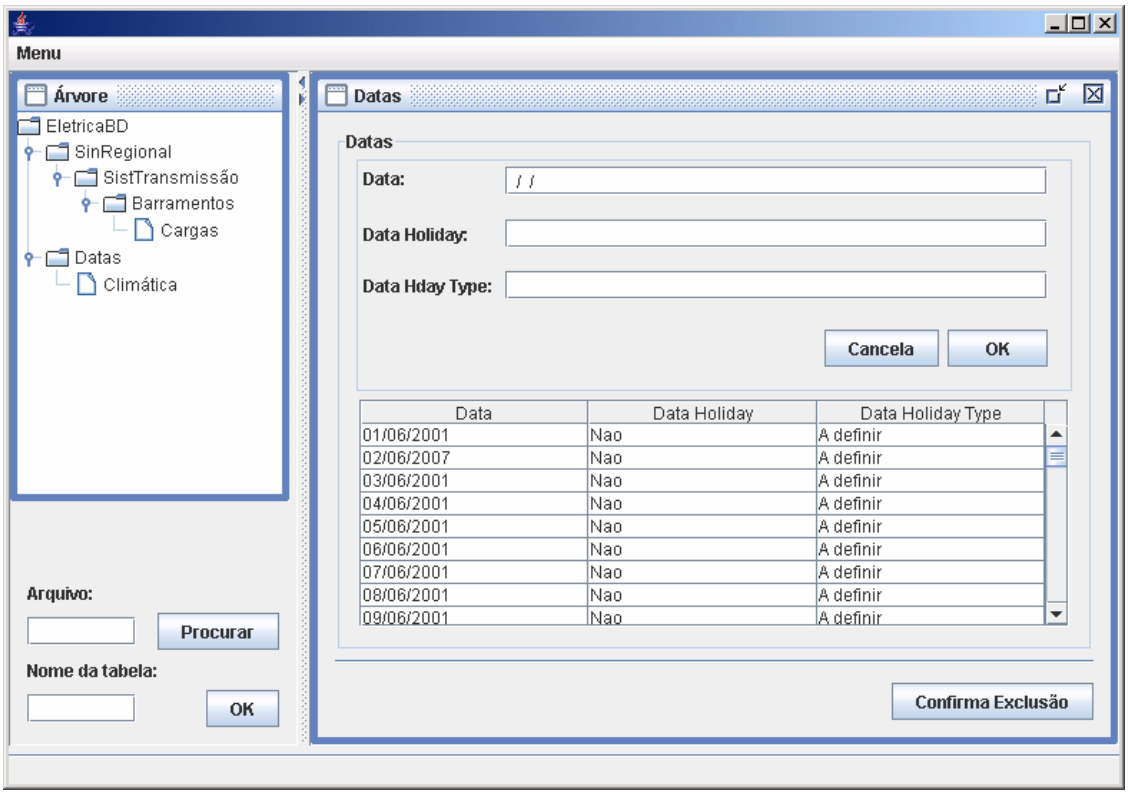

Figura 17: Datas

A figura 17 representa a tabela Datas, podendo ser efetuada as operações de inclusão, alteração e exclusão das informações. Tendo o resultado dessas operações a representação simultânea na tabela.

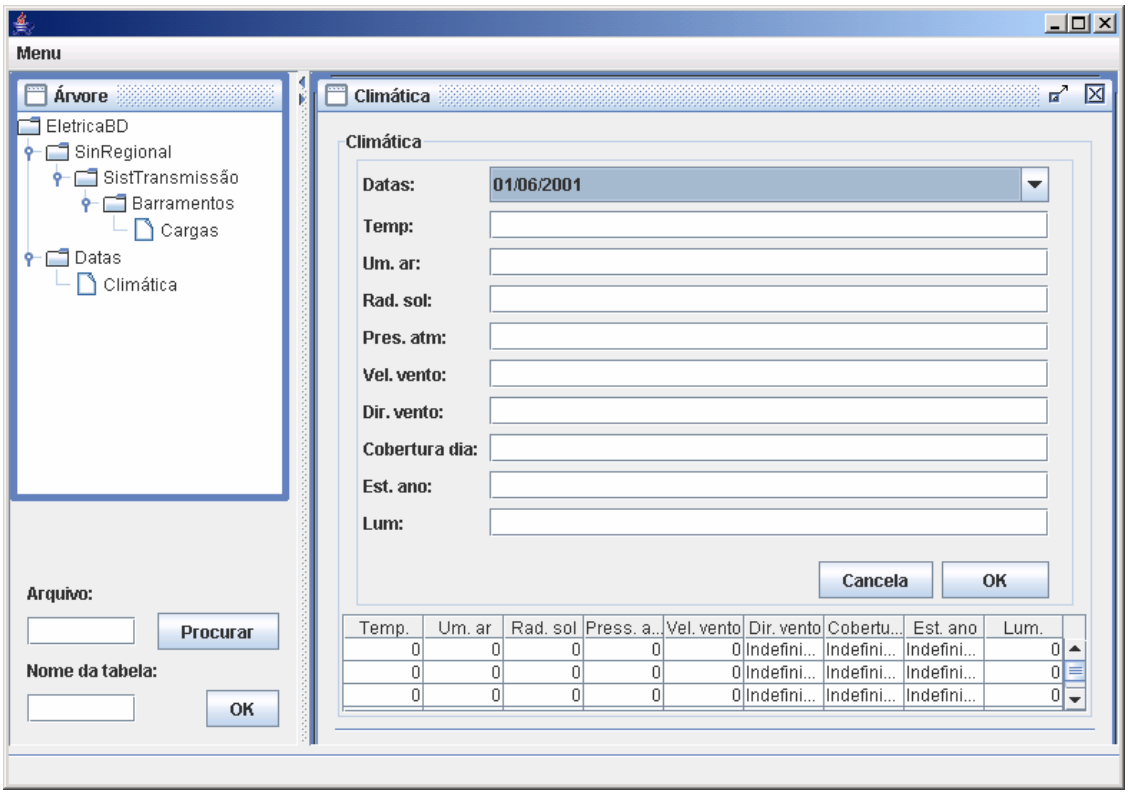

Figura 18: Climatica

A figura 18 representa a tabela Climatica, que contém a tabela Datas como chave estrangeira, podendo ser efetuada as operações de inclusão, alteração e exclusão das informações. Tendo o resultado dessas operações a representação simultânea na tabela.

Vale a pena ressaltar que todas as tabelas foram implementadas as ações de inclusão, alteração, exclusão e consulta. Ficando para uma futura implementação a inserção de arquivo texto gerando uma nova tabela, e com isso, sendo representada na árvore.

8.1 Análise de banco de dados

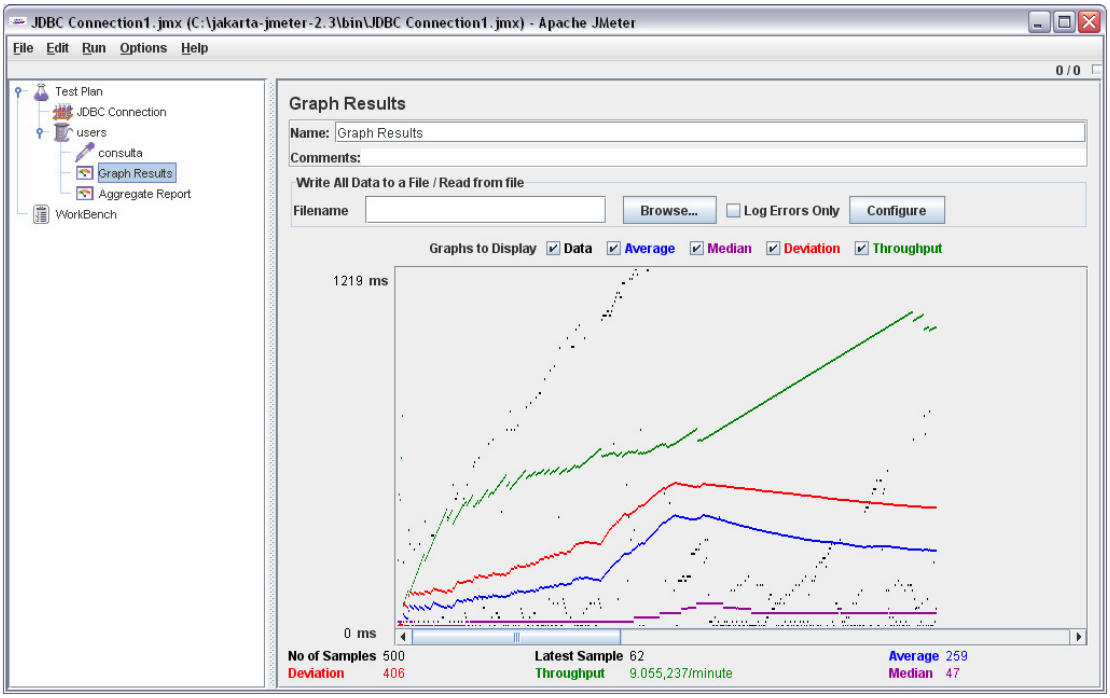

Figura 19: Gráfico MySQL

Na figura 19 é apresentado o gráfico do MySQL, em preto os dados que foram solititados durante a consulta, em verde a vazão, ou tempo de resposta do SGBD, em vermelho o desvio padrão, em azul a mediana e em roxo a média. Observa-se que a média se mantém, pois o desvio e a mediana permanecem em paralelo. A vazão é crescente conforme as solicitações efetuadas.

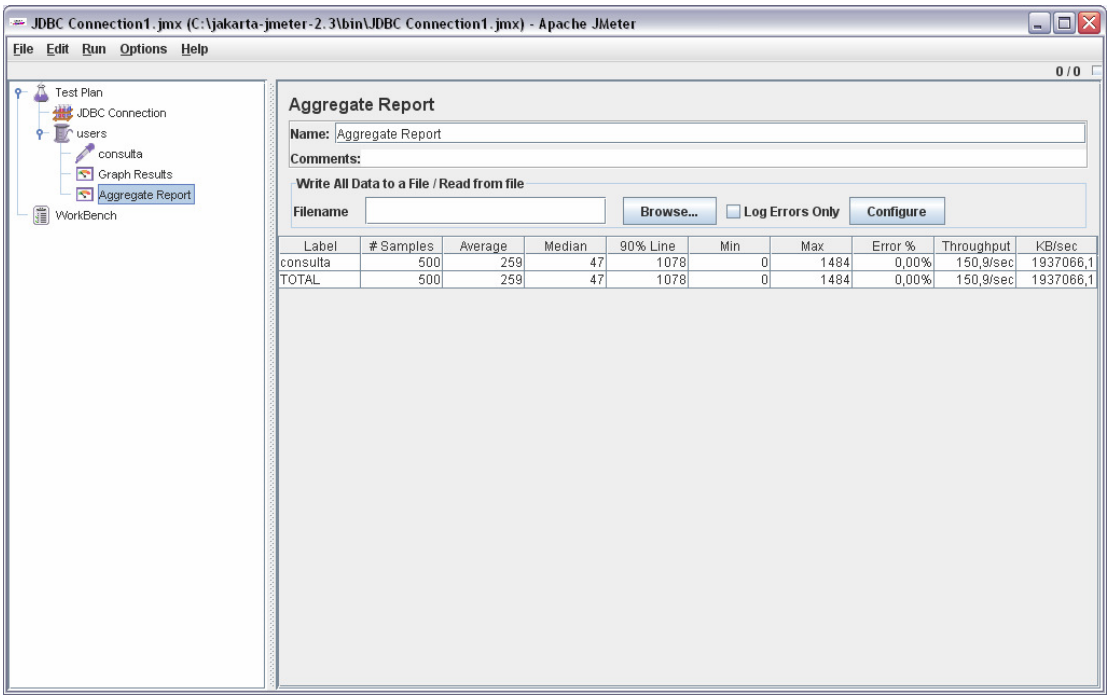

Figura 20: Relatório MySQL

Na figura 20 é apresentado o relatório onde foram feitas 500 transações em 150,9 segundos.

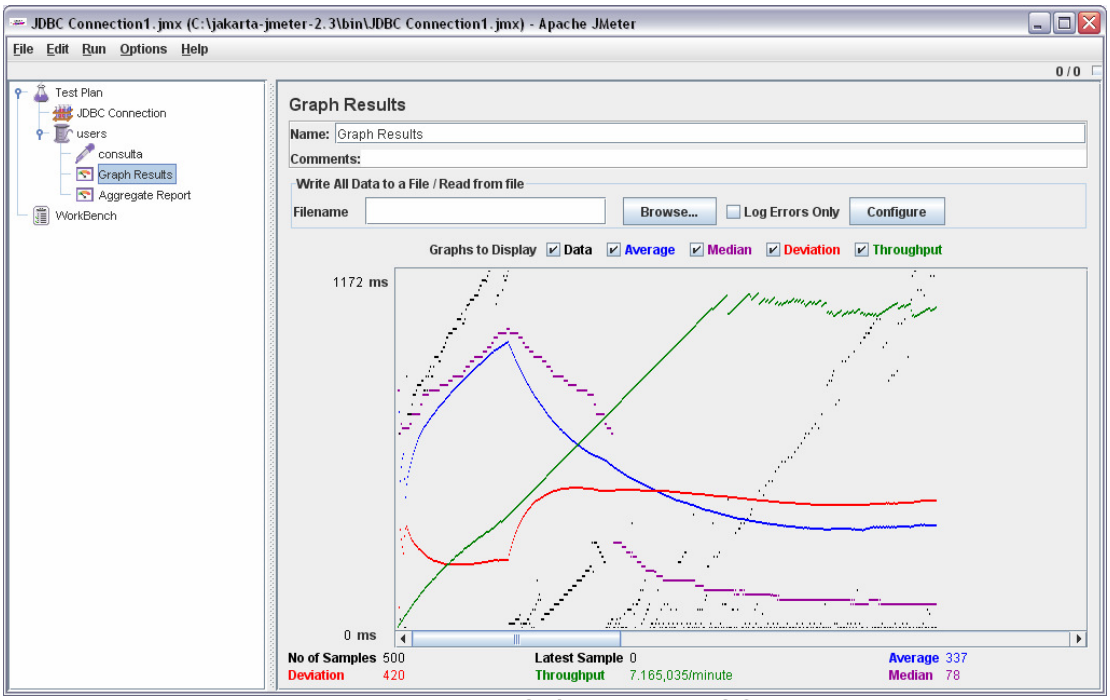

Figura 21: Gráfico PostgreSQL

Na figura 21 é apresentado o gráfico do PostgreSQL, em preto os dados, em verde a vazão, em vermelho o desvio padrão, em azul a mediana e em roxo a média. Observa-se que a média no inicio é alta, se mantém, pois o desvio e a mediana permanecem em paralelo. A vazão é crescente conforme as solicitações efetuadas. Portando o PostgreSQL, obteve maior rapidez nas consultas testadas.

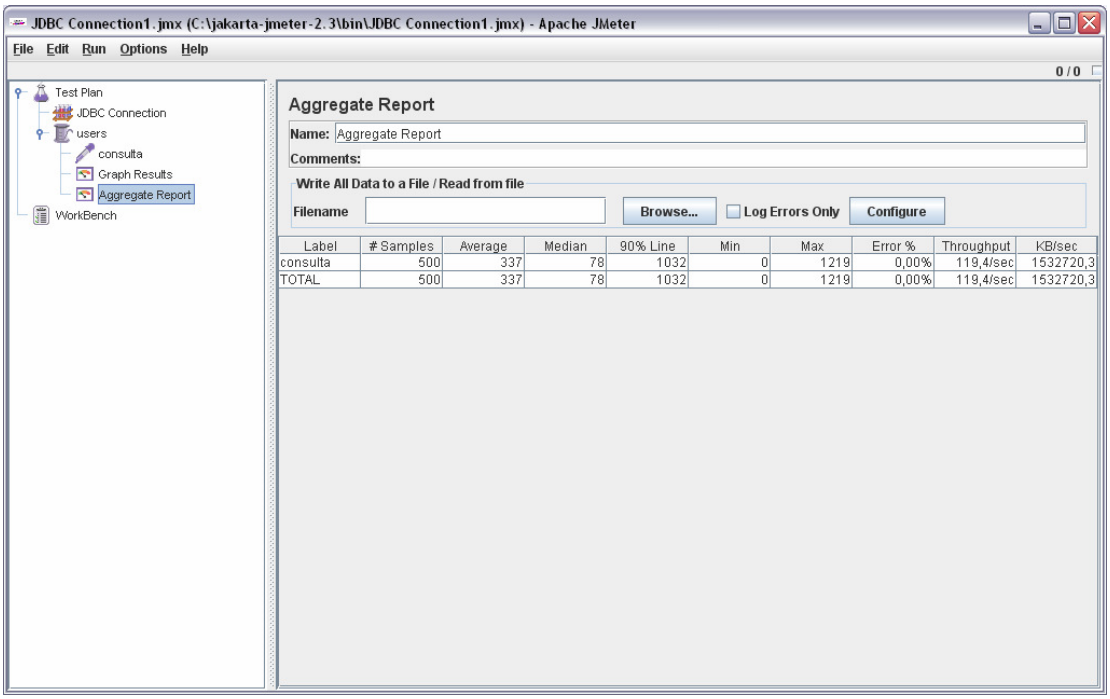

Figura 22: Relatório PostgreSQL

O teste foi realizado pelo software JMeter 2.3, sobre os bancos de dados MySQL 5.0 e PostgreSQL 8.2. Ambos os testes com 500 transações, o banco de dados MySQL apresenta maior vazão por segundo em relação ao banco de dados PostgreSQL, ao realizar uma consulta simples, com os seguintes parâmetros: 100 usuários efetuando uma consulta simultaneamente, com um *loop* de 5 vezes.

8.2 Análise de tempo

A seguir são apresentados os resultados dos testes efetuados nos bancos de dados MySQL e PostgreSQL.

MySQL

Consulta uma tabela								
	Hibernate	JDBC		Hibernate	JDBC		Hibernate	JDBC
	5875	2063		3360	859		2328	2234
	0	78		0	78		0	16
	0	62		0	62		0	31
	0	141		0	47		16	16
	0	172		0	125		16	32
Média	1175	503,2		672	234,2		472	465,8

Consulta duas tabelas								
	Hibernate	JDBC		Hibernate	JDBC		Hibernate	JDBC
	4407	845		4907	313		1579	344
	0	78		16	47		16	62
	16	94		0	47		0	47
	16	63		15	31		0	31
	16	16		0	31		0	16
Média	891	219,2		987,6	93,8		319	100

Consulta quatro tabelas								
	Hibernate	JDBC		Hibernate	JDBC		Hibernate	JDBC
	5609	1219		2032	499		2859	1593
	156	172		291	93		281	78
	156	78		156	63		141	78
	156	141		172	78		171	156
	234	47		125	63		110	62
Média	1262,2	331,4		555,2	159,2		712,4	393,4

Tabela 5: Consultas MySQL, terceiro plano de testes

Consulta a uma tabela						
	Hibernate	JDBC	Hibernate	JDBC	Hibernate	JDBC
	7543	2654	4762	1064	3854	3543
	45	87	31	89	31	52
	25	84	29	72	27	29
	29	253	17	243	45	41
	15	187	29	185	15	32
Média	1531,4	653	973,6	330,6	794,4	739,4
Consulta duas tabelas						
	Hibernate	JDBC	Hibernate	JDBC	Hibernate	JDBC
	4863	953	5432	359	1895	653
	29	134	31	76	45	67
	27	86	23	51	31	36
	41	73	45	45	31	45
	32	67	27	78	25	67
Média	998,4	262,6	1111,6	121,8	405,4	173,6
Consulta quatro tabelas						
	Hibernate	JDBC	Hibernate	JDBC	Hibernate	JDBC
	7543	1753	2864	765	3564	1743
	342	296	432	154	432	97
	218	431	275	157	231	254
	154	98	198	98	207	153
	123	164	231	205	198	128
Média	1676	548,4	800	275,8	926,4	475

Tabela 6: Consultas MySQL, quarto plano de testes

Na tabela 5 e 6 são apresentadas três baterias de testes seguindo o terceiro e quarto plano de teste, respectivamente, medindo o tempo da consulta usando o *Hibernate* e JDBC. Foram efetivadas três tipos de consultas a primeira em uma tabela simples, a segunda em duas tabelas simultâneas e a terceira em quatro tabelas. Observa-se que o JDBC obtém um melhor desempenho em relação ao *Hibernate*.

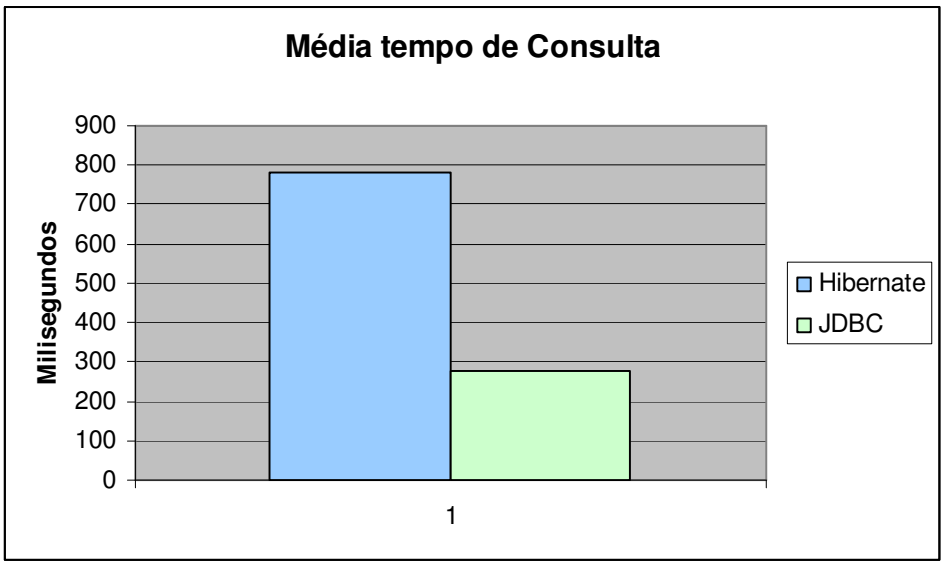

Gráfico 1: Média tempo de consultas MySQL, terceiro plano de teste.

O gráfico acima mostra a diferença em milisegundos do tempo médio de consulta das aplicações que utilizam o *Hibernate* e o JDBC. O gráfico é referente ao terceiro plano de teste.

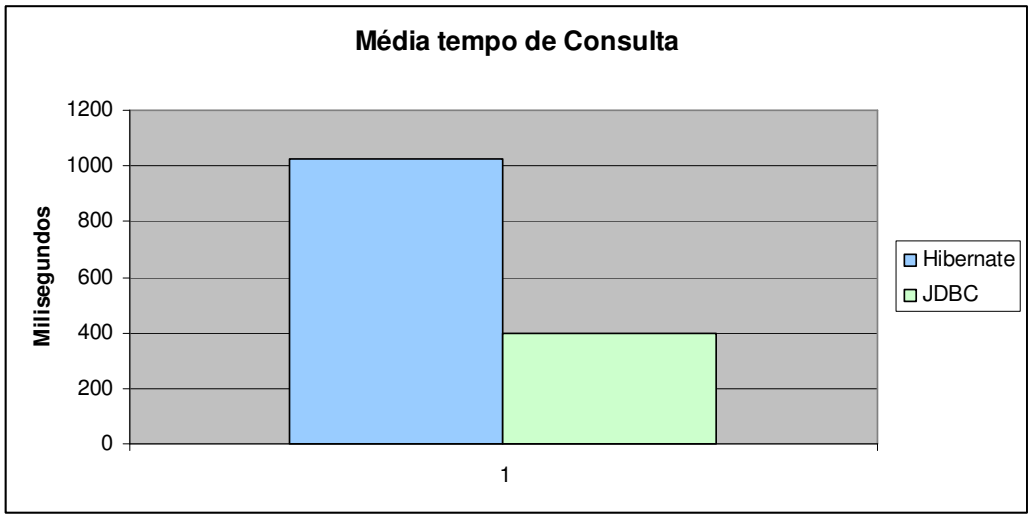

Gráfico 2: Média tempo de consultas MySQL, quarto plano de teste.

O gráfico acima mostra a diferença em milisegundos do tempo médio de consulta das aplicações que utilizam o *Hibernate* e o JDBC. O gráfico é referente ao quarto plano de teste.

PostgreSQL

Consulta uma tabela								
	Hibernate	JDBC		Hibernate	JDBC		Hibernate	JDBC
	5391	1031		4234	141		5000	3313
	0	15		0	16		0	31
	15	16		15	16		0	15
	0	15		16	438		16	16
	16	16		15	31		0	16
Média	1084,4	218,6		856	128,4		1003,2	678,2
Consulta duas tabelas								
	Hibernate	JDBC		Hibernate	JDBC		Hibernate	JDBC
	1454	438		1531	188		1593	296
	16	32		0	47		16	47
	15	46		31	31		15	47
	31	47		31	47		31	63
	15	47		16	31		16	47
Média	306,2	122		321,8	68,8		334,2	100
Consulta quatro tabelas								
	Hibernate	JDBC		Hibernate	JDBC		Hibernate	JDBC
	1953	857		2422	344		1923	751
	422	172		422	187		406	156
	406	266		422	250		391	250
	251	265		250	234		267	171
	251	156		249	156		266	218
Média	656,6	343,2		753	234,2		650,6	309,2

Tabela 7: Consultas PostgreSQL, terceiro plano de teste

Consulta a uma tabela

	Hibernate	JDBC	Hibernate	JDBC	Hibernate	JDBC
	6334	1282	5678	523	6351	3813
	32	31	22	45	12	45
	32	31	34	31	31	29
	31	31	21	71	12	32
	32	31	21	29	31	32
Média	**1292,2**	**281,2**	**1155,2**	**139,8**	**1287,4**	**790,2**

Consulta duas tabelas

	Hibernate	JDBC	Hibernate	JDBC	Hibernate	JDBC
	1865	654	2078	243	2421	567
	31	47	29	61	36	75
	45	52	41	41	29	64
	44	65	56	68	21	78
	32	52	31	41	32	52
Média	**403,4**	**174**	**447**	**90,8**	**507,8**	**167,2**

Consulta quatro tabelas

	Hibernate	JDBC	Hibernate	JDBC	Hibernate	JDBC
	3546	1564	4837	721	3621	953
	534	189	764	342	742	284
	475	306	583	429	642	463
	376	467	521	542	374	274
	321	286	352	274	385	463
Média	**1050,4**	**562,4**	**1411,4**	**461,6**	**1152,8**	**487,4**

Tabela 8: Consultas PostgreSQL, quarto plano de teste

Na tabela 7 e 8 são apresentadas três baterias de testes, seguindo o terceiro plano e o quarto plano de teste, respectivamente, medindo o tempo da consulta usando o *Hibernate* e JDBC. Foram efetivadas três tipos de consultas a primeira em uma tabela simples, a segunda em duas tabelas simultâneas e a terceira em quatro tabelas. Apesar do *Hibernate* conseguir ser mais rápido depois da primeira consulta, a melhor média de desempenho é obtida pelo JDBC.

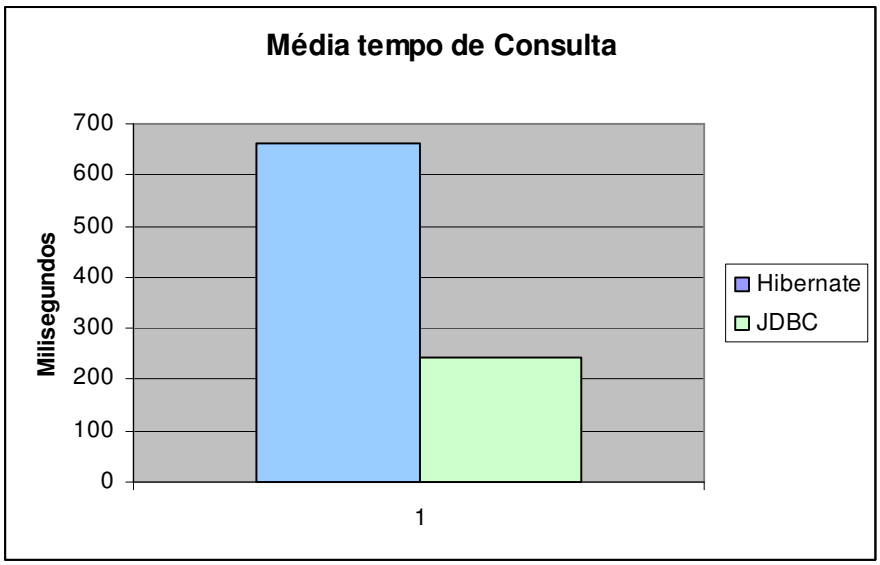

Gráfico 3: Média tempo de consultas PostgreSQL, terceiro plano de teste.

O gráfico 3 mostra a diferença em milisegundos do tempo médio de consulta das aplicações que utilizam o *Hibernate* e o JDBC. O gráfico é referente ao terceiro plano de teste.

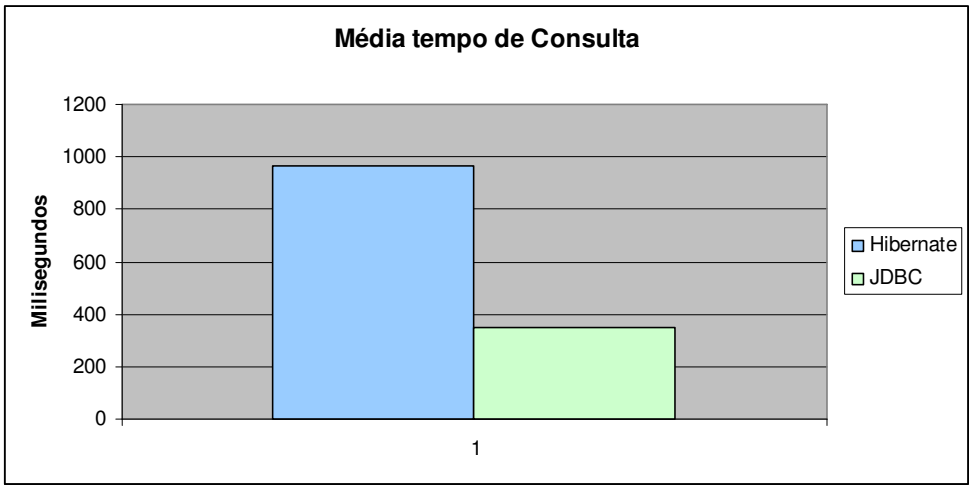

Gráfico 4: Média tempo de consultas PostgreSQL, quarto plano de teste.

O gráfico 4 mostra a diferença em milisegundos do tempo médio de consulta das aplicações que utilizam o *Hibernate* e o JDBC. O gráfico é referente ao quarto plano de teste.

Inclusão

	MySQL		PostgreSQL	
	Hibernate	JDBC	Hibernate	JDBC
	78	31	672	78
	16	32	0	47
	0	31	0	0
	0	16	0	16
	0	16	0	31
Média	18,8	25,2	134,4	34,4
	Hibernate	JDBC	Hibernate	JDBC
	46	31	735	109
	0	16	0	31
	0	16	16	47
	109	0	16	0
	0	16	0	16
Média	31	15,8	153,4	40,6
	Hibernate	JDBC	Hibernate	JDBC
	94	16	345	94
	15	16	78	31
	0	0	0	15
	0	15	31	16
	0	31	0	16
Média	21,8	15,6	90,8	34,4

Tabela 9: Testes de Inclusão

Na tabela 9 é apresentada três baterias de testes medindo o tempo da inclusão usando o *Hibernate* e JDBC. Observa-se que o JDBC obtém um melhor desempenho em relação ao *Hibernate*.

