ADRIANO DOS SANTOS FERNANDES

INTEGRAÇÃO DA JVM AO SGBD FIREBIRD PARA EXECUÇÃO DE FUNÇÕES, *STORED PROCEDURES* E *TRIGGERS* FEITOS EM JAVA

Resumo

Neste trabalho foi implementada uma integração entre o SGBD (Sistema Gerenciador de Banco de Dados) Firebird e a JVM (*Java Virtual Machine*), tornando possível a criação de funções, *stored procedures* e *triggers* do Firebird na linguagem Java. A integração é realizada carregando-se a JVM por JNI (*Java Native Interface*) dentro do processo do Firebird. Através do mapeamento de rotinas SQL para métodos estáticos Java e arquivos JAR (*Java Archive*) disponibilizados pelo usuário, o Firebird consegue chamar estes métodos presentes nas classes. O código do usuário pode executar qualquer operação possível em Java e ainda pode chamar o Firebird através de seu *driver* JDBC (*Java Database Connectivity*), inclusive podendo realizar operações dentro do contexto da transação iniciada pela aplicação cliente. A integração também provê o isolamento das classes de diferentes bancos de dados e une os sistemas de segurança da plataforma Java com o sistema de segurança do Firebird.

Palavras-chave: Java, Firebird, Banco de Dados.

Lista de Ilustrações

Lista de Tabelas

Lista de Abreviaturas

2PC – Two Phase Commit
ABI – Application Binary Interface
ACID – Atomicidade, Consistência, Isolamento e Durabilidade
API – Application Program Interface
BLOB – Binary Large Object
COM – Component Object Model
CSV – Comma Separated Values
DDL – Data Definition Language
DML – Data Manipulation Language
IDPL – Initial Developer Public License
IP – Internet Protocol
IPL – InterBase Public License
JAAS – Java Authentication and Authorization Service
JAR – Java Archive
JCA – Java Connector Architecture
JDBC – Java Database Connectivity
JNI – Java Native Interface
JVM – Java Virtual Machine
KB – Kilobytes (1024 bytes)
PSQL – Procedural Structured Query Language
SGBD – Sistema Gerenciador de Banco de Dados
SO – Sistema Operacional
SQL – Structured Query Language
TCP – Transmission Control Protocol
URL – Uniform Resource Locator
UTF – Unicode Transformation Format
XML – Extensible Markup Language

Sumário

Capítulo 1

1. Introdução

1.1 Justificativa e Motivação

Apesar do padrão SQL (*Structured Query Language*) definir uma linguagem para codificação de funções, *stored procedures* e *triggers*, poucos Sistemas Gerenciadores de Bancos de Dados (SGBD) a usam. A maioria dos SGBD possui sua própria linguagem a ser usada para este fim. Isto se deve ao fato das primeiras versões dos principais SGBD atualmente disponíveis antecederem este padrão, que foi introduzido como um adendo ao SQL-92 (Taylor, 2003), e da necessidade de construções na linguagem para manipulação de recursos específicos de cada SGBD.

Este fato faz com que aplicativos de bancos de dados escritos para um SGBD tenham que ser reescritos caso seja necessária a troca do SGBD. Isto faz com que certas empresas e desenvolvedores, que queiram escrever aplicativos de bancos de dados independentes de SGBD, implementem os aplicativos sem utilizar estes importantes recursos. Ao abster-se destes recursos, o aplicativo geralmente perde desempenho e a integridade do banco de dados pode ser comprometida. A queda de desempenho pode decorrer-se devido à necessidade de transferência para manipulação de grande quantidade de dados entre processos diferentes ou até mesmo máquinas diferentes em uma rede. A integridade pode ser comprometida, pois o

banco de dados não terá *triggers* definidos para este fim, necessitando que cada aplicação que conecte-se a este banco de dados defina as regras de integridade do mesmo.

Já os desenvolvedores que optam em usar as linguagens dos SGBD às vezes se encontram em situações difíceis. Estas linguagens são linguagens de propósito específico, para manipulação de dados. Muitas vezes surge a necessidade de acrescentar processos, como envio de e-mails ou execução de outros aplicativos, que não são atingíveis com o uso destas linguagens. Isto faz com que os desenvolvedores tenham que alterar a arquitetura dos sistemas, movendo códigos de dentro do banco de dados para as aplicações cliente.

Ao mesmo tempo em que os SGBD tornam-se cada vez mais divergentes e ainda assim limitados pelas suas linguagens de propósito específico, uma excelente linguagem de programação de propósito geral, com ótimo suporte a aplicativos de banco de dados, se torna a cada dia mais popular: Java. Devido às suas características, como robustez, segurança e portabilidade, Java fez e faz sucesso tanto no lado cliente quanto no lado servidor. Estas características tornam Java um potencial candidato à união dos SGBD com uma única linguagem de programação, evitando os transtornos causados aos desenvolvedores de aplicativos.

1.2 Objetivos

Neste trabalho será desenvolvido e integrado ao *engine* do SGBD Firebird um módulo (baseado em interfaces extensíveis C++) de execução de rotinas externas independentes de linguagem, que possa fazer uso de todos os recursos do Firebird neste contexto. Com base neste módulo, será desenvolvido e integrado ao projeto Jaybird (driver JDBC do Firebird) um *plugin* para execução de rotinas feitas em Java.

A interface pública do *plugin* Firebird/Java será baseada no padrão ISO SQL/JRT 2008 (ISO/IEC 9075-13:2008 - *SQL Routines and Types Using the Java Programming Language*) e padrões de fato usados por outros SGBD, considerando a facilidade e possibilidades de uso, a portabilidade de código entre diferentes SGBD, bem como as diferenças entre os mesmos. O resultado final deverá ser suficientemente simples para uso em situações corriqueiras como a codificação de rotinas com funcionalidades específicas, porém suportará situações mais complexas, como a criação de novas interfaces inteiramente em Java para integração de linguagens de *script* que rodem dentro desta plataforma.

1.3 Organização da Monografia

No capítulo 2 serão apresentados o SGBD Firebird e seus principais recursos que o torna o SGBD escolhido para este trabalho. No capítulo 3 serão apresentados a linguagem de programação Java e seus principais recursos relevantes para a integração com um SGBD. O capítulo 4 terá uma descrição detalhada do desenvolvimento do projeto, e o capítulo 5 apresentará as conclusões deste trabalho.

Capítulo 2

2. Firebird

2.1 O Firebird

O Firebird é um SGBD relacional, multiplataforma, escalável, de uso simples, gratuito e de código aberto, disponibilizado sob as licenças InterBase Public License (IPL) e Initial Developer Public License (IDPL). Estas licenças permitem seu uso e distribuição, inclusive junto a aplicações comerciais, sem qualquer custo. O Firebird é derivado da versão 6.0 *beta* do SGBD Borland InterBase e teve seu início em 31 de julho de 2000, após a abertura do código do InterBase pela Borland e divergências desta companhia com a comunidade de usuários sobre a maneira em que o desenvolvimento do InterBase seria continuado sem a participação direta da comunidade. O desenvolvimento do InterBase teve seu início em 1984, o que mostra que o Firebird possui um código confiável, maduro e robusto suficiente para sobreviver a 25 anos (Firebird History, 2009).

Seus principais recursos incluem transações com as características ACID (Atomicidade, Consistência, Isolamento e Durabilidade) e 2PC (*Two Phase Commit)*, suporte a *stored procedures* e *triggers*, suporte a funções definidas pelo usuário, visões, integridade referencial, arquitetura multigeracional, recuperação rápida após falhas, simplicidade na instalação e na manutenção de bancos de dados, versão embutida ou cliente-servidor, *drivers*

gratuitos e comerciais de acesso em linguagens C, C++, Delphi, PHP, Python e Perl, para plataformas Java e .NET, bem como *drivers* ODBC e OLEDB para uso em linguagens e plataformas compatíveis com estas tecnologias.

Segundo as estatísticas do *site* SourceForge.net[1], que hospeda o Projeto Firebird, o Firebird (incluindo todos os seus subprojetos) já foi *baixado* mais de 6.200.000 vezes, sendo mais de 1.200.000 vezes apenas no ano de 2008. No Brasil, empresas renomadas como Banco do Brasil[2] e Caixa Econômica Federal (Freitas, Paris, 2007) utilizam o Firebird.

O planejamento e desenvolvimento do Firebird é realizado de forma descentralizada, através da Internet, com o auxílio de listas de discussões públicas com a participação da comunidade de usuários. Seu núcleo, feito na linguagem C++, é atualmente desenvolvido principalmente na Rússia, Ucrânia e Brasil. O Projeto Firebird tem uma permanente porta aberta a bons programadores C++ que queiram contribuir ao seu desenvolvimento (Borrie, 2004).

O Projeto Firebird conta com o auxílio da *Firebird Foundation*. Segundo (Bitzer, Schröder, 2006), a *Firebird Foundation* existe para prover uma infraestrutura não-comercial com mecanismos para aceitar e gerenciar recursos financeiros e encorajar a cooperação e afiliação com indivíduos, outras instituições sem fins lucrativos e companhias comerciais. Os fundos arrecadados são repassados aos desenvolvedores para trabalharem estendendo, testando e melhorando o Firebird (Borrie, 2004).

O Projeto Firebird é dividido em subprojetos. Seus principais subprojetos são:

- *Core*
- *Driver* JDBC para a plataforma Java
- *Driver* .NET

1 <http://sourceforge.net/project/stats/detail.php?group_id=9028&ugn=firebird&type=prdownload&mode=alltime&package_id=0>. Acesso em 08 mar. 2009.
2 <http://office.bb.com.br/office/plugins/frame6.html>. Acesso em 08 mar. