UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

DRIVER JDBC PARA CONSULTAS EM BANCOS DE DADOS HETEROGÊNEOS DISTRIBUÍDOS FRAGMENTADOS HORIZONTALMENTE

JACSON GONÇALVES

BLUMENAU
2007

2007/2-19

JACSON GONÇALVES

DRIVER JDBC PARA CONSULTAS EM BANCOS DE DADOS HETEROGÊNEOS DISTRIBUÍDOS FRAGMENTADOS HORIZONTALMENTE

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciências da Computação - Bacharelado.

Prof. Adilson Vahldick - Orientador

BLUMENAU
2007

2007/2-19

DRIVER JDBC PARA CONSULTAS EM BANCOS DE DADOS HETEROGÊNEOS DISTRIBUÍDOS FRAGMENTADOS HORIZONTALMENTE

Por

JACSON GONÇALVES

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Prof. Adilson Vahldick, Especialista, FURB
Membro:	Prof. Marcel Hugo, Mestre – FURB
Membro:	Prof. Alexander Roberto Valdameri, Mestre – FURB

Blumenau, 07/12/2007

Dedico este trabalho a toda minha família, principalmente meus pais, amigos e meu orientador que acreditou no potencial deste trabalho.

AGRADECIMENTOS

A Deus, pelo seu imenso amor e graça.

Aos meus pais, Dálci Carmen Gonçalves e Aldo José Gonçalves que ajudaram a custear meus estudos, sempre incentivando para que o futuro fosse certo e promissor.

Aos meus amigos, pelos empurrões e cobranças.

Ao meu orientador, Adilson Vahldick, por ter me incentivado e acreditado na conclusão deste trabalho.

A sorte favorece a mente mais bem preparada.

Albert Einstein

RESUMO

Este trabalho apresenta a especificação e implementação de um *driver* JDBC que tem por objetivo realizar as conexões e execução de consultas em bancos de dados heterogêneos, local ou remotamente distribuídos, utilizando a técnica de fragmentação horizontal de dados. O documento XML utilizado pelo *driver* para realização das consultas, chamado de Esquema Conceitual Global (ECG), contém o mapeamento das tabelas de cada banco de dados envolvido na fragmentação. Este documento é desenvolvido a partir de uma ferramenta de construção e edição de documentos XML, tornando-se parte integrante deste trabalho.

Palavras-chave: Fragmentação horizontal. JDBC. Banco de dados distribuído.

ABSTRACT

This work presents the specification and implementation of JDBC driver, that has for objective to carry through the connections and execution of queries in heterogeneous data bases, local or remotely distributed, using the technique of data horizontal fragmentation. The XML document used for driver for execution of queries, call of Global Conceptual Schema (ECG), contains the mapping of tables of each involved data base in the fragmentation. This document is developed from a tool of construction and edition of XML document, having become integrant part of this work.

Palavras-chave: Horizontal fragmentation. JDBC. Distributed database.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

API – *Application Programming Interface*

BD – Banco de Dados

BDD – Banco de Dados Distribuído

DBA – *Data Base Administrator*

DTD – *Document Type Definition*

ECG– Esquema Conceitual Global

JDBC – *Java Database Connectivity*

MER – Modelo Entidade Relacionamento

SBDD – Sistema de Banco de Dados Distribuído

SGBD – Sistema Gerenciador de Banco de Dados

SQL – *Structure Query Language*

UML – *Unified Modeling Language*

URL – Uniform Resource Locator

XML - eXtensible Markup Language

SUMÁRIO

1 INTRODUÇÃO

> A tecnologia de sistemas de bancos de dados distribuídos é um dos principais desenvolvimentos recentes na área de sistemas de bancos de dados. Muitos sustentam que nos próximos dez anos os sistemas de gerenciamento de bancos de dados centralizados serão uma "antiga curiosidade", e a maioria das organizações migrará para gerenciadores de bancos de dados distribuídos. (STONEBRAKER, 1988, p. 189).

Sabe-se que, dez anos após Stonebraker afirmar isto, no final da década de 90 poucos sistemas utilizavam a arquitetura de Bancos de Dados Distribuídos (BDDs). Porém, a grande curiosidade no assunto e interesse nesta área da tecnologia, tanto em universidades quanto em empresas desenvolvedoras de software, oferecem sustento a esta afirmação. Atualmente, as grandes corporações, com várias filiais distribuídas em diversos lugares do mundo, necessitam ter acesso a dados umas das outras de forma rápida, segura e íntegra, pois a falta de informações como um todo e da visão geral administrativa de uma grande corporação, pode pôr fim às ambições competitivas da empresa.

Uma das motivações importantes por trás do uso de Sistemas de Bancos de Dados Distribuídos é o desejo de integrar os dados operacionais de um empreendimento e proporcionar acesso centralizado, e portanto controlado, a esses dados (ÖZSU; VALDURIEZ, 2001, p. 1).

Um Sistema de Banco de Dados Distribuído (SBDD) é composto de uma rede de BDs locais, armazenados em diversas máquinas, devendo ser visto pelo usuário como um único BD lógico, instalado em uma única máquina. BDDs consistem numa coleção de sítios, conectados através de uma rede de comunicação, onde os sítios concordam em cooperar, de forma que um usuário em qualquer sítio acessa um dado na rede, de forma transparente, como se o dado estivesse no próprio sítio do usuário. Cada sítio possui seu próprio banco de dados dentro de um ambiente descentralizado (CERÍCOLA, 1995, p. 287).

Para a execução de consultas em BDDs é necessário que as relações em um esquema sejam decompostas em fragmentos menores, sendo que cada um deles é tratado como uma unidade, permitindo que várias consultas sejam executadas de forma concorrente. Para isso, optou-se por dividir as relações de forma horizontal.

Özsu e Valduriez (2001, p. 13) afirmam que na fragmentação horizontal, uma relação é particionada em um conjunto de sub-relações, cada uma das quais tem um subconjunto das tuplas (linhas) da relação original. Este trabalho visa disponibilizar um recurso de acesso a dados fragmentados onde, por exemplo, cada empresa possa visualizar dados distribuídos

entre as filiais e ter uma visão global destes dados, parecendo de forma transparente para o usuário.

Uma das características dos BDDs é a heterogeneidade. Isso implica em SGBDs distintos, ou mesmo projetos diferentes de BDs. O *driver* JDBC desenvolvido é responsável pelo acesso às bases de dados e mapeamento das tabelas de cada BD, servindo como uma ponte de comunicação entre a aplicação e o SGBD.

Santos (2005, p. 5), afirma que JDBC é uma API que suporta as funcionalidades básicas da *Structured Query Language* (SQL) e proporciona integração do código Java com sistemas de BDs.

Os mapeamentos são armazenados em documentos XML, que constituem o Esquema Conceitual Global (ECG), servindo de um artefato indispensável para a realização de consultas globais. O ECG contém o mapeamento global, ou seja, as informações comuns a todos os BDs envolvidos e o mapeamento local, cujas informações são específicas de cada BD. Estes documentos são criados a partir de uma ferramenta construída especificamente para geração de arquivos XML, que são interpretados pelo *driver* na realização das consultas. Foram criados arquivos XML *Schema* para definição de atributos, elementos e tipos de dados que o documento XML pode conter, e em que local do documento eles devem aparecer.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver um *driver* JDBC que servirá como uma ponte entre a aplicação e as bases de dados, permitindo realizar consultas em um ou mais BDs, independente do fabricante e local onde estejam alojados.

Os objetivos específicos do trabalho são:

a) criar um *driver* JDBC capaz de executar consultas em BDDs;
b) desenvolver uma aplicação para construção do ECG.

1.2 ESTRUTURA DO TRABALHO

Este trabalho está estruturado em quatro capítulos que estão descritos a seguir:

O primeiro capítulo contextualiza e justifica o desenvolvimento do trabalho, separado em introdução, objetivos e estrutura do trabalho.

No segundo capítulo é disponibilizada a fundamentação teórica necessária para um razoável conhecimento nas tecnologias e componentes utilizados no desenvolvimento do trabalho.

O terceiro capítulo tem como foco o desenvolvimento do *driver* JDBC e da ferramenta de criação de ECGs, descrevendo suas principais funcionalidades, bem como suas especificações e implementações.

O quarto capítulo apresenta as conclusões finais, mostrando os resultados obtidos, suas limitações e sugestões para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são explanados os fundamentos teóricos necessários para compreensão do trabalho. Na última seção são descritos alguns trabalhos correlatos.

2.1 BANCO DE DADOS DISTRIBUÍDO

Um SBDD existe quando um banco de dados integrado logicamente é fisicamente distribuído sobre diferentes nós de computação interligados por uma rede (COUCEIRO; BARRENECHA, 1984, p. 3).

Casanova (1985, p. 1) afirma que se tem um SBDD quando o armazenamento de um banco de dados pode ser dividido ao longo dos nós de uma rede de comunicação, sem que o usuário perca a visão integrada do banco.

> Podemos definir um banco de dados distribuído como uma relação de vários bancos de dados, logicamente inter-relacionados, distribuídos por uma rede de computadores. Um sistema de gerenciamento de banco de dados distribuído é definido então como um sistema de software que permite o gerenciamento do banco de dados distribuído e que torna a distribuição transparente para os usuários. (ÖZSU; VALDURIEZ, 2001, p. 5).

Os sistemas de BDs têm sido criados de forma independente, sem preocupação com as possíveis integrações. A integração de BDs heterogêneos tem início com o processo de conversão de esquemas, ou seja, cada BD possui suas próprias tabelas, inclusive nomenclatura. Neste processo são estabelecidas equivalências entre o modelo de origem e os modelos de destino.

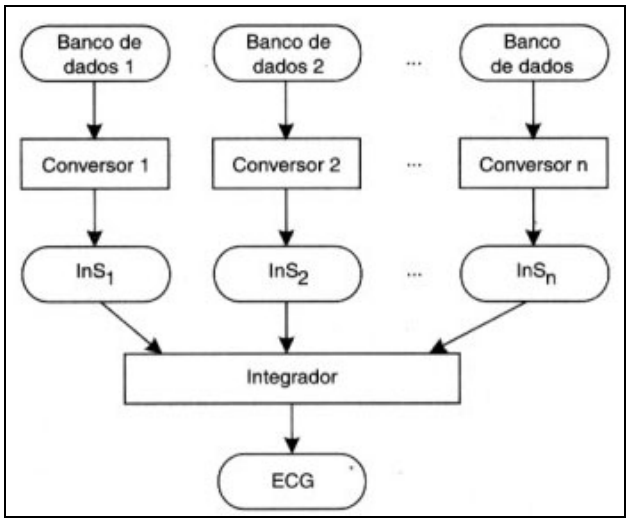
Segundo Date (1988, p. 276), "[...] um banco de dados distribuído é aquele que não é armazenado em sua inteireza em uma única localização física mas, ao contrário, está espalhado por uma rede de localizações geograficamente dispersas e ligadas por meio de *links* de comunicação."

> A integração de esquema é o processo de identificar os componentes de um banco de dados relacionados uns aos outros, selecionar a melhor representação para o esquema conceitual global e, finalmente, integrar os componentes de cada esquema intermediário. (SHETH et al,. 1988 apud ÖZSU; VALDURIEZ, 2001, p. 574).

Özsu e Valduriez (2001, p. 570), afirmam que "a integração de bancos de dados envolve o processo pelo qual as informações de bancos de dados participantes podem ser

integradas conceitualmente para formar uma única definição coesa de um banco de dados múltiplo".

O ECG é a relação entre todas as bases de dados envolvidas e contém as informações globais, comuns a todos os bancos. Como mostra a figura 1, o esquema de cada banco é convertido e alocado num integrador. A conversão só é necessária se os BDs componentes forem heterogêneos, ou seja, que contenham as bases de dados ou fabricantes diferentes.



Fonte: Özsu e Valduriez (2001, p. 570).

Figura 1 – Processo de integração de BD

A localização das tabelas de cada BD é identificada na decomposição de consultas, que particiona a consulta distribuída em uma consulta sobre relações globais. As informações necessárias para essa transformação são encontradas no ECG que descreve as relações globais (ÖZSU; VALDURIEZ, 2001, p. 213).

2.1.1 Processamento de consultas globais

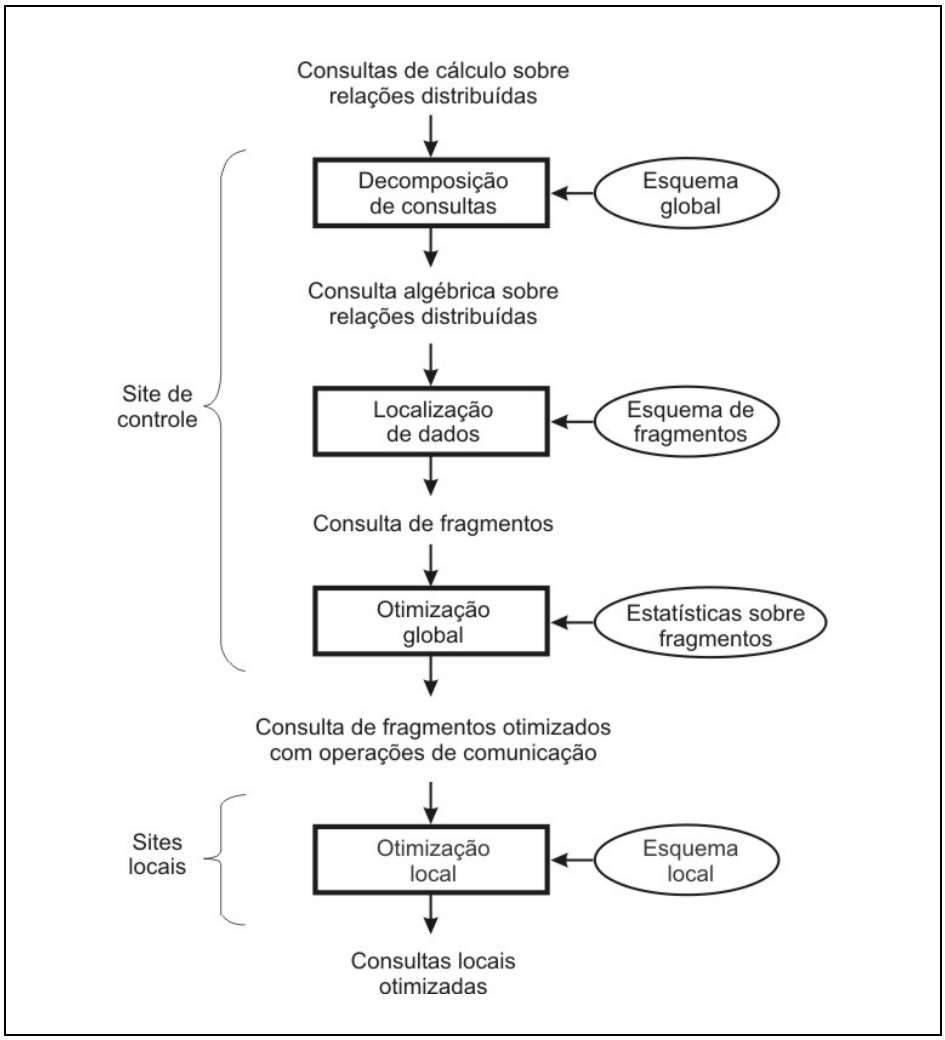
Segundo Özsu e Valduriez (2001, p. 200), o processamento de consultas é muito mais complexo em ambientes distribuídos que em ambientes centralizados, devido ao fato de que um número maior de parâmetros afeta o desempenho de consultas distribuídas. Além disso, se forem acessados muitos sítios, o tempo de resposta da consulta poderá se tornar muito alto. As consultas distribuídas são executadas sobre relações globais distribuídas, onde a distribuição de dados está oculta.

Özsu e Valduriez (2001, p. 211) afirmam que existem quatro camadas principais para o

mapeamento, juntamente com uma seqüência otimizada de operações locais, onde cada uma atua sobre um banco de dados local. Estas camadas executam as seguintes funções:

a) decomposição de consultas: decompõe a consulta de cálculo distribuído em uma consulta sobre relações globais. As informações necessárias para essa transformação são encontradas no ECG que descreve as relações globais;
b) localização de dados: tem por função principal, localizar os dados da consulta com o uso de informações de distribuição de dados. Esta camada determina quais fragmentos estão envolvidos na consulta e transforma a consulta distribuída em uma consulta sobre fragmentos;
c) otimização de consultas globais: envolve operações de comunicação que minimizam uma função de custo, que se refere a recursos como espaço em disco, operações de E/S, CPU e comunicação;
d) otimização de consultas locais: executada por todos os sítios que têm fragmentos envolvidos na consulta. Cada consulta é otimizada com o uso do esquema local do sítio.

Na figura 2 é apresentado um esquema genérico em camadas para o processamento de consultas, no qual cada camada resolve um problema bem definido.



Fonte: Özsu e Valduriez (2001, p. 212).

Figura 2 – Esquema genérico de camadas para processamento de consultas distribuídas

Segundo Casanova (2005), "o processamento de consultas sobre BDDs corresponde a tradução de pedidos, formulados em uma linguagem de alto nível, para seqüências de operações elementares sobre os dados armazenados nos vários bancos locais". Este, por sua vez, difere do caso centralizado em três aspectos básicos:

a) o diretório de dados, em geral, é distribuído e a sua forma de armazenamento afeta fortemente a eficiência do processador de consultas;
b) como o banco é distribuído, uma relação do esquema conceitual pode estar fragmentada e replicada ao longo da rede. O processador deverá selecionar os fragmentos, localizar as cópias apropriadas e eventualmente movê-las para que a consulta possa ser processada;
c) se o sistema for heterogêneo, o processador deverá, ainda, efetuar traduções entre modelos de dados distintos.

Em um ambiente centralizado, as estratégias de execução de consultas podem ser

expressas em uma extensão de álgebra relacional, onde a principal função do processador de consultas é escolher a melhor consulta de álgebra relacional entre todas as equivalentes. Em um sistema distribuído, a álgebra relacional não é suficiente para expressar as estratégias de execução, que deve ser complementada com operações para intercâmbio de dados entre os sítios (ÖZSU; VALDURIEZ, 2001, p. 204).

2.2 FRAGMENTAÇÃO HORIZONTAL

Cerícola (1995, p. 298) afirma que a fragmentação horizontal de dados permite que uma ou mais linhas de um arquivo sejam armazenadas em um nó da rede, e as demais em locais diferentes.

A fragmentação horizontal de dados utiliza-se da união dos fragmentos, ao contrário da fragmentação vertical, que se baseia na junção dos mesmos. Diferente da replicação de dados, na fragmentação não há existência de cópias de registros ou tabelas. Por exemplo, a união dos clientes de todas as filiais de uma empresa forma uma relação global. Porém, esta relação pode ser dividida em *n* fragmentos diferentes, cada um dos quais formado pelas tuplas dos clientes de uma filial em particular.

O grau de fragmentação que um BD deve ser submetido representa uma decisão que afeta o desempenho do processamento de consultas. Este grau pode variar entre nenhuma fragmentação até o nível de fragmentação que divide a relação em tuplas individuais. A fragmentação é definida através de regras de fragmentação que podem ser expressas com operações relacionais. Um exemplo disso são as consultas distribuídas que armazenam subconjuntos disjuntos, chamados fragmentos, cada um deles alocado em um sítio diferente. A localização dos dados para uma relação fragmentada é a união desses fragmentos. Estas regras se resumem basicamente em junções e reduções (ÖZSU; VALDURIEZ, 2001, p. 116).

2.3 DRIVER JDBC

Segundo Ramon (2001, p. 7), JDBC é uma camada de abstração que permite a um programa Java utilizar uma interface padrão para acessar um BD relacional através da SQL.

A API JDBC é um conjunto de classes Java que definem as conexões ao BD permitindo ao desenvolvedor manipular os resultados das consultas realizadas nestes bancos.

Hübner (2005, p. 4), referindo-se ao JDBC, afirma que uma das vantagens da sua utilização é que a API para programação do sistema é a mesma para qualquer SGBD, não havendo necessidade de desenvolver aplicações para um BD específico.

Thompson (2002, p. 125), referindo-se ao padrão Java de conectividade a BDs, "[...] é a interface que possibilita as aplicações Java acessarem bancos de dados relacionais e demais arquivos de dados."

Ramon (2000, p. 14) descreve que um programa Java pode utilizar simultaneamente vários *drivers* JDBC distintos, acessando bancos de dados heterogêneos, onde todos os *drivers* devem ser registrados na classe `DriverManager`, pertencente ao pacote `java.sql`, que estabelece a conexão, controla os *logins* e as mensagens entre os BDs. Contudo, todos os *drivers* devem possuir um bloco estático onde um objeto de sua classe é instanciado e registrado.

O Quadro 1 mostra um exemplo de conexão com o banco de dados a partir das informações do XML, gravando as tabelas retornadas da execução da consulta.

```
//Busca os configurações de conexão banco
BancoDados bd = getCadastroBanco().getBancoDados().get(nomeBanco);
//Faz a conexão com o banco
getConn().conectar(bd.getDriver(), bd.getUrl(), bd.getUsuario(),
                   bd.getSenha());
//Busca as tabelas do banco
ResultSet rs = getConn().executaQuery("show tables");
DefaultMutableTreeNode filhos;
//Adiciona as tabelas na JTree
try{
  HashMap tabelas = new HashMap();
  while(rs.next()){
    String tabela = rs.getString(1);
    filhos        = new DefaultMutableTreeNode(tabela);
//Buscar os atributos e armazenar
    ResultSet rsAtrib= getConn().executaQuery("desc "+tabela);
    ArrayList atributos = new ArrayList();
    while(rsAtrib.next()){
       atributos.add(rsAtrib.getString(1));
    }
    rsAtrib.close();
    tabelas.put(tabela, atributos);
    getCadastroBanco().addTabela(nomeBanco, tabelas);
    node.add(filhos);
  }
  rs.close();
}catch(Exception e){
}
```

Quadro 1 – Exemplo de conexão com BDDs

A interface `Driver é` responsável por registrar o *driver* JDBC e a carregar a classe `DriverManager`. Esta classe estabelece a conexão e é responsável pelo gerenciamento dos *drivers* de todos os BDs disponíveis. A interface `Connection` define um objeto para receber a conexão estabelecida entre a aplicação e a base de dados. A interface `Statement` é utilizada para execução de uma instrução SQL através de uma `Connection`. A interface `ResultSet` proporciona o acesso aos dados provindos da execução destas instruções.

A figura 3 mostra o diagrama de classes com a relação entre as principais classes e interfaces de um *driver* JDBC.

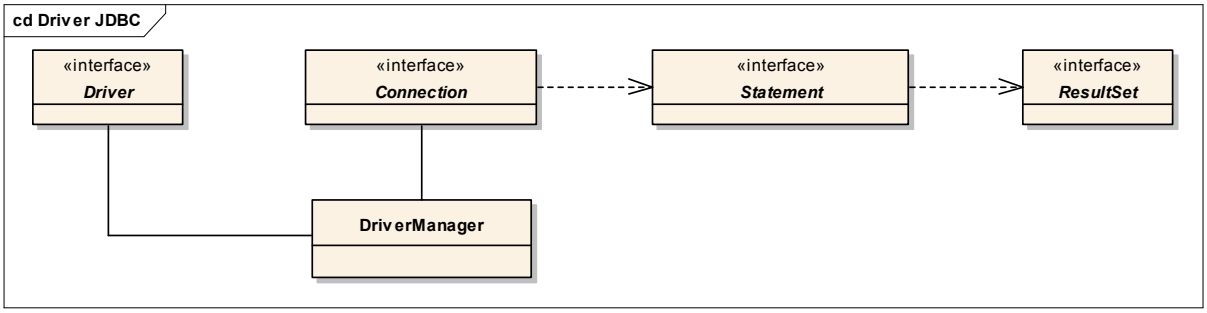


Figura 3 – Diagrama de classes de um *driver* JDBC

2.4 XML

XML é uma linguagem de marcação, e assim como as linguagens de formatação, requer aplicativos que possam buscar as informações, analisando-as e exibindo-as. Entretanto, XML fornece um formato padrão para descrição de dados estruturados, e não mais para a formatação destes mesmos dados.

Segundo Tesch Júnior (2002, p. 17), XML descreve uma classe de dados chamada documentos XML. Os documentos XML são formados por unidades de armazenamento chamadas entidades. Os dados analisados sintaticamente são constituídos por caracteres, alguns dos quais formam dados de caracteres e parte dos quais formam a marcação. A marcação codifica uma descrição do armazenamento e da estrutura lógica do documento. O processador XML é usado para ler documentos XML e fornece acesso a seus conteúdos e estruturas.

Marchal (2000, p. 46), definiu XML como sendo um "conjunto de padrões para troca e publicação de informações de uma forma estruturada". A principal vantagem da linguagem se

concentra na estrutura do documento, possibilitando editar e publicar estes documentos em diferentes meios.

Um dos pontos fortes da XML é que possui a descrição da forma de apresentação do documento separada do conteúdo do mesmo. Documentos XML fornecem uma representação estruturada de dados que pode ser implementada a partir de um grande número de aplicações. A partir da definição conferida pela W3C, garante-se que os dados estruturados serão uniformes e independentes de aplicações ou distribuidores.

XML é um padrão de representação de documentos armazenado em arquivos texto, legível por pessoas e programas, facilmente adaptado à novas aplicações e estruturado em forma de árvore (HÜBNER, 2006, p. 5).

Um dos maiores objetivos da linguagem XML foi possibilitar que os criadores de páginas *web*, descrevessem e manipulassem suas próprias *tags*, superando as limitações impostas pela linguagem HTML. Ou seja, a XML representa um aperfeiçoamento da HTML, independente de plataforma, designada por um padrão aberto, onde o documento XML desenvolvido pertence ao seu próprio criador.

O XML provê um padrão que pode codificar o conteúdo, as semânticas e os esquemas para uma grande variedade de aplicações, independente da complexidade, como um registro de dados que seria o resultado de uma consulta a um BD ou uma apresentação gráfica de interface de aplicação com o usuário (FURTADO JÚNIOR, 2003).

O exemplo de XML representado no Quadro 2 consiste de um elemento raiz "`shiporder`". Este elemento contém três elementos filhos: "`orderperson`", "`shipto`" e "`item`".

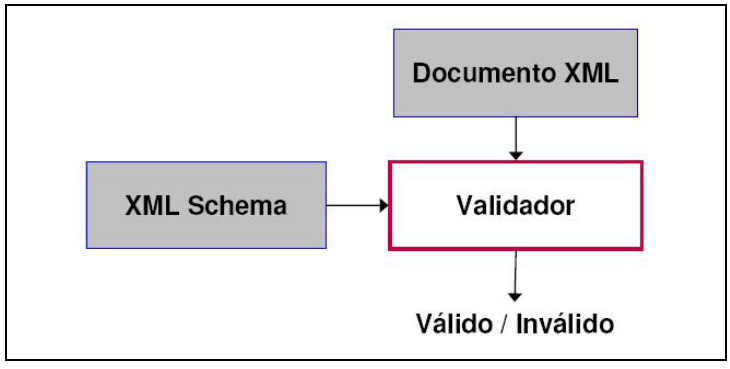
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<shiporder ordered="889923">
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="shiporder.xsd">
  <orderperson>John Smith</orderperson>
  <shipto>
    <name>Ola Nordmann</name>
    <address>Lanngt 23</address>
    <city>4000 Stavanger</city>
    <country>Norway</country>
  </shipto>
  <item>
    <title>Empire Burlesque</title>
    <note>Special Edition</note>
    <quantity>1</quantity>
    <price>10.90</price>
  </item>
  <item>
    <title>Hide your heart</title>
    <quantity>1</quantity>
    <price>9.90</price>
  </item>
</shiporder>
```

Quadro 2 – Exemplo de árvore do XML

2.4.1 XML *Schema*

XML *Schema* especifica a estrutura de um conjunto de documentos XML, juntamente com os tipos de dados de cada elemento ou atributo. A linguagem para especificação de XML *Schemas* adota conceitos de orientação a objetos e permite a criação de novos tipos de dados (CASANOVA, 2005, p. 4).

A figura 4 mostra a utilização do XML *Schema*que é utilizado pelo documento XML, através do validador, que retornará se o documento é válido ou não.



Fonte: Casanova (2005).

Figura 4 – Utilização do XML *Schema*

Esquema é o nome que se dá aos documentos que são criados com o objetivo de

definir e especificar o conteúdo de documentos XML.

Maia (2005) afirma que XML *Schema* é uma alternativa ao DTD baseada em XML. Um esquema XML descreve a estrutura de um documento XML, também chamada de XML *Schema Definition* (XSD).

Segundo Tesch Júnior (2002, p. 44), um XML *Schema* define como determinado conjunto de documentos XML devem ser construídos, quais os elementos eles podem ou devem conter e em que ordem devem aparecer no documento. Os documentos XML *Schema* são construídos através de uma sintaxe XML, podendo ser analisados por *parsers* XML.

O XML *Schema*, segundo Maia (2005), especifica os blocos de construção permitidos em um documento XML, definindo:

a) elementos que podem aparecer em um documento;
b) atributos que podem aparecer em um documento;
c) que elementos são elementos filhos;
d) a ordem dos elementos filhos;
e) o número de elementos filhos;
f) se um elemento é vazio ou pode incluir texto;
g) tipos de dados para elementos e atributos;
h) valores padrão e fixos para elementos e atributos.

XML *Schemas* utilizam sintaxe XML, proporcionando facilidades ao desenvolvedor, podendo utilizar para criação dos esquemas um editor XML e um *parser* para verificar arquivos XML *Schema*

Segundo Tesch Júnior (2002, p. 53), "tipos simples são usados para definição dos tipos de dados dos elementos e dos atributos". O Quadro 3 mostra um exemplo do XML *Schema* de tipos simples utilizado na validação do XML. O elemento "`restriction`" indica que o valor deve ser uma *string* e o elemento "`pattern value`" exige que o valor do atributo ou elemento tenha exatamente seis caracteres e cada caractere dever ser um número de 0 a 9.

```
<xs:simpleType name="stringtype">
  <xs:restriction base="xs:string"/>
</xs:simpleType>

<xs:simpleType name="dectype">
  <xs:restriction base="xs:decimal"/>
</xs:simpleType>

<xs:simpleType name="inttype">
  <xs:restriction base="xs:positiveInteger"/>
</xs:simpleType>

<xs:simpleType name="orderidtype">
  <xs:restriction base="xs:string"/>
    <xs:pattern value="[0-9]<6>"/>
  </xs:restriction>
</xs:simpleType>
```

Quadro 3 – Exemplo do XML *Schema* de tipos simples

Maia (2005) afirma que um elemento complexo contém outros elementos e/ou atributos. Há quatro tipos de elementos complexos:

a) elementos vazios;

b) elementos que contém apenas outros elementos;

c) elementos que contém somente texto;

d) elementos que contém elementos e texto.

O Quadro 4 mostra um exemplo de um XML *Schema* de tipos complexos.

```
<xs:complexType name"shiptotype">
   <xs:sequence>
      <xs:element name="name" type="stringtype">
      <xs:element name="address" type="stringtype">
      <xs:element name="city" type="stringtype">
      <xs:element name="country" type="stringtype">
   </xs:sequence>
<xs:complexType>

<xs:complexType name"itemtype">
   <xs:sequence>
      <xs:element name="title" type="stringtype">
      <xs:element name="note" type="stringtype">
      <xs:element name="quatity" type="inttype">
      <xs:element name="price" type="dectype">
   </xs:sequence>
<xs:complexType>
```

Quadro 4 – XML *Schema* de tipos complexos

Segundo Tittel (2003, p. 207), esquemas levam vantagens sobre DTDs principalmente por suportar *namespaces* e tipos de dados, características importantes para interfaces XML com banco de dados. Além disso, XML *Schemas* são extensíveis para adições futuras, permitem a criação de vários tipos de dados a partir de tipos de dados primitivos e possibilitam referenciar múltiplos esquemas em um mesmo documento.

Para Tesch Júnior (2002, p. 47), "o XML *Schema* utiliza *namespaces* para fazer a limitação do nome de elementos e atributos. O elemento *schema* é o elemento raiz de qualquer documento XML *Schema*, e os *namespaces* que limitam o esquema são declarados dentro dele".

2.4.2 Manipulação de XML em Java

Para realizar a leitura dos arquivos com as informações dos BDs pode ser utilizada a API JDOM que é a implementação da API *Document Object Model* (DOM) disponibilizada pela linguagem Java. Esta API fornece interfaces para manipulação de arquivos XML, sendo utilizada para ler o valor de atributos e elementos contidos nestes arquivos. Possui também, recursos de validação do arquivo XML.

DOM é um conjunto de interfaces que decompõem um documento XML em uma árvore transversal hierárquica de nós/objetos, agilizando a pesquisa por elementos, bastando apenas indicar o caminho de nós na árvore (VELOSO, 2003, p. 11).

O Quadro 5 indica o código fonte de leitura do arquivo XML, que tem como parâmetro o caminho onde se encontra este arquivo.

```
try{
   File  file       = new File(caminho);
   SAXBuilder  sax  = new SAXBuilder();
   return sax.build(file);
   }catch(Exception e){
      e.printStackTrace();
   return null;
   }
```

Quadro 5 – Exemplo de código fonte da leitura do XML

O *parser* `SAXBuilder` é utilizado para processar a estrutura do documento XML para dentro de uma variável do tipo `Document`. O método utilizado para processar o XML é o `build`, que recebe como parâmetro o caminho do arquivo.

Para Veloso (2003, p. 10), "SAX é um conjunto de interfaces que decompõem um documento XML em uma sucessão linear de chamadas de métodos".

O Quadro 6 mostra o processo de gravação a partir de um XML. O `XMLSerializer` converte um objeto para XML e o `FileWriter` serve para escrever em um arquivo.

```
Public void gravarXML(Document doc) throws Exception{
  StringWriter fos = new StringWriter();
  OutputFormat of = new OutputFormat("XML","ISO-8859-1",true);
  XMLSerializer serializer = new XMLSerializer(fos,of);
  serializer.asDOMSerializer();
  serializer.serialize( doc.getDocumentElement() );
  FileWriter out = new
   FileWriter(getFuncoes().getCaminhoXML()+getFuncoes().getNome()+".xml");
  out.write(fos.toString());
  out.close();
}
```

Quadro 6 – Exemplo de código fonte da gravação do XML

O Quadro 7 mostra um exemplo de código para obtenção de valores de nós do XML. Estes são armazenados para serem lidos ou manipulados.

```
private void setMapeamentoGlobal(Element eleBanco){

   //buscar <tabelas>
   List    listaTab      =  eleBanco.getChildren();
   for(int i=0; i < listaTab.size(); i++){
      Element eleTabs     = (Element)listaTab.get(i);
      List    listaTabs   = eleTabs.getChildren();

      //buscar <tabela>
      Element eleTab      =   (Element)listaTabs.get(0);
      String  nmTabela    =   eleTab.getText();

      //buscar <colunas>
      Element eleColunas      =   (Element)listaTabs.get(1);
      List    listaColunas    =   eleColunas.getChildren();
      ArrayList       colunas = new ArrayList();
      for(int j=0; j < listaColunas.size(); j++){
         Element eleCol  = (Element) listaColunas.get(j);
         colunas.add(eleCol.getText());
      }
      cadBanco.addTabelasGlobal(nmTabela, colunas);
   }
}
```

Quadro 7 – Exemplo de código fonte da obtenção de nós do XML

2.5 TRABALHOS CORRELATOS

Gianisini Júnior (2006) produziu um *framework* capaz de interceptar e replicar as transações de um BD local para um BD remoto, garantindo qualidade de serviço mesmo que não haja sincronismo de comunicação, fornecendo um nível considerável de abstração das bases de dados envolvidas na replicação.

3 DESENVOLVIMENTO

Neste trabalho foram desenvolvidos dois softwares:

a) um *driver* JDBC que tem por finalidade realizar consultas em BDDs;
b) uma ferramenta de edição de ECG para realizar o mapeamento das tabelas locais de cada BD com as tabelas globais.

Para demonstrar o funcionamento do *driver*, foi desenvolvida uma aplicação, onde é possível realizar consultas, deixando transparente o modo como ocorre a mesma.

Para a especificação foi utilizada a técnica de *Unified Modeling Language* (UML). Maiores detalhes sobre a UML podem ser obtidos em PAGE-JONES (2001).

3.1 DRIVER JDBC

Neste capítulo são apresentados os requisitos, diagramas da UML, a implementação e exemplos de uso do *driver* JDBC.

3.1.1 Requisitos

O *driver* desenvolvido deverá:

a) realizar conexão entre os BDs envolvidos, possibilitando acesso as informações locais e remotas, baseando-se na interpretação de um documento XML. Este documento conterá os mapeamentos entre os BDs (Requisito Funcional – RNF);
b) permitir a busca de dados no destino informado, aplicando o conceito de fragmentação horizontal de dados, utilizando o documento XML mencionado acima (RF);
c) manter a independência física dos dados, sendo que o armazenamento dos dados no banco deve ser transparente ao usuário (RF);
d) ser implementado através do ambiente Java utilizando a ferramenta NetBeans 5.5 (Requisito Não-Funcional – RNF).

3.1.2 Especificação

Nesta seção são apresentados os diagramas de classes e de seqüência do *driver* JDBC desenvolvido.

3.1.2.1 Diagrama de classes

A figura 5 apresenta o diagrama de classes do *driver* JDBC.

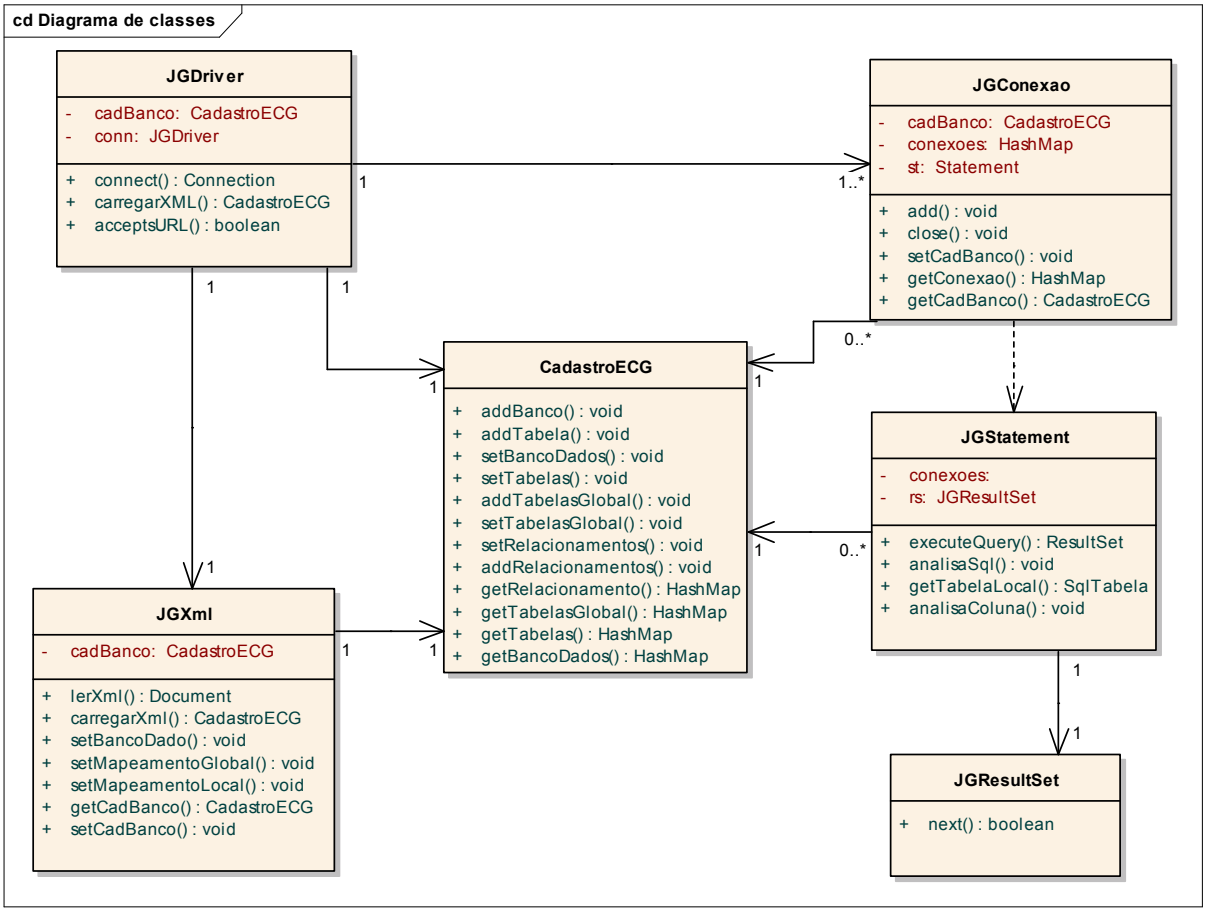


Figura 5 – Diagrama de classes

A classe `JGDriver` é responsável por registrar o *driver*. Nesta classe os BDs são lidos do XML e as conexões são efetuadas. A criação do *`Statement`* é realizada na classe `JGConexao`, onde são armazenadas as conexões. A classe `JGXml` carrega o XML e armazena-o num objeto `CadastroECG`. Neste objeto estão contidos os BDs e os mapeamentos local e global. A montagem do SQL de cada BD é realizada na classe `JGStatement`. Nesta classe também são realizadas as análises léxica, sintática e semântica

para validação do *select* informado como entrada, retornado as conexões e as consultas que cada BD deve executar. A classe `JGResultSet`, percorre os BDs e executa as consultas localmente, armazenando os resultados num *`resultSet`*.

3.1.2.2 Diagrama de seqüência

Abaixo, na figura 6, é apresentado o diagrama de seqüência de utilização geral do *driver* JDBC.

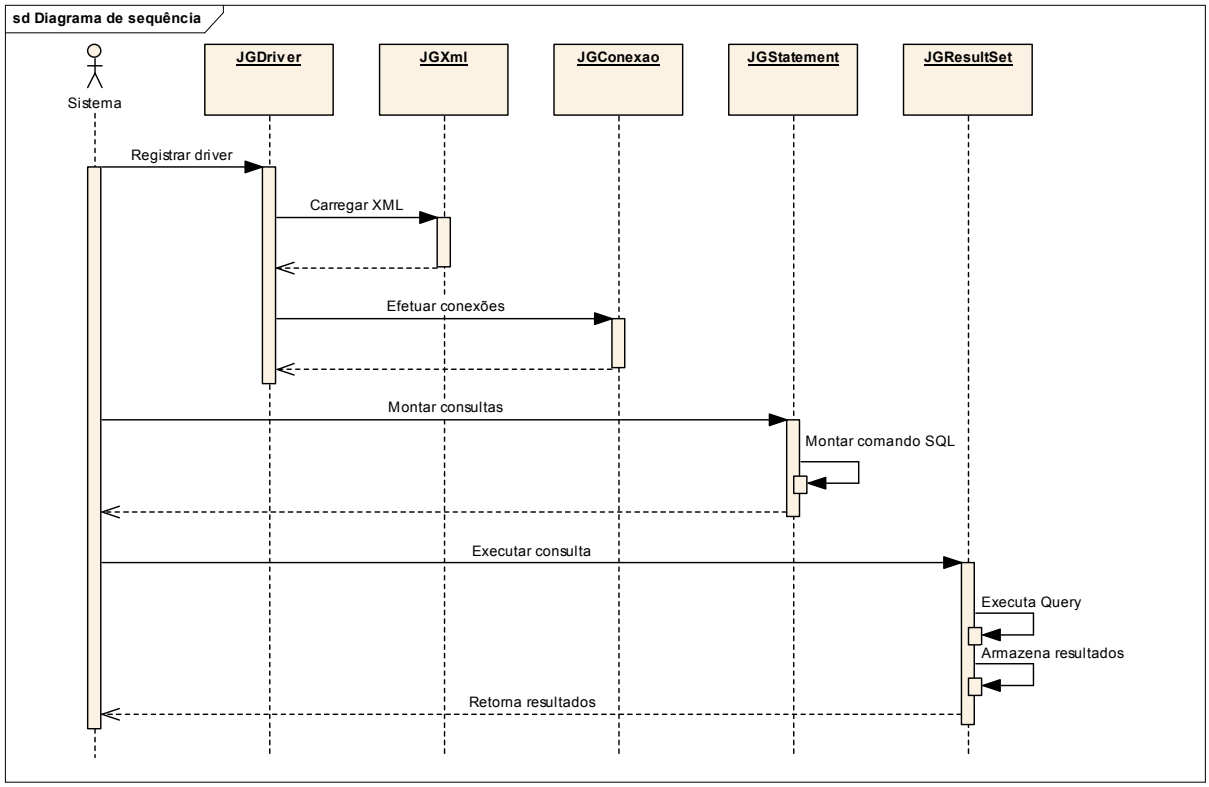


Figura 6 – Diagrama de seqüência de utilização geral do *driver* JDBC

O sistema solicita o registro do *driver*, chamado de "`br.furb.jdbc.JGDriver`" . O *driver* é registrado e a partir da leitura do XML são efetuadas as conexões. Em seguida, as consultas de cada BD são montadas e associadas a cada conexão. Cada consulta é executada no seu respectivo BD e os resultados são retornados para o sistema.

3.1.3 Esquema do XML

Para validação do ECG foram desenvolvidos dois XML *Schemas* : um de tipos simples

e outro de tipos complexos. Estes esquemas definem os atributos e elementos que o ECG pode conter e a seqüência como os dados devem aparecer. O tratamento quanto aos tipos de dados de cada coluna das tabelas é de responsabilidade do desenvolvedor, sendo que o *driver* JDBC desenvolvido retorna o tipo da coluna de acordo com a tabela local. O Quadro 8 apresenta o XML *Schema* de tipos simples.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema" targetNamespace=""
   elementFormDefault="qualified">
   <simpleType name="st_nomeBanco">
      <restriction base="string"/>
   </simpleType>
   <simpleType name="st_driver">
      <restriction base="string"/>
   </simpleType>
   <simpleType name="st_url">
      <restriction base="string"/>
   </simpleType>
   <simpleType name="st_usuario">
      <restriction base="string"/>
   </simpleType>
   <simpleType name="st_senha">
      <restriction base="string"/>
   </simpleType>
   <simpleType name="st_nomeTabela">
      <restriction base="string"/>
   </simpleType>
   <simpleType name="st_nomeColuna">
      <restriction base="string"/>
   </simpleType>
</schema>
```

Quadro 8 – XML *Schema* de tipos simples

Quando um elemento ou atributo tem um tipo definido, pode-se criar uma restrição (`restriction base`) ao conteúdo dele. Por exemplo, se o tipo de um elemento é "*`date`*" e contém uma *string* "Computação", o elemento não vai ser validado.

O Quadro 9 mostra um trecho do XML *Schema* de tipos complexos para a validação do ECG.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema" targetNamespace=""
   elementFormDefault="qualified">
   <include schemaLocation="JGSimpleTypes.xsd"/>
   <complexType name="ct_ecg">
      <sequence>
         <element name="bancosDados" type="ct_bancosDados"/>
         <element name="mapeamentoGlobal" type="ct_mapeamentoGlobal"/>
         <element name="mapeamentoLocal" type="ct_mapeamentoLocal"/>
      </sequence>
   </complexType>
   <complexType name="ct_bancosDados">
      <sequence>
         <element name="bancoDado" maxOccurs="unbounded">
            <complexType>
               <sequence>
                  <element name="nome" type="st_nomeBanco"/>
                  <element name="driver" type="st_driver"/>
                  <element name="url" type="st_url"/>
                  <element name="usuario" type="st_usuario"/>
                  <element name="senha" type="st_senha"/>
               </sequence>
            </complexType>
         </element>
      </sequence>
   </complexType>
   <complexType name="ct_mapeamentoGlobal">
      <sequence>
         <element name="tabela" maxOccurs="unbounded">
            <complexType>
               <sequence>
                  <element name="nomeTabela" type="st_nomeTabela"/>
                  <element name="colunas" type="ct_colunaGlobal"
                                 maxOccurs="unbounded"/>
               </sequence>
            </complexType>
         </element>
      </sequence>
   </complexType>
   <complexType name="ct_colunaGlobal">
         <element name="coluna" type="st_nomeColuna"/>
   </complexType>
</schema>
```

Quadro 9 – XML *Schema* de tipos complexos

Um tipo complexo (`complexType`) é um elemento XML que contém outros elementos ou atributos. O indicador `sequence` identifica que os elementos filhos devem aparecer na mesma ordem da declaração. O indicador de ocorrência (`maxOccurs`) representa com que freqüência um elemento pode ocorrer. Neste caso, `unbounded` indica que o elemento pode aparecer inúmeras vezes. Além disso, na quarta linha do XML de tipos complexos, o atributo `schemaLocation` indica o XML *Schema* que também será utilizado na validação.

3.1.4 Implementação

Para o desenvolvimento do *driver* JDBC foi utilizada a linguagem Java JDK 1.6 através da plataforma Netbeans 5.5.

Os *drivers* específicos de cada BD foram adicionados ao projeto, os quais são disponibilizados pelos seus fabricantes. Estes *drivers* são responsáveis pelas conexões locais.

Foi desenvolvida uma *Backus-Naur Form* (BNF) para validação das consultas realizadas pelo usuário. A criação da BNF e geração das classes de validação foram desenvolvidas através do Gerador de Analisadores Léxicos e Sintáticos (GALS). O Quadro 10 apresenta um trecho do *select* da gramática desenvolvida.

```
<SQLTest>          ::=  <Comando>  <PontoVirgula>;
<PontoVirgula>     ::= ";" | î ;
<Comando>          ::= <SelectSQL>;
<SelectSQL>        ::=  <SelectStmt> <UnionSelect>;
<UnionSelect>      ::= UNION <AllOpcional> <SelectStmt> <UnionSelect>| î;
<AllOpcional>      ::= ALL | î ;
<SelectStmt>       ::= <SelectClause> <FromClause> <WhereOpcional>
                       <GroupByClauseOpcional> <HavingClauseOpcional>
                       <OrderByClauseOpcional>;
<SelectClause>     ::= SELECT #1 <DistinctAllOpcional> <SelectFieldList>;
<DistinctAllOpcional>   ::= DISTINCT #200 | ALL #201 | î ;
<Null>                  ::= NULL #202;
...
<L>                ::= <InSetExpr> | <BetweenExpr> | <LikeTest>;
<LikeTest>         ::= LIKE #50 <SqlParam>;
<BetweenExpr>      ::= BETWEEN #51 <Field> AND #52 <Field>;
<InSetExpr>        ::= IN #53 <OpenParens> <FieldSub> <CloseParens>;
<FieldSub>         ::= <FieldList> | <SubSelectSQL>;
<SqlParam>         ::= SQLString #309;
<Operator>         ::= <MathOperator> | <WordOperator>;
<MathOperator>     ::= "*" #98 | "/"| "+"| "-";
<WordOperator>     ::= AND #52 | OR #54;
...
<FromClause>       ::= FROM #9 <FromTableList>;
<FromTableList>    ::= <QualifiedTable> <QualifiedSeparator>;
<QualifiedTable>   ::= ident #4 <AsAliasOpcional>;
...
<WhereOpcional>    ::= <WhereClause> | î;
<WhereClause>      ::= WHERE #10 <SearchCondition>;
<SearchCondition>  ::= <Expression>;
<GroupByClauseOpcional> ::= <GroupByClause> | î;
<GroupByClause>    ::= GROUP BY #11 <FieldList>;
<HavingClauseOpcional>  ::= <HavingClause> | î;
<HavingClause>     ::= HAVING #12 <SearchCondition>;
<OrderByClauseOpcional> ::= <OrderByClause> | î;
<OrderByClause>    ::= ORDER BY #13 <OrderByFldList>;
...
```

Quadro 10 – Trecho da gramática da BNF

No Apêndice A é demonstrada a implementação da classe `Semantico` que faz a análise do comando SQL recebido como entrada. Foi definido um modelo de SQL padrão para a criação da BNF. As cláusulas que compõem o comando *select* são consideradas relativamente simples e foram criadas a partir das sintaxes suportadas pelos BDs utilizados na realização dos testes. Portanto, o *driver* desenvolvido pode ser utilizado para acesso a qualquer BD, independente do fabricante, contanto que suporte este modelo.

A classe `JGDriver` implementa a leitura do arquivo XML. Nela também são criadas as conexões com os BDs relacionados no ECG. O Quadro 11 apresenta a parte do código fonte onde são realizadas estas tarefas.

```
private CadastroBanco   cadBanco = null;
private JGConexao       conn;

public JGDriver() throws SQLException {
   conn = new JGConexao();
}

public Connection connect(String url, Properties info) throws
    SQLException {

    if (acceptsURL(url) == false)
       return null;
    // abrir e carregar o arquivo xml
    if (cadBanco == null){
       cadBanco =  carregarXML(url.substring(9,url.length()));
       auxiliar =  cadBanco.getBancoDados();
    }
    conn.setCadBanco(cadBanco);
    // fazer a conexao com cada bd
    try{
       if (auxiliar != null ) {
          Iterator it = auxiliar.keySet().iterator();
          while(it.hasNext()){
          String chave = (String)it.next();
          if(!existeConexao(conn.getConexao(), chave)){
         BancoDados bancoDados = (BancoDados)auxiliar.get(chave);
         Class.forName(bancoDados.getDriver());
         conn.add(bancoDados.getNomeBanco(),
            DriverManager.getConnection(bancoDados.getUrl(),
            bancoDados.getUsuario(), bancoDados.getSenha()));
              }
          }
       }
    }catch(SQLException e){
    conn.close();
    throw e;
    } catch(Exception e){
    conn.close();
    e.printStackTrace();
    }
return conn;
}
```

Quadro 11 – Leitura do XML e conexão com BDs na classe `JGDriver`

As conexões efetuadas pelo *driver* ficam armazenadas em objetos da classe `JGConexao`. Nesta classe são criados os objetos `Statement`, utilizados para enviar comandos SQL para um banco de dados. A consulta é montada com uma `String` e invocada pelo método `executeQuery`, da classe `JGStatement`. O Quadro 12 apresenta um trecho do código onde são localizadas a colunas e tabelas de cada BD para criação do novo comando.

```
private SqlTabela getTabelaLocal(ArrayList tabelas, String
                                 nmTabela, SqlTabela sqlTabela){
   boolean achou = false;
   int i = 0;
   String tabelaLocal = "";
   while(i < tabelas.size() && !achou){
      Tabela tabCol = (Tabela)tabelas.get(i);
      NomeTabelas nmTabelas = (NomeTabelas)tabCol.getNomeTabelas();
      if(nmTabelas.getTabelaGlobal().equalsIgnoreCase(nmTabela)){
    achou = true;
    sqlTabela.setTabelaLocal(nmTabelas.getTabelaLocal());
    //Buscar as colunas
         ArrayList   colunas = (ArrayList)tabCol.getColunas();
         tabCol.setColunas(analisaColuna(colunas,
                           sqlTabela.getColunas()));
         tabelas.set(i, tabCol);
      }else
         i++;
   }
   return sqlTabela;
}

private String getColunaLocal(ArrayList colunas, String nmColuna){
   int i=0;
   boolean achou = false;
   String  nmColunaLocal = "";
   while(i < colunas.size() && !achou){
      Coluna col = (Coluna)colunas.get(i);
      if(col.getColunaGlobal().equalsIgnoreCase(nmColuna)){
         achou = true;
         nmColunaLocal   = col.getColunaLocal();
      }else
         i++;
   }
   return nmColunaLocal;
}
```

Quadro 12 – Localização de tabelas e colunas locais na classe `JGStatement`

A classe `JGResultSet` representa o conjunto de registros que retornam de uma consulta SQL. Nesta classe é representada a técnica de fragmentação horizontal, que retorna as linhas (tuplas) de uma relação. O Quadro 13 apresenta um trecho desta classe, onde se realiza a execução das consultas para cada BD e grava as informações num atributo.

```
public boolean next() throws SQLException {
     boolean resultado;
     if ( cursor > -1 )
          resultado = rs.next();
     else
          resultado = false;
     if ( resultado == false){
          if ( cursor > -1 )
               rs.close();
          cursor++;
          if ( cursor > bancos.size()-1)
               return false;
          while (!resultado && cursor <= bancos.size()-1){
               String sql = sqls.get(bancos.get(cursor)).toString();
               if (!sql.equalsIgnoreCase("")){
                    executaSql();
                    resultado = rs.next();
               }else
                    Resultado = false;
               if (!resultado)
                    cursor++;
          }
     }
     return resultado;
}

private void executaSql(){
     try{
          Connection con = conexoes.get(bancos.get(cursor));
          Statement  st  = con.createStatement();
          rs = st.executeQuery(sqls.get(bancos.get(cursor)).toString());
     }catch(Exception e){
          e.printStackTrace();
     }
}
```

Quadro 13 – Execução de consultas locais na classe `JGResultSet`

A variável `cursor` representa a quantidade de BDs envolvidos na consulta.A conexão é fechada após a execução da consulta. Depois que todos os BDs efetuaram suas respectivas consultas o sistema retorna *`false`* e é finalizada a execução.

3.2 FERRAMENTA DE EDIÇÃO DE ECG

Neste capítulo são apresentados os requisitos, diagramas da UML, a implementação e a forma de utilização da ferramenta de edição de XML.

3.2.1 Requisitos

A ferramenta de edição de ECG deverá:

a) realizar o mapeamento das tabelas e colunas de cada BD local a partir do mapeamento global (Requisito Funcional - RF);
b) permitir abrir, salvar e manipular estes arquivos (RF);
c) utilizar XML como mecanismo de persistência (Requisito Não-Funcional – RNF);
d) ser implementado através do ambiente Java utilizando a ferramenta NetBeans 5.5 (RNF).

3.2.2 Especificação

Nesta seção são apresentados os diagramas de casos de uso e o diagrama de atividades da ferramenta de edição de ECG.

3.2.2.1 Diagrama de casos de uso

O diagrama de casos de uso é utilizado pela UML para definir as interações que o usuário ou algum ator específico da aplicação tem com o software. A figura 7 mostra o diagrama de casos de uso.

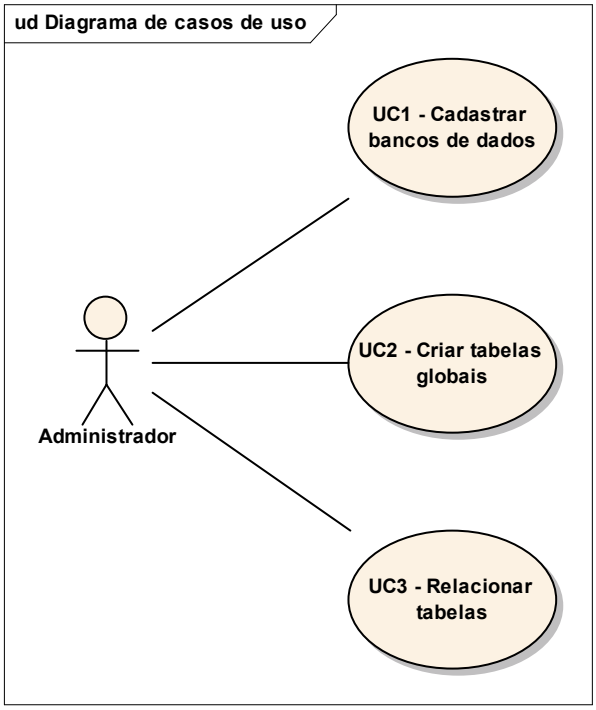


Figura 7 – Diagrama de casos de uso

O Quadro 14 apresenta a descrição do caso de uso cadastrar bancos de dados.

UC1 – Cadastrar bancos de dados

Sumário: O usuário cadastra um banco de dados.

Ator primário: Usuário.

Precondições: Nenhuma

Fluxo principal:
1. O usuário informa o caminho e nome do arquivo a ser salvo.
2. O usuário informa os parâmetros para conexão com o BD e salva.
3. O sistema grava no XML o BD cadastrado.
4. O sistema carrega os BDs cadastrados e mostra ao usuário, finalizando o caso de uso.

Fluxo alternativo (1): abrir um arquivo XML existente
a) O usuário carrega um arquivo XML existente.
b) O caso de uso prossegue a partir do passo 2.

Fluxo alternativo (3): exclusão de um banco de dados
a) O usuário escolhe um banco de dados cadastrado e exclui.
b) O sistema solicita confirmação para exclusão.
c) Se o usuário confirmar o sistema excluí o cadastro do banco de dados e o caso de uso prossegue a partir do passo 4.
d) Se o usuário não confirmar o caso de uso prossegue a partir do passo 5.

Pós-condições: BD cadastrado.

Quadro 14 – Descrição do caso de uso UC1 – Cadastrar banco de dados

O Quadro 15 apresenta a descrição do caso de uso criar tabelas globais.

UC2 – Criar tabelas globais

Sumário: O usuário cadastra as tabelas globais.

Ator primário: Usuário.

Precondições: Possuir pelo menos um BD cadastrado.

Fluxo principal:
1. O usuário informa o nome da tabela global e das colunas globais.
2. O sistema grava no XML o mapeamento global cadastrado.

Fluxo alternativo (2): utilização de tabela local como base
a) O usuário seleciona um banco de dados e a tabela desejada.
b) O usuário confirma a utilização da tabela como base.
c) O sistema grava a tabela global e o mapeamento local a partir desta tabela.
d) O caso de uso retorna ao passo 2.

Fluxo alternativo (3): exclusão de uma tabela global
a) O usuário escolhe uma tabela global cadastrada e exclui.
b) Se a tabela global foi criada a partir de uma tabela local, o sistema exclui o mapeamento global e local da tabela e o caso de uso prossegue a partir do passo 4.
c) Se a tabela global não foi criada a partir de uma tabela local, o sistema exclui o mapeamento global da tabela e o caso de uso prossegue a partir do passo 4.

Pós-condições: Tabelas globais cadastradas.

Quadro 15 – Descrição do caso de uso UC2 – Criar tabelas globais

O Quadro 16 apresenta a descrição do caso de uso relacionar tabelas.

UC3 – Relacionar tabelas

Sumário: O usuário cria o relacionamento das tabelas globais com as tabelas de cada BD.

Ator primário: Usuário.

Precondições: BDs e tabelas globais cadastradas.

Fluxo principal:
1. O usuário carrega uma tabela local de um BD específico.
2. O usuário seleciona a tabela global para realizar o relacionamento.
3. O usuário seleciona a coluna global e a coluna local e confirma.
4. O sistema grava no XML os dados.
5. O sistema carrega os relacionamentos cadastrados e mostra ao usuário, finalizando o caso de uso.

Fluxo alternativo (3): exclusão de um relacionamento
a) O usuário seleciona um relacionamento e exclui.
b) O caso de uso prossegue a partir do passo 4.

Pós-condições: Relacionamento cadastrado.

Quadro 16 – Descrição do caso de uso UC3 – Relacionar tabelas

3.2.2.2 Diagrama de atividades

A figura 8 representa o diagrama de atividades da ferramenta de edição de ECG.

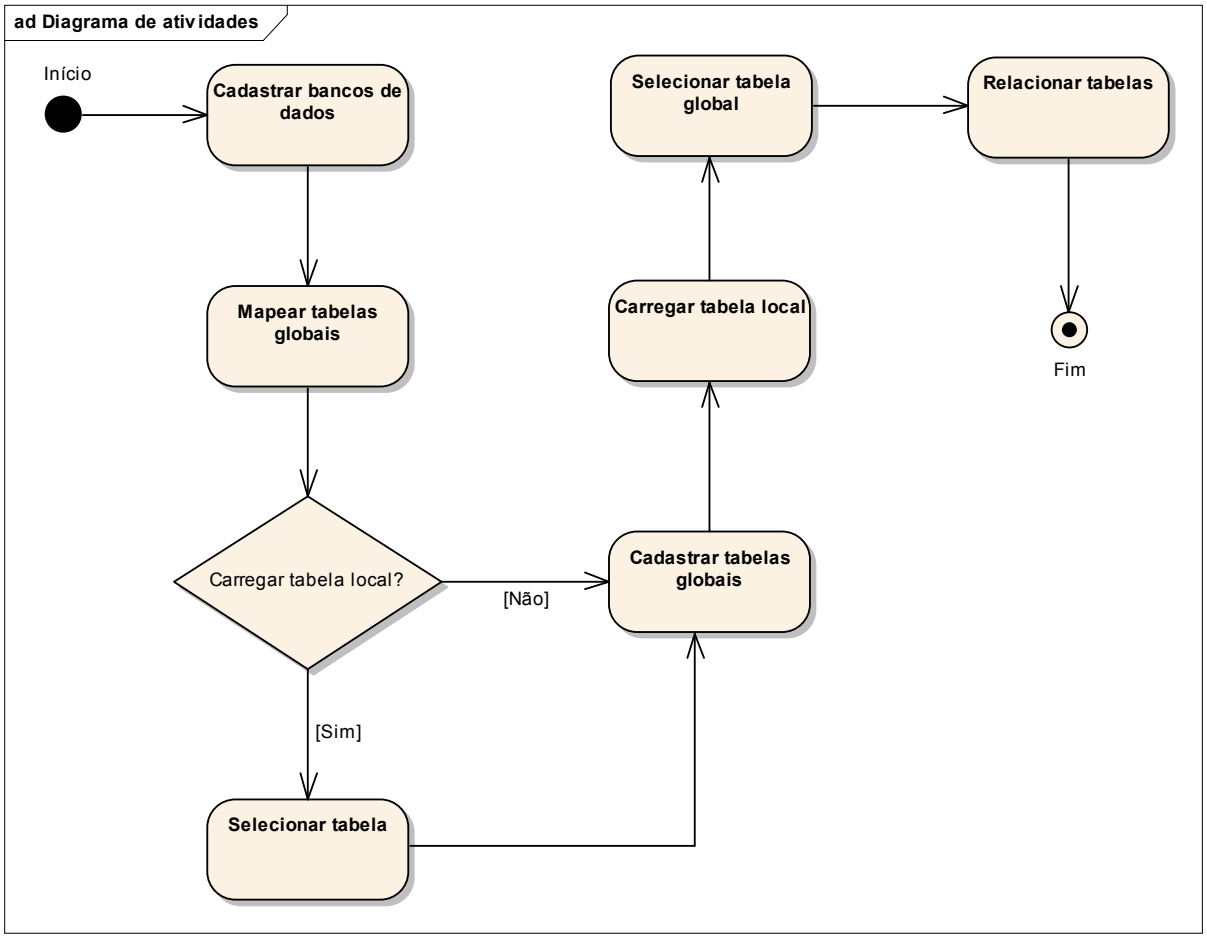


Figura 8 – Diagrama de atividades

Inicialmente, o usuário realiza o cadastro dos BDs. Neste passo, o mesmo pode cadastrar ou abrir um arquivo XML já existente para manipulação dos dados. Nesta tela, o administrador deve informar o usuário e senha do BD, bem como o *driver* e a *url* de conexão de cada BD.

Após realizar o cadastro dos BDs, o usuário deverá realizar o cadastro das tabelas globais. Neste momento, o usuário pode optar por cadastrar uma tabela global com a nomenclatura definida pelo mesmo ou selecionar uma tabela local, de um BD previamente cadastrado e utilizá-la como base. Caso opte por selecionar uma tabela local, o sistema apresenta uma tela com os BDs cadastrados e suas respectivas tabelas para que o administrador possa selecioná-las.

O próximo passo é o relacionamento entre as tabelas globais e locais. O usuário seleciona uma tabela local de um BD cadastrado e relaciona com uma tabela global, indicando

que as duas têm o mesmo significado. Tanto neste passo quanto nos anteriores é possível manipular os registros, excluindo, inserindo e alterando as informações.

3.2.3 Implementação

Para o desenvolvimento da ferramenta de edição de ECGs foi utilizada a linguagem Java JDK 1.6 através da plataforma Netbeans 5.5. Foram utilizados componentes gráficos da plataforma para criação da interface com o usuário. O Quadro 17 mostra um trecho do código fonte utilizado na criação do XML, no que se refere ao cadastro dos BDs.

```
/* cria Element(s) para gravar no xml*/
public Element gravarXmlBanco(Document doc){
  Element banco = doc.createElement("jg:bancosDados");

  Iterator it = getBancoDados().keySet().iterator();
    while(it.hasNext()){
      BancoDados obj = (BancoDados)getBancoDados().get(it.next());
      Element bd = doc.createElement("jg:bancoDado");

      Element nomeBanco = doc.createElement("jg:nome");
      nomeBanco.appendChild(doc.createTextNode(obj.getNomeBanco()));
      bd.appendChild(nomeBanco);

      Element driver = doc.createElement("jg:driver");
      driver.appendChild(doc.createTextNode(obj.getDriver()));
      bd.appendChild(driver);

      Element url = doc.createElement("jg:url");
      url.appendChild(doc.createTextNode(obj.getUrl()));
      bd.appendChild(url);

      Element usuario = doc.createElement("jg:usuario");
      usuario.appendChild(doc.createTextNode(obj.getUsuario()));
      bd.appendChild(usuario);

      Element senha = doc.createElement("jg:senha");
      senha.appendChild(doc.createTextNode(obj.getSenha()));
      bd.appendChild(senha);

      banco.appendChild(bd);
   }
   return banco;
}
```

Quadro 17 – Cadastro dos BDs em XML

3.2.4 Operacionalidade da implementação

A seguir são apresentadas as telas da ferramenta para construção de ECGs com uma breve explicação de sua funcionalidade.

Na tela inicial da ferramenta são cadastrados os BDs que serão utilizados para realização das consultas. Informa-se o nome do arquivo a ser salvo, juntamente com o nome do *driver* para conexão com o banco, a *url* que é composta pelo endereço IP da máquina onde o BD está alojado, a porta e o próprio BD. O nome do usuário e a senha do banco também são necessários para realizar a conexão. Para a efetuar os cadastros, o usuário desta ferramenta deve ter conhecimentos suficientes de BDs, principalmente sobre modelagem entidade-relacionamento. A figura 9 mostra a tela de cadastro destas informações.

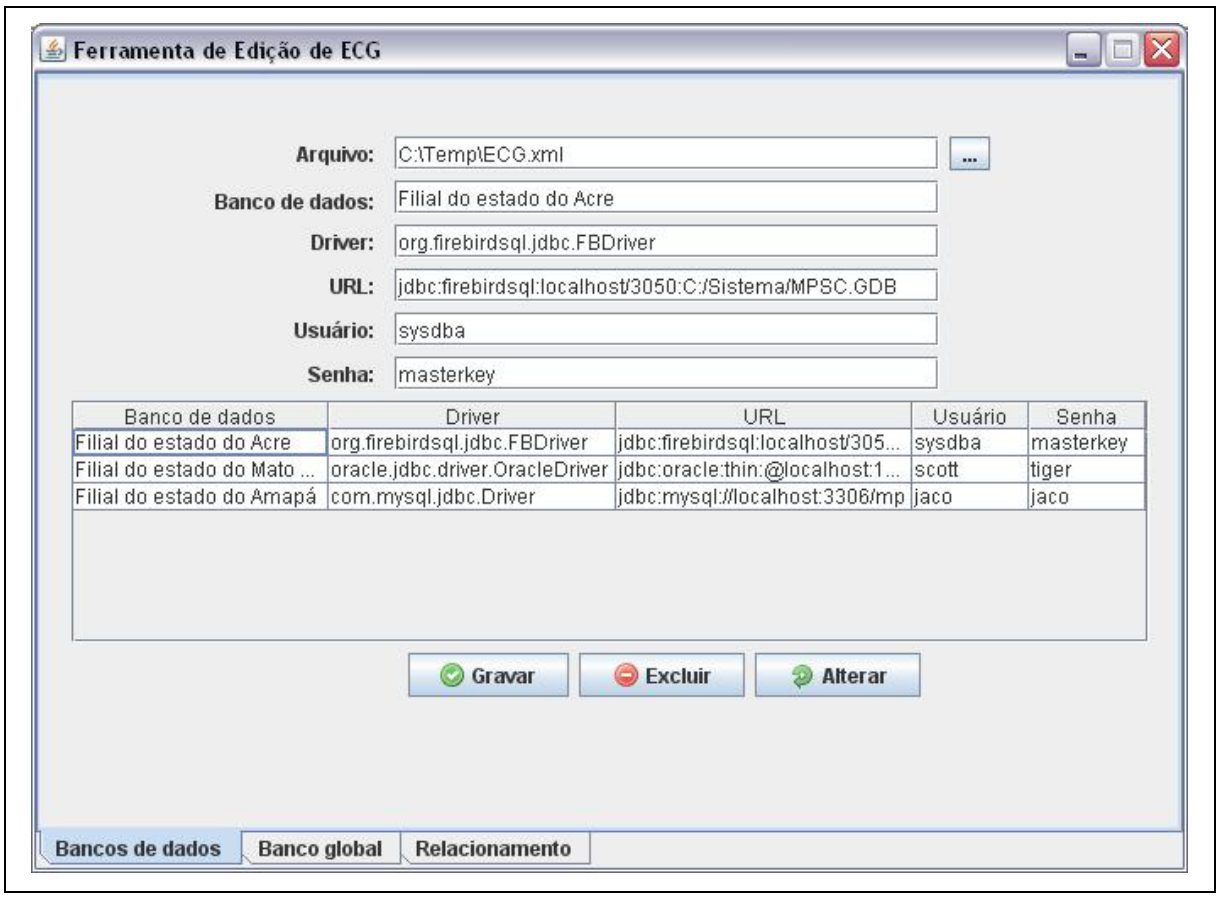


Figura 9 – Tela do cadastro de BDs

Realizado o cadastro dos BDs juntamente com os parâmetros necessários para a conexão, o próximo passo é cadastrar o mapeamento global. É importante adotar um padrão para o cadastro destas informações, pois estas tabelas e colunas serão utilizadas para realização das consultas globais. Como mostra a figura 10, a nomenclatura da tabela é clara e expressa o real significado da tabela.

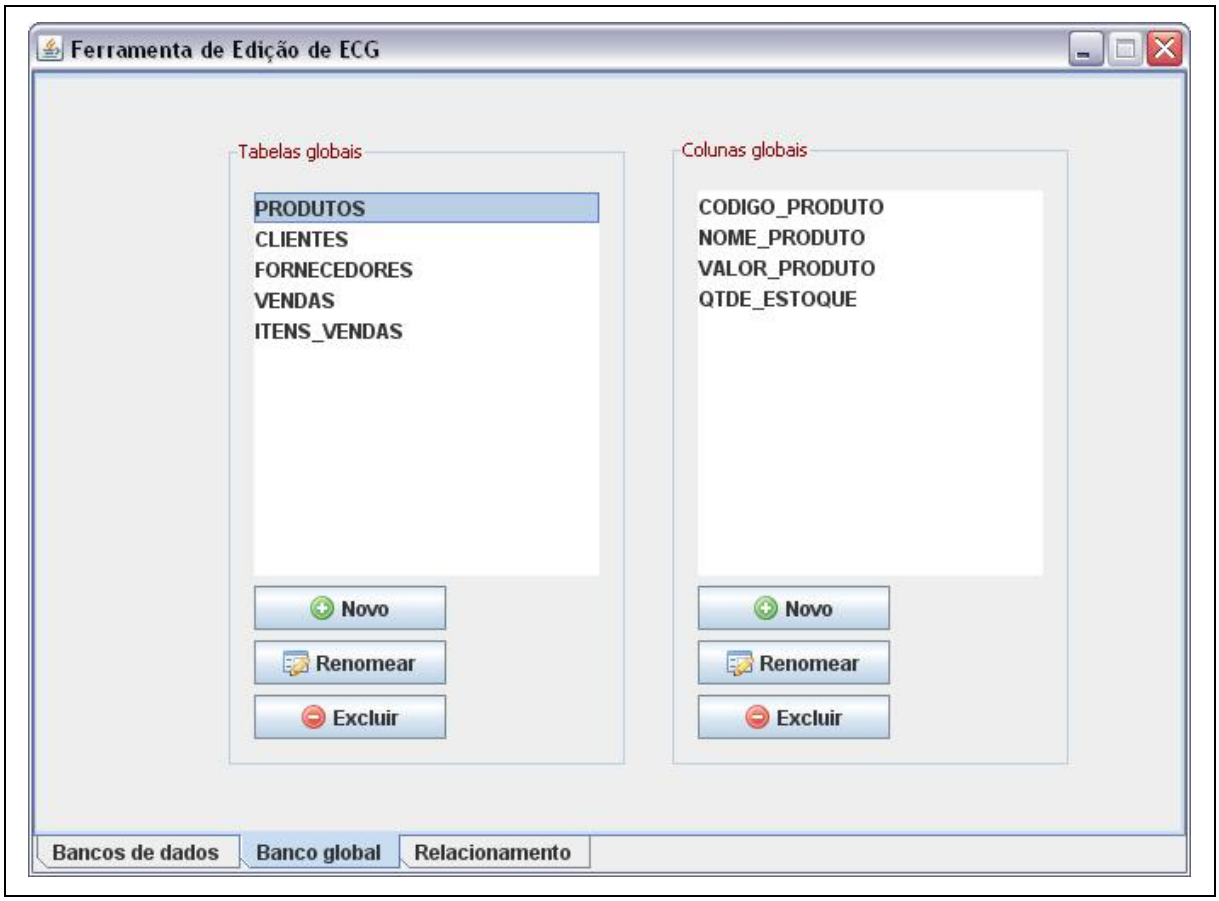


Figura 10 – Tela de visualização e manipulação das tabelas globais

Na parte esquerda da tela são visualizadas as tabelas globais. Tem-se a possibilidade de incluir, renomear e excluir uma tabela. No lado esquerdo da tela, estão as colunas relacionadas a estas tabelas, onde pode-se realizar as mesmas ações referente as tabelas. Ao incluir uma nova tabela, o sistema apresenta uma tela para realização deste cadastro. A figura 11 mostra a tela de cadastro de tabelas globais.

Figura 11 – Tela de cadastro de tabelas globais

Caso o usuário opte por selecionar uma tabela local de um BD previamente cadastrado, o mesmo deverá clicar em "`Importar tabela local`". Esta opção abre uma janela com os BDs e suas respectivas tabelas para serem selecionadas pelo usuário. Caso não opte por isto, o usuário poderá cadastrar tabelas e colunas locais utilizando a nomenclatura das mesmas a sua escolha. A figura 12 apresenta a tela de seleção de tabelas locais para ser utilizadas como base para tabelas globais.

Figura 12 – Tela de seleção de tabelas locais

Para que as consultas possam ser efetuadas localmente, é necessário fazer um

relacionamento entre as tabelas globais e as tabelas locais. O primeiro passo é selecionar uma tabela local de um BD qualquer. Depois deve-se selecionar uma tabela local respectiva e clicar em "`Relacionar colunas`". O botão "`Excluir colunas`" pode ser utilizado caso o usuário tenha se equivocado no cadastro de algum relacionamento. Tem-se também a opção de "`Excluir relacionamento`" que elimina o relacionamento da tabela por completo. A figura 13 mostra um exemplo de cadastro de relacionamentos entre uma tabela local "`CLIENTE`" e a tabela global "`CLIENTES`".

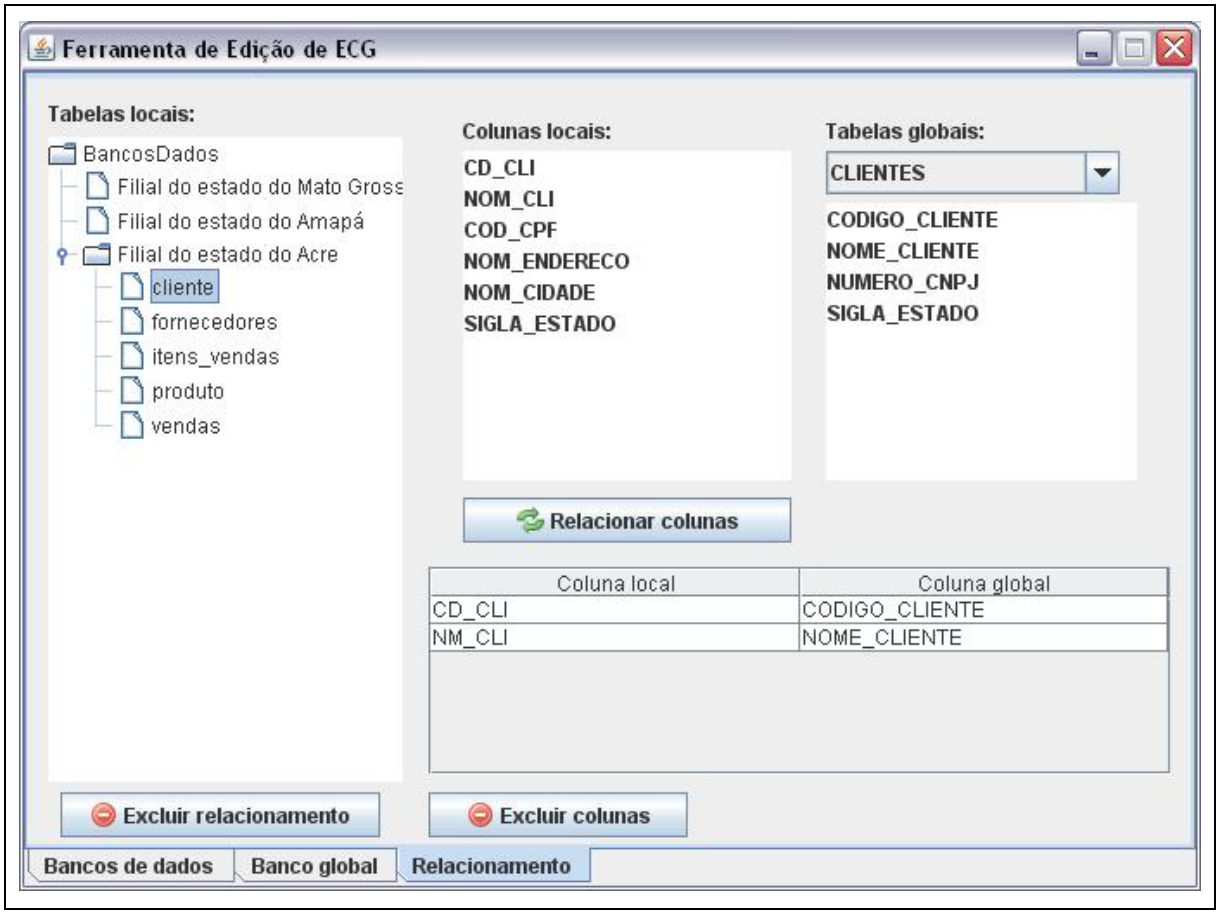


Figura 13 – Tela do cadastro dos relacionamentos

Na parte esquerda da tela, utiliza-se o processo de seleção de tabelas locais, através dos BDs cadastrados. No canto superior direito da tela, seleciona-se uma tabela global para efetuar o relacionamento, que é realizado através do botão "`Relacionar colunas`" partindo da seleção de colunas locais e globais. Na parte inferior da tela, estão dispostos os relacionamentos cadastrados entre as tabelas, podendo-se excluir estes relacionamentos caso necessário.

3.3 ESTUDO DE CASO

O *driver* JDBC desenvolvido tem por função realizar a conexão entre os BDs envolvidos na fragmentação de dados, retornando os dados de forma transparente para o usuário. Com isso, o *driver* pode ser utilizado por empresas que tenham filiais distantes umas das outras, podendo realizar consultas a dados atualizados e servirem como base para indicadores de gestão. A abstração dos dados é de nível considerável, pois o usuário dispara um comando SQL como entrada e o retorno disponibilizado pelo *driver* são as linhas de execução das consultas locais.

Para demonstrar o funcionamento do *driver* JDBC, foi desenvolvido um sistema de validação composto basicamente por uma tela de consultas, onde o usuário seleciona o arquivo XML cadastrado na ferramenta de edição de ECGs e realiza consultas baseado no mapeamento das tabelas cadastradas. Nesta tela, o usuário poderá selecionar o arquivo XML com o mapeamento, para que seja interpretado pelo *driver*. A base para as consultas são as tabelas globais, que foram mapeadas para cada BD cadastrado. Os BDs utilizados para a realização de testes foram o Oracle 9i, o MySQL 5.0 e o Firebird 1.5.

3.3.1 Requisitos

O sistema de validação deverá:

a) realizar consultas nas tabelas identificadas no mapeamento (RF);
b) permitir selecionar um ECG para realização destas consultas (RF);
c) utilizar a linguagem SQL como padrão para a realização de consultas (RF).
d) ser implementado na linguagem Java, para que a passagem de parâmetros para o *driver* ocorra de forma correta (RNF).

3.3.2 Modelo Entidade Relacionamento (MER)

Para a realização de testes, foram criadas tabelas em três BDs distintos, onde cada BD e cada tabela têm suas particularidades. Optou-se por criar tabelas e colunas bem diversificados para que fique expressa a relação entre as tabelas globais e as tabelas locais de

cada de cada BD. A criação dos modelos para realização dos testes não seguem o padrão e a normalização quanto à estruturação de tabelas e colunas. A figura 14 mostra o modelo físico de entidade-relacionamento de um banco em Oracle.

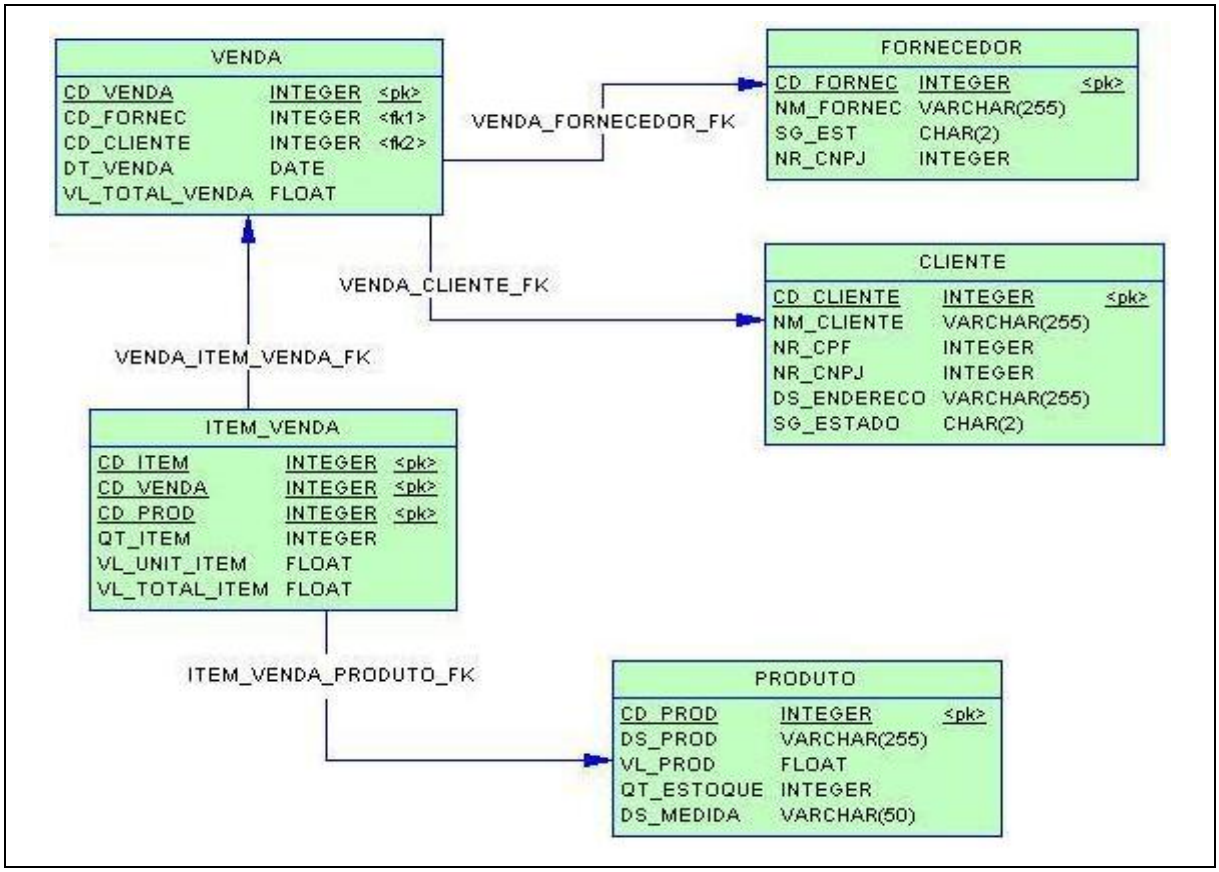


Figura 14 – MER do banco Oracle

Como mostra a figura 15, um banco em Firebird não possui a tabela de fornecedores, exemplificando a diversidade entre os BDs.

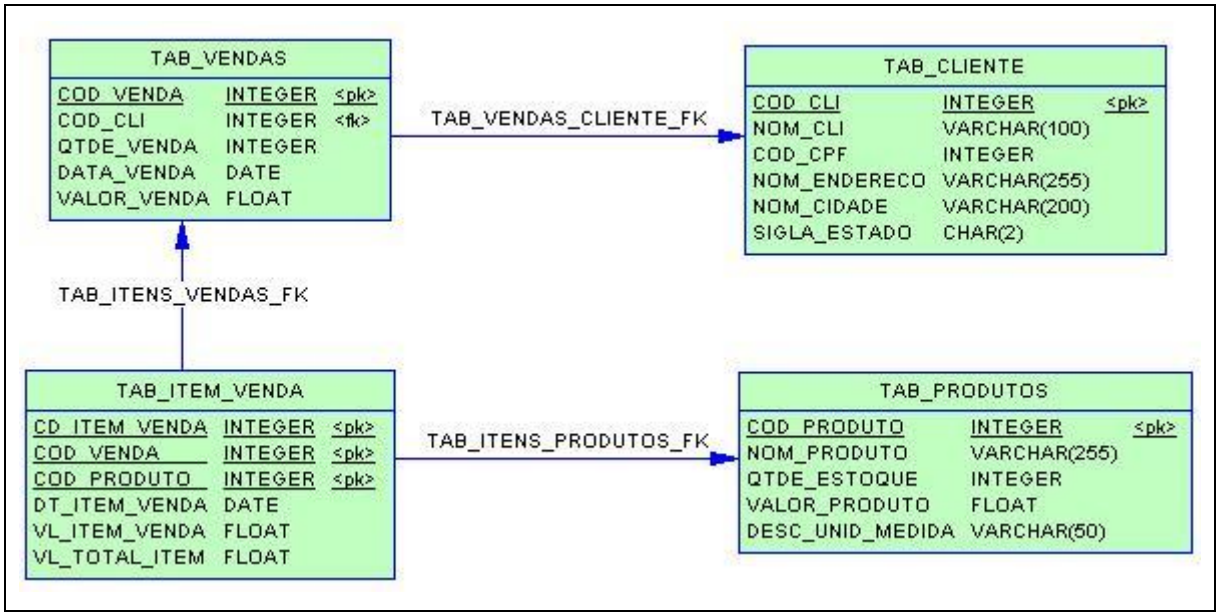


Figura 15 – MER do banco Firebird

A figura 16 mostra o modelo físico de um banco em MySQL, onde as tabelas e colunas divergem com relação a nomenclatura de tabelas e atributos em relação aos outros BDs.

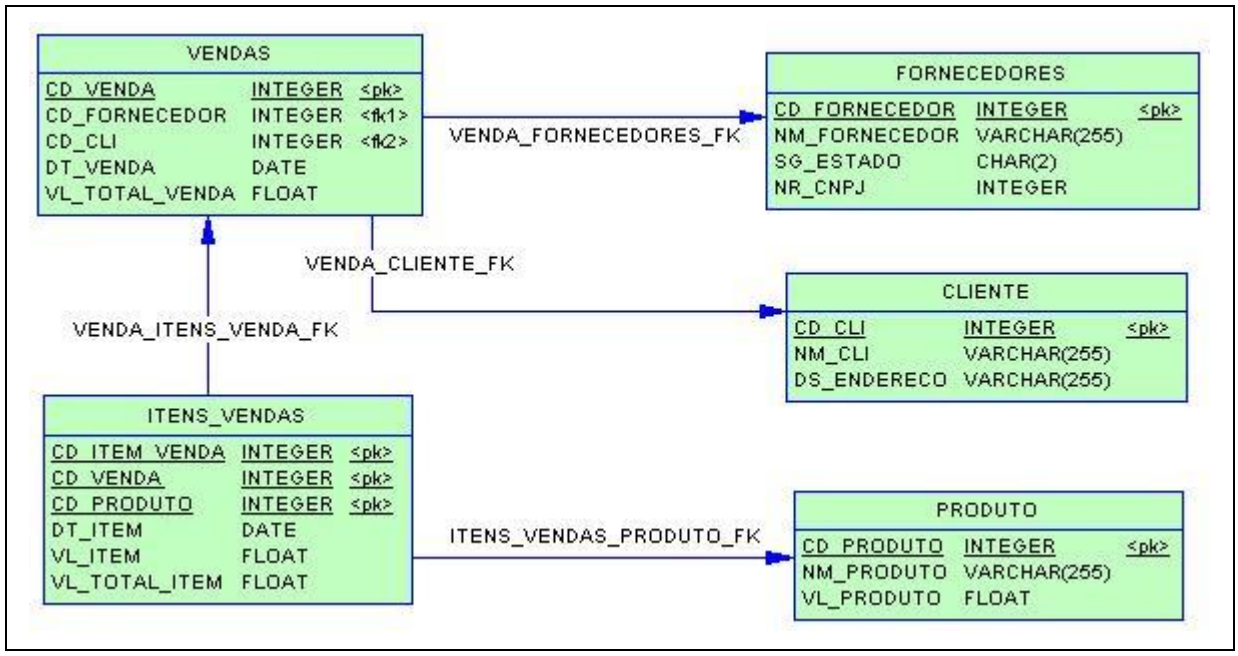


Figura 16 – MER do banco MySQL

3.3.3 ECG

O Quadro 18 mostra o XML gerado pela ferramenta de edição de ECGs contendo o mapeamento da tabela global `produtos` para a `Filial do estado do Amapá` que utiliza o banco MySQL. Este XML é o modelo interpretado pelo *driver* JDBC para conexão e montagem das consultas locais.

```
 <?xml version="1.0" encoding="ISO-8859-1" ?>
<<jg:ecg xmlns:jg="http://www.w3schools.com"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="C:/Temp/JGComplexTypes.xsd">
   <jg:bancosDados>
      <jg:bancoDado>
         <jg:nome>Filial do estado do Amapá</jg:nome>
         <jg:driver>com.mysql.jdbc.Driver</jg:driver>
         <jg:url>jdbc:mysql://localhost:3306/mp</jg:url>
         <jg:usuario>jaco</jg:usuario>
         <jg:senha>jaco</jg:senha>
      </jg:bancoDado>
   </jg:bancosDados>
   <jg:mapeamentoGlobal>
     <jg:tabela>
       <jg:nomeTabela>PRODUTOS</jg:nomeTabela>
       <jg:colunas>
          <jg:coluna>NOME_PRODUTO</jg:coluna>
       </jg:colunas>
      </jg:tabela>
   </jg:mapeamentoGlobal>
   <jg:mapeamentoLocal>
     <jg:relacionamento>
        <jg:banco>Filial do estado do Amapá</jg:banco>
        <jg:tabelas>
           <jg:tabela>
             <jg:nomeTabelas>
                 <jg:tabelaGlobal>PRODUTOS</jg:tabelaGlobal>
                 <jg:tabelaLocal>PRODUTO</jg:tabelaLocal>
             </jg:nomeTabelas>
             <jg:colunas>
               <jg:coluna>
                 <jg:colunaGlobal>NOME_PRODUTO</jg:colunaGlobal>
                 <jg:colunaLocal>NM_PRODUTO</jg:colunaLocal>
               </jg:coluna>
               <jg:coluna>
                  <jg:colunaGlobal>VALOR_PRODUTO</jg:colunaGlobal>
                  <jg:colunaLocal>VL_PRODUTO</jg:colunaLocal>
               </jg:coluna>
             </jg:colunas>
           </jg:tabela>
        </jg:tabelas>
     </jg:relacionamento>
   </jg:mapeamentoLocal>
 </jg:ecg>
```

Quadro 18 – ECG

3.3.4 Implementação

O sistema de validação foi desenvolvido na linguagem Java no ambiente de desenvolvimento NetBeans 5.5. Foram utilizados os componentes gráficos disponibilizados pelo ambiente para criação da interface com o usuário. O código fonte a ser implementado

pelo desenvolvedor, na utilização do *driver* JDBC desenvolvido é considerado simples e de fácil entendimento. O Quadro 19 mostra um trecho do código fonte implementado.

```
//implementação do evento do botão consultar
try{
    // registrar o driver
    Class.forName("br.furb.jdbc.JGDriver");
    Connection con  =
               DriverManager.getConnection("jdbc:jg//"+nome_arquivo);
    Statement    st = con.createStatement();
    ResultSet    rs = st.executeQuery(SQL);
    ResultSetMetaData meta = rs.getMetaData();
    while (rs.next()) {
            ...
    }
}catch(SQLException e ){
       JOptionPane.showMessageDialog(this, e);
}catch(ClassNotFoundException e){
       JOptionPane.showMessageDialog(this, e.getMessage());
}
```

Quadro 19 – Código fonte do sistema de validação

A variável `SQL` é carregada através da interface com o usuário, e refere-se ao quadro onde é informada a consulta global. Deve-se passar como parâmetro para o `Class.forName` o *driver* desenvolvido. Nota-se que a implementação da conexão com o banco de dados não difere em nada da conexão em que se utiliza, por exemplo, um *driver* JDBC do MySQL.

3.3.5 Operacionalidade da implementação

Nesta seção é apresentada a operacionalidade do sistema de validação, conectando-se a três bases de dados distintas. Inicialmente é informado o caminho do arquivo a ser utilizado na realização das consultas e o comando SQL para execução das mesmas.

A figura 17 apresenta um exemplo do sistema de validação, mostrando o retorno de uma consulta em três BDs relacionados no ECG.

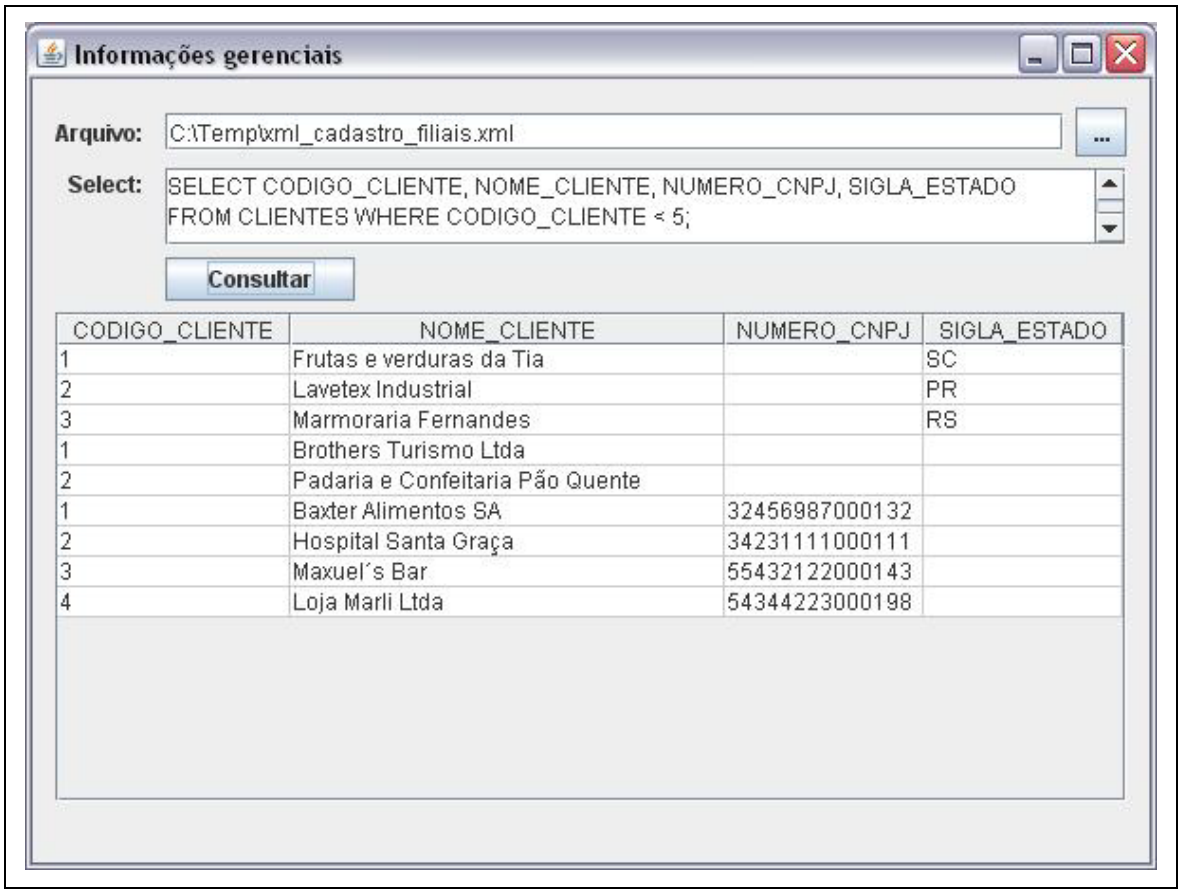


Figura 17 – Tela do sistema de validação

As consultas realizadas em cada BD são apresentadas de forma transparente para o usuário, sendo que são realizadas conexões em vários BDs, conforme ECG cadastrados, podendo apresentar um certo custo de tempo devido a qualidade do sinal, distância entre os sítios e disponibilidades dos BDs envolvidos.

Para realização dos testes, foram criadas tabelas de clientes em três bases de dados de fabricantes distintos. O Quadro 20 mostra a tabela global `clientes` mapeada para a tabela local `cliente` de um banco em Oracle.

Cd_cliente	nm_cliente	nr_cnpj	ds_endereco
1	Baxter Alimentos SA	3245698700013	Rua das palmeiras, 54 – SC
2	Hospital Santa Graça	3423111100011	Avenida Pedro Paulo, 132 – PR
3	Maxuel´s Bar	5543212200014	Rua Quintino Bocaiúva, 601 – SP
4	Loja Marli Ltda	5434422300019	Rua Duarte Schuttel, 75 – MG

Quadro 20 – Tabela `cliente` de um banco Oracle

O Quadro 21 mostra a tabela global `clientes` mapeada para a tabela local `tab_clientes` de um banco em Firebird.

cod_cli	nom_cli	cod_cpf	nom_cidade	sg
1	Frutas e verduras da Tia	98762341101	Blumenau	SC
2	Lavetex Industrial	29320939219	Londrina	PR
3	Marmoraria Fernandes	12324432566	Porto Alegre	RS

Quadro 21 – Tabela `tab_clientes` de um banco Firebird

Nota-se que os nomes das colunas e tabelas são diferentes e que algumas colunas que estão em uma tabela não necessariamente estão contidas na outra. O Quadro 22 mostra a tabela global `clientes` mapeada para a tabela local `clientes` de um banco em MySQL.

cd_cli	nm_cli	ds_endereco
1	Brothers Turismo Ltda	Rua Pedro Souza, 28
2	Padaria e Confeitaria Pão Quente	Rua Malasarte Duarte, 1231
16	Oficina de veículos Barni	Rua Almirante Tamandaré, 48

Quadro 22 – Tabela `clientes` de um banco em MySQL

3.4 RESULTADOS E DISCUSSÃO

O desenvolvimento do *driver* é considerado complexo, devido ao alto grau de interação com os *drivers* específicos de cada BD envolvido na realização das consultas. Segue abaixo, os resultados alcançados e não alcançados com o desenvolvimento do *driver*:

a) utilização: se torna fácil, sendo que o desenvolvedor utiliza as mesmas técnicas de programação com relação à utilização do *driver* de um fabricante qualquer, sendo que as classes, os métodos e as interfaces chamadas no programa principal são as mesmas;
b) desempenho: foram realizadas conexões em três BDs instalados localmente, e o tempo de execução é considerado rápido, muito próximo a execução de uma consulta numa ferramenta de *front-end* de qualquer BD envolvido. Porém, muitos fatores podem influenciar no tempo de execução das consultas, como recursos de rede, disponibilidade do servidor, quantidade de registros nas tabelas, entre outros.

Em relação ao trabalho de conclusão de Gianisini Júnior (2006), explorou-se a técnica de replicação de dados em bases heterogêneas, provando o funcionamento nos SGBDs MsSQL Server e MySQL. A heterogeneidade foi alcançada aderindo a utilização da

tecnologia Hibernate para persistência dos dados, que é compatível com vários SGBDs, permitindo que sejam facilmente configurados em sua arquitetura. O *driver* JDBC desenvolvido no presente trabalho utiliza a técnica de fragmentação horizontal de dados, sem nenhuma tecnologia extra para persistência de dados.

4 CONCLUSÕES

Através do desenvolvimento deste trabalho foi possível adquirir conhecimentos sobre BDDs, especificamente no que diz respeito a processamento de consultas, e constatou-se ainda que muito pode ser explorado, pois a utilização de sistemas distribuídos nas empresas cresce diariamente.

O *driver* JDBC implementado conseguiu alcançar os objetivos desejados, demonstrando a praticidade de realizar consultas em vários BDs, aplicando a técnica de fragmentação horizontal. Estas consultas podem ser locais ou remotas, parecendo de forma transparente para o usuário e para o programador.

As funções de agrupamento e ordenação são executadas localmente e não aparecem de forma integrada para o usuário. Por exemplo, na utilização de uma função de agrupamento em uma consulta, esta será executada localmente nos BDs relacionados e a visão global dos resultados será apresentada para o usuário de forma não agrupada.

Com a utilização de uma ferramenta de edição de ECGs foi possível desenvolver um documento XML que possa ser interpretado pelo *driver*, facilitando a construção das consultas locais. Os processos de fragmentação de dados se tornam transparentes após a configuração destes documentos, sendo realizadas a montagem das consultas nestas configurações.

A ferramenta de edição de ECGs também se demonstrou muito útil, pois o administrador facilmente cadastra os BDs, a definição de tabelas globais e o relacionamento entre os mesmos.

Por fim, este trabalho teve como objetivo maior resolver problemas quanto ao processamento de consultas em BDs heterogêneos, e que os dados sejam confiáveis, isto é, completos, considerando todos os BDs envolvidos.

4.1 EXTENSÕES

Como sugestão para trabalhos futuros, tem-se as seguintes:

a) permitir aceitar comandos de *insert*, *update* e *delete*;

b) desenvolvimento de técnicas para tratamento de erros de conexão e falhas na comunicação entre os sítios;
c) implementar o método *preparedStatement* para realização das consultas;
d) desenvolver um método para permitir que as cláusulas *group by, order by* e *where* sejam consideradas globalmente. Neste trabalho, estas cláusulas restringem-se às consultas locais;
e) implementar a utilização de *join* nas consultas globais;
f) permitir a utilização de *database link*.
g) implementar um recurso para que comandos *select* específicos de um BD possam ser mapeados e executados em outros BDs, atendendo suas particularidades.

O conceito sobre o desenvolvimento da maioria dos itens citados acima podem ser obtidos em GARCIA-MOLINA (2001).

REFERÊNCIAS BIBLIOGRÁFICAS

BORCHARDT, Christiano Marcio. **Desenvolvimento de um mecanismo gerenciador de transações para sistemas distribuídos**. 2002. 89 f. Trabalho de Conclusão do Curso (Bacharelado em Ciência da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

CASANOVA, Marco Antonio. **Princípios de sistemas de gerência de bancos de dados distribuídos.** São Paulo: Campus, 1985.

CERÍCOLA, Vicento Oswald. **Oracle:** banco de dados relacional e distribuído. São Paulo: Makron Books, 1995.

COAD, Peter; MAYFIELD, Mark. **Projeto de sistemas em Java**: construindo aplicativos e melhores *applets*. São Paulo: Makron Books, 1998.

COUCEIRO, Luiz Antonio Carneiro da Cunha; BARRENECHA, Hugo Fernando Spencer. **Sistemas de gerência de banco de dados distribuídos.** Rio de Janeiro: LTC, 1984.

FURTADO JÚNIOR, Miguel Benedito. **XML.** Rio de Janeiro, 2003. Disponível em <http://www.gta.ufrj.br/grad/00_1/miguel/index.html>. Acesso em: 28 set. 2007.

GARCIA-MOLINA, Hector; ULLMAN, Jeffrey D; WIDOM, Jennifer. **Implementação de sistemas de bancos de dados.** Rio de Janeiro: Campus, 2001.

GIANISINI JÚNIOR, João Batista. **Desenvolvimento de um framework para replicação de dados entre bancos heterogêneos**. 2006. 101 f. Trabalho de Conclusão do Curso (Bacharelado em Ciência da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

GROTT, Marcio Carlos. **Reutilização de soluções com patterns e frameworks na camada de negócio**. 2003. 116 f. Trabalho de Conclusão do Curso (Bacharelado em Ciência da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

HÜBNER, Jomi Fred. **Acesso a bancos de dados em Java (JDBC).** Blumenau, [2005?]. Disponível em: <http://www.inf.furb.br/~jomi/java/pdf/jdbc.pdf >. Acesso em: 18 abr. 2007.

MAIA, Maurício M. **Tutorial XML Schema.** Campinas, 2005. Disponível em: <http://www.dicas-l.com.br/dicas-l/20050326.php >. Acesso em: 12 set. 2007.

MARCHAL, Benoit. **XML conceitos e aplicação.** Tradução Daniel Vieira. São Paulo: Berkeley Brasil, 2000.

ÖZSU, Tamer; VALDURIEZ, Patrick. **Princípios de sistemas de bancos de dados distribuídos**. Tradução Vandenberg D. de Souza. 2. ed. Rio de Janeiro: Campus, 2001.

PAGE-JONES, Meilir. **Fundamentos do desenho orientado a objeto com UML**. São Paulo: Makron Books, 2001.

RAMON, Fábio. **JDBC 2:** acesso a banco de dados utilizando a linguagem Java. Guia de consulta rápida. Rio de Janeiro: Novatec, 2001.

SANTOS, Ismael H. F. **Interface com banco de dados JDBC**. Rio de Janeiro, 2005. Disponível em: <http://www.tecgraf.puc-rio.br/~ismael/Cursos/XJavaDatabase/aulas/1-JDBC/JavaDatabase_1-JDBC.pdf>. Acesso em: 12 out. 2007.

STONEBRAKER, Michael. **Readings in database systems**. San Mateo: Morgan Kaufmann, 1988.

TESCH JÚNIOR, José Roberto. **XML Schema.** Florianópolis: Visual Books, 2002.

THOMPSON, Marco A. **Java 2 & banco de dados:** aprenda na prática a usar Java e SQL para acessar bancos de dados relacionais. 4. ed. São Paulo: Érica, 2002.

TITTEL, Ed. **Teoria e problemas do XML.** Porto Alegre: Bookman, 2003.

VELOSO, René Rodrigues. **Java e XML:** processamento de documentos XML com Java. Guia de consulta rápida. São Paulo: Novatec, 2003.

APÊNDICE A – Implementação das ações semânticas

A seguir é apresentado o código fonte da classe `Semântico`. Esta classe é responsável pela análise semântica do comando SQL que deverá ser gerado e executa em cada BD através da classe `JGStatement`.

```
public class Semantico implements Constants {

    private ArrayList<SqlTabela> tabelas         = new ArrayList();
    private ArrayList<SqlColuna> colunas         = new ArrayList();
    private LinkedHashMap<String, SqlTabela>tabCol = new LinkedHashMap();
    private ArrayList            colunasSql      = new ArrayList();
    private String               nmTabela        = "";
    private String               aliasTabela     = "";
    private int                  acaoPai;
    private SqlTabela            tab;
    private String               nmColuna        = "";
    private String               sql             = "";
    private boolean              ehAlias         = false;
    private boolean              ehFrom          = true;

    public void executeAction(int action,Token token) throws SemanticError
    {
        switch (action) {
            case 1:{ // expressão select
                setSql(getSql() + (" "+token.getLexeme()+" "));
                break;
            }
            case 2: { // colunas e alias da tabela
                //System.out.println(token.getLexeme());
                setSql(getSql() + (token.getLexeme()+"#"));
                if(ehAlias){
                    //Trocar o alias pela coluna
                    trocarColuna(token.getLexeme());
                    ehAlias = false;
                }else
                    //Verificar se a coluna já existe
                    if(!containsColuna(token.getLexeme(), colunas)){
                        //Armazenar a coluna
                        SqlColuna   sqlColuna = new SqlColuna();
                        sqlColuna.setNmColuna(token.getLexeme());
                        sqlColuna.setAlias("@");
                        colunas.add(sqlColuna);
                         if(ehFrom)
                            getColunasSql().add(token.getLexeme());
                    }
                break;
            }
            case 3:{ // alias coluna
                setSql(getSql() + (" "+token.getLexeme()+" "));
                if(!colunas.isEmpty()){
                    addAliasColuna(token.getLexeme());
                    if(ehFrom){
                        int pos = getColunasSql().size() - 1;
                        getColunasSql().set(pos, token.getLexeme());
                    }
```

```
                }
                break;
            }
            case 4: { // tabelas
                setSql(getSql() + (" "+token.getLexeme()+"# "));
                //aki - Verificar se a tabela já existe
                SqlTabela sqlTab = new SqlTabela();
                sqlTab.setNmTabela(token.getLexeme());
                sqlTab.setAlias("@");
                getTabelas().add(sqlTab);
                break;
            }
        /*  case 5: { // alias coluna
                if(!getListaTabelas().isEmpty()){
                    aliasTabela = token.getLexeme();
                }
                //gravar no array list das tabelas
                break;
            } */
            case 6: { // alias tabela
                setSql(getSql() + (" "+token.getLexeme()+"# "));
                int         pos     = getTabelas().size() - 1;
                SqlTabela   sqlTab  = (SqlTabela)getTabelas().get(pos);
                sqlTab.setAlias(token.getLexeme());
                getTabelas().set(pos, sqlTab);
                break;
            }
            case 9:{ // from
                setSql(getSql() + (" "+token.getLexeme()+" "));
                ehFrom = false;
                break;
            }
            case 10:{ // where
                setSql(getSql() + (" "+token.getLexeme()+" "));
                break;
            }
            case 11:{ // group by
                setSql(getSql() + " group by ");
                break;
            }
            case 12:{ // having
                setSql(getSql() + (" "+token.getLexeme()+" "));
                break;
            }
            case 13:{ // order by
                setSql(getSql() + " order by ");
                break;
            }
            case 14:{ // union
                setSql(getSql() + " union ");
                break;
            }
            case 50:{ //  like
                setSql(getSql() + (" "+token.getLexeme()+" "));
                break;
            }
            case 51:{ // between
                setSql(getSql() + (" "+token.getLexeme()+" "));
                break;
            }
```

```
            case 52:{ // and
                setSql(getSql() + (" "+token.getLexeme()+" "));
                break;
            }
            case 53:{ // in
                setSql(getSql() + (" "+token.getLexeme()+" "));
                break;
            }
            case 54:{ // or
                setSql(getSql() + (" "+token.getLexeme()+" "));;
                break;
            }
            case 55:{ // upper
                setSql(getSql() + token.getLexeme());
                break;
            }
            case 56:{ // month
                setSql(getSql() + token.getLexeme());
                break;
            }
            case 57:{ // year
                setSql(getSql() + token.getLexeme());
                break;
            }
            case 90:{ // sinal de "=""
                setSql(getSql() + (" "+token.getLexeme()+" "));
                break;
            }
            case 91:{ // sinal de "<""
                setSql(getSql() + (" "+token.getLexeme()+" "));
                break;
            }
            case 92:{ // sinal de ">""
                setSql(getSql() + (" "+token.getLexeme()+" "));
                break;
            }
            case 93:{ // sinal de "<=""
                setSql(getSql() + (" "+token.getLexeme()+" "));
                break;
            }
            case 94:{ // sinal de ">=""
                setSql(getSql() + (" "+token.getLexeme()+" "));
                break;
            }
            case 95:{ // sinal de "<>""
                setSql(getSql() + (" "+token.getLexeme()+" "));
                break;
            }
            case 98:{ // asterisco
                setSql(getSql() + "asterisco#");
                if(!ehAlias){
                    SqlColuna   sqlColuna = new SqlColuna();
                    sqlColuna.setNmColuna("asterisco");
                    sqlColuna.setAlias("@");
                    colunas.add(sqlColuna);
                }else{
                    trocarColuna("asterisco");
                    ehAlias = false;
                }
                break;
```

```
            }
            case 99:{ // sinal de "."
                setSql(getSql() + token.getLexeme());
                ehAlias = true;
                break;
            }
            case 100:{ // vírgula
                setSql(getSql() + (token.getLexeme()+" "));
                break;
            }
            case 101:{ // sinal de "("
                setSql(getSql() + token.getLexeme());
                break;
            }
            case 102:{ // sinal de ")"
                setSql(getSql() + token.getLexeme());
                break;
            }
            case 103:{ // is
                setSql(getSql() + (" "+token.getLexeme()+" "));
                break;
            }
            case 104:{ // not
                setSql(getSql() + (" "+token.getLexeme()+" "));
                break;
            }
            case 105:{ // sinal de "-"
                setSql(getSql() + (token.getLexeme()+" "));
                break;
            }
            case 200:{ // distinct
                setSql(getSql() + (token.getLexeme()+" "));
                break;
            }
            case 201:{ // all
                setSql(getSql() + (" "+token.getLexeme()+" "));
                break;
            }
            case 202:{ // null
                setSql(getSql() + (" "+token.getLexeme()+" "));
                break;
            }
            case 300:{ // count
                setSql(getSql() + token.getLexeme());
                break;
            }
            case 301:{ // sum
                setSql(getSql() + token.getLexeme());
                break;
            }
            case 302:{ // max
                setSql(getSql() + token.getLexeme());
                break;
            }
            case 303:{ // min
                setSql(getSql() + token.getLexeme());
                break;
            }
            case 304:{ // avg
                setSql(getSql() + token.getLexeme());
```

```
                break;
            }
            case 305:{ // desc
                setSql(getSql() + (" "+token.getLexeme()+" "));
                break;
            }
            case 306:{ // asc
                setSql(getSql() + (" "+token.getLexeme()+" "));
                break;
            }
            case 307:{ // integer_
                setSql(getSql() + (" "+token.getLexeme()+" "));
                break;
            }
            case 308:{ // float_
                setSql(getSql() + token.getLexeme());
                break;
            }
            case 309:{ // parâmetro dentro do like
                setSql(getSql() + token.getLexeme());
                break;
            }
        }
    }

     /** Junta as colunas nas respectivas tabelas */
    public ArrayList    getListaTabelas(){
        for(int i=0; i < tabelas.size(); i++){
            SqlTabela   sqlTab = (SqlTabela)tabelas.get(i);
            for(SqlColuna sqlCol: colunas){
                if(sqlTab.getAlias().equalsIgnoreCase(sqlCol.getAlias()))
                    sqlTab.addColuna(sqlCol);
            }
            tabelas.set(i, sqlTab);
        }
        return tabelas;
    }

    /** Verificar se a coluna já existe */
    private boolean containsColuna(String coluna, ArrayList colunas){
        boolean achou = false;
        int i = 0;
        while(i < colunas.size() && !achou ){
            SqlColuna col = (SqlColuna)colunas.get(i);
            if(col.getNmColuna().equalsIgnoreCase(coluna)){
                achou = true;
            }else
                i++;
        }
        return achou;
    }

    /** Trocar o alias pela coluna */
    private void trocarColuna(String coluna){

        int         pos         = colunas.size() -1;
        SqlColuna   sqlColuna   = (SqlColuna)colunas.get(pos);
        String      alias       = sqlColuna.getNmColuna();

        if(!containsColunaAlias(coluna, alias)){
```

```
            sqlColuna.setNmColuna(coluna);
            sqlColuna.setAlias(alias);
            colunas.set(pos, sqlColuna);
            if(coluna.equalsIgnoreCase("asterisco") &&
                !alias.equalsIgnoreCase("@")){
                int posColSql = getColunasSql().size() - 1;
                getColunasSql().remove(posColSql);
            }
        } else
            colunas.remove(pos);
    }

    private boolean containsColunaAlias(String coluna, String alias){
        int         i       = 0;
        boolean     achou   = false;
        while(colunas.size() > i && !achou){
            SqlColuna sqlCol = (SqlColuna)colunas.get(i);
            if(sqlCol.getNmColuna().equalsIgnoreCase(coluna) &&
                sqlCol.getAlias().equalsIgnoreCase(alias))
                achou =true;
            else
                i++;
        }
        return achou;
    }

    /** Adiciona as colunas no Sqlcoluna*/
    private ArrayList addColuna(String coluna, String alias, ArrayList
                                colunas, int acao){
        if(colunas == null)
            colunas = new ArrayList();
        SqlColuna col = new SqlColuna();
        col.setNmColuna(coluna);
        col.setAlias(alias);
        col.setAcao(acao);
        colunas.add(col);
        return colunas;
    }

    /** Adiciona o aliasColuna na coluna */
    private void addAliasColuna(String aliasColuna){
        int         pos     =   colunas.size() - 1;
        SqlColuna   coluna  =   colunas.get(pos);
        coluna.setAliasColuna(aliasColuna);
        colunas.set(pos, coluna);
    }
```