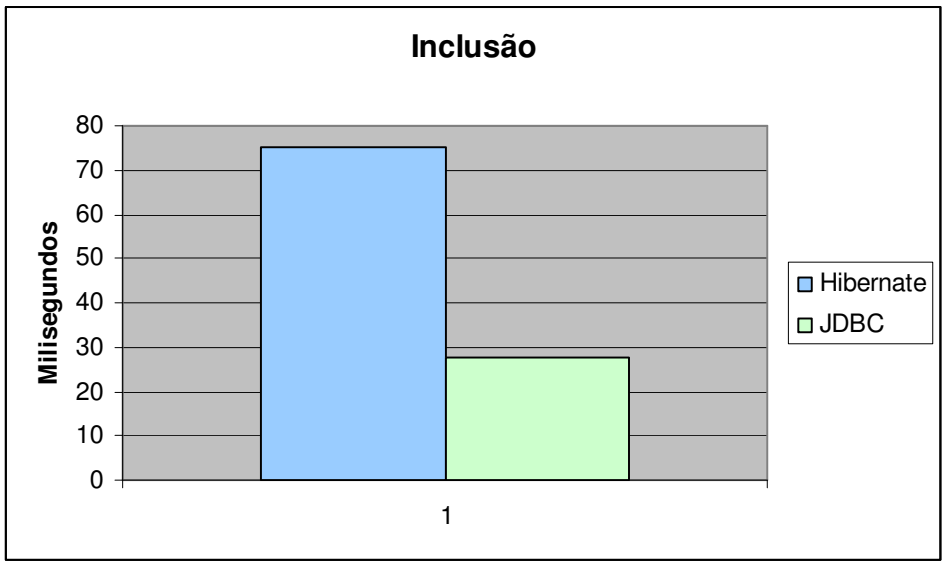

Gráfico 5: Média tempo de inclusão.

O gráfico 5 mostra a diferença em milisegundos do tempo médio de inclusão das aplicações que utilizam o *Hibernate* e o JDBC. O gráfico é referente ao quinto plano de teste.

Exclusão

	MySQL		PostgreSQL	
	Hibernate	JDBC	Hibernate	JDBC
	93	32	157	56
	0	31	16	31
	0	0	0	47
	16	0	0	15
	0	16	16	16
Média	21,8	15,8	37,8	33
	Hibernate	JDBC	Hibernate	JDBC
	47	15	63	32
	0	16	0	0
	0	0	0	16
	0	16	0	15
	0	0	16	15
Média	9,4	9,4	15,8	15,6
	Hibernate	JDBC	Hibernate	JDBC
	31	16	109	47
	0	0	0	16
	0	16	0	31
	16	0	0	0
	0	16	16	15
Média	9,4	9,6	25	21,8

Tabela 10: Testes de Exclusão.

Na tabela 10 é apresentada três baterias de testes medindo o tempo da exclusão usando o *Hibernate* e JDBC. Observa-se que o JDBC obtém um melhor desempenho em relação ao *Hibernate*.

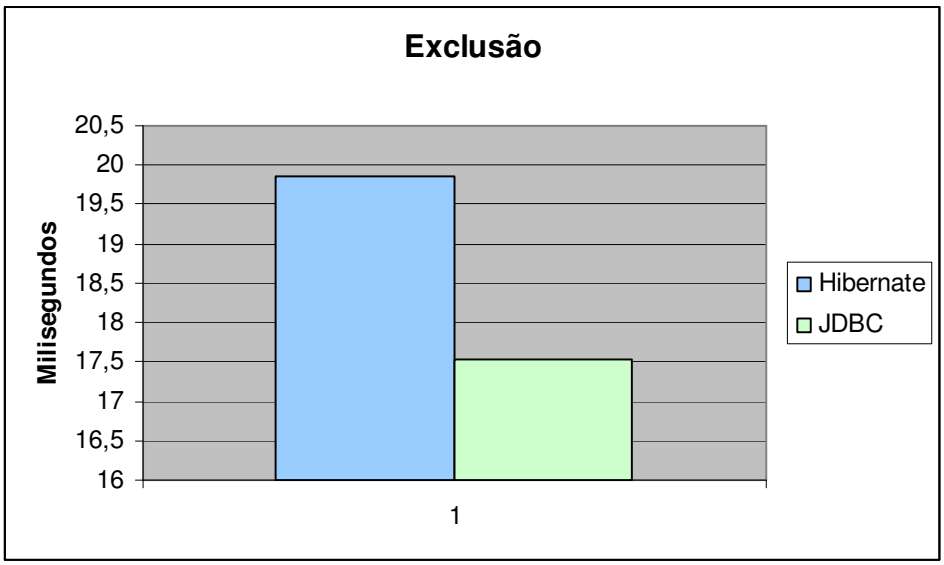

Gráfico 6: Média tempo de exclusão.

O gráfico acima mostra a diferença em milisegundos do tempo médio de exclusão das aplicações que utilizam o *Hibernate* e o JDBC. . O gráfico é referente ao sexto plano de teste.

Cabe ressaltar que os testes acima foram realizados medindo o tempo antes e depois da operação de banco (consulta, inclusão e exclusão), dentro do código das aplicações.

9 CONCLUSÃO

Neste trabalho foram apresentadas as diferenças potenciais de incompatibilidade de paradigma, entre o modelo relacional e o modelo de objetos. A solução para obter melhor aproveitamento das tecnologias orientada a objetos e relacional é criar uma ponte entre os dois mundos. Verifica-se que o mapeamento objeto/relacional soluciona de maneira bastante eficiente os problemas de incompatibilidade, apresentando diversas técnicas de mapeamento e de busca visando um melhor desempenho.

Foram implementadas duas aplicações, uma usando o SQL/JDBC e outra utilizando o *framework Hibernate*. Ambas as aplicações fazem uso de uma modelagem simples de banco de dados e implementam funcionalidades básicas como inclusão, alteração, exclusão e consulta de dados.

Observa-se que a aplicação que utiliza o *Hibernate* atingiu uma redução de código significativa relacionado a camada de persistência, além de conseguir a interdependência de dados, tornando a aplicação portável para diferentes SGBDs.

Com a estratégia adotada pode-se afirmar que com o uso do *Hibernate*, há um ganho de produtividade, onde o desenvolvedor concentra seus esforços na lógica da aplicação.

Porém há uma perda de desempenho, segundo os testes realizados, já que aplicação que utiliza o *Hibernate* possui uma camada a mais, que se sobrepõe, acima do JDBC. Cabe ressaltar que não foi utilizado nenhuma otimização do *Hibernate*, como ativar cachê secundário ou utilizar opções de busca como *lazy loading* (busca preguiçosa).

Conforme os autores Bauer e King, o *Hibernate* não provê a solução para todos os problemas e não pode ser considerada a única solução para a tarefa de persistência. Apesar de reduzir em 95 por cento o trabalho relacionado a camada de persistência, os 5 por cento restantes exige bastante esforço, pois se trata de consultas complexas e específicas. Nesses casos aconselha-se o uso do SQL/JDBC para promover mais desempenho maior nas consultas mais complicadas.

Outro fator relevante relacionado a implementação ou migração de sistemas para obter maior aproveitamento do *Hibernate*, o grupo de desenvolvedores e arquitetos devem possuir certa experiência ou um aprendizado bem profundo da tecnologia.

Podemos concluir que apesar do *Hibernate* oferecer um ganho de produtividade na camada de persistência, nota-se uma perda de desempenho em relação a aplicação que faz uso do SQL/JDBC, pois a mesma mantém uma comunicação direta com o banco de dados.

Considerando o tempo de desenvolvimento, o *Hibernate* é mais vantajoso em relação ao SQL/JDBC, porém cabe analisar se a performance é um fator critico, pois alguns milisegundos podem não fazer diferença para o usuário final. É preciso quantificar se essa demora é “aceitável”.

10 REFERÊNCIAS BIBLIOGRÁFICAS

ABREU M.; MACHADO F.; **Projeto de Banco de Dados: Uma visão prática** –1° Ed. Érica, 1996.

ALVIM P.; OLIVEIRA H. Hibernate mapeamento OOxSGBDR. **SQL Magazine**, Ed 2, p. 40-45, 2004.

AMBLER S.; **The Design of a Robust Persistence Layer For Relational Databases.** Disponível em: http://www.ambysoft.com/downloads/persistenceLayer.pdf Acesso em: 30/01/2007.

BAUER C.; KING G.; **Java Persistence with Hibernate** – 1º Ed. Manning, 2007.

BAUER C.; KING G.; **Hibernate em ação** – 1º Ed. Ciência Moderna, 2005.

BOAGLIO F. Hibernate, **SQL Magazine**, Ed. 17, p.38-46, 2005.

CALÇADO P.; **Arquitetura de Camadas em Java EE.** Disponível em: http://www.mundojava.com.br/NovoSite/15materiacapa.shtml Acesso em: 05/02/2007.

FERNANDES R.; LIMA G.; **Hibernate Annotations.** Disponível em: http://www.jeebrasil.com.br/tags/hibernate. Acesso em 10/10/2007.

FONSECA, G; Análise Comparativa de Sistemas de Gerenciamento de Banco de Dados. 2006. 52f. Monografia – Faculdade de Jaguariúna, Jaguariúna.

GALHARDO R.; LIMA G.; **Introdução ao Hibernate**. Disponível em: http://www.jeebrasil.com.br/mostrar/4 Acesso em: 05/11/2006.

GONÇALVES K.; Teste de Software em aplicações de banco de dados relacional. 2003. 61f. Tese (Mestrado). Universidade Estadual de Campinas, Campinas.
HEUSER C.; **Projeto de Banco de Dados.** – 5° Ed. Sagra Luzzatto, 2004.

Hibernate; **Hibernate.org**. Disponível em: http://www.hibernate.org/ Acesso em: 12/09/2006.

IMARTERS; **Os conceitos primordiais em banco de dados.** Disponível em: http://www.devmedia.com.br/articles/viewcomp.asp?comp=3173&hl=banco%20de%20dados. Acesso em 25/08/2007.

JÚNIOR H. F. de A.; **Mapeando Objetos para Bancos de Dados Relacionais: técnicas e implementações.** Disponível em: http://www.mundooo.com.br/php/mooartigos.php?pa=showpage&pid=19. Acesso em: 23/10/2007.

LINHARES M.; **Introdução ao Hibernate**. Disponível em: http://www.guj.com.br/content/articles/hibernate/intruducao_hibernate3_guj.pdf Acesso em: 23/09/2006.

LOZANO F. Hibernate na Web. **Java Magazine**, Ed. 33, p. 30-45, 2006.

LOZANO F. Persistência com Hibernate. **Java Magazine**, Ed. 28, p. 18-28, 2006.

NASCIMENTO G. Fast Hibernate 3. **Mundo Java**, Ed. 17, p. 20-25, 2006.
Oliveira, C.; **SQL:curso prático** – 1° Ed. Novatec, 2002.

PEAK P.; Heudecker N.; **Hibernate Quickly** – 1° Ed. Manning, 2006

PINHEIRO J.; Um framework para persistência de objetos em banco de dados relacionais. 2005. 207f. Tese (Mestrado) – Universidade Federal Fluminense, Niterói.

RUMBAUGH, James. et al. **Modelagem e Projetos Baseados em Objetos**. Trad. Dalton Conde de Alencar. Rio de Janeiro: Editora Campus, 1994.

SILBERSCHATZ A.; KORTH H.; SUDARSHAN S.; **Sistema de Banco de Dados** – 3° Ed. Makron Books, 1999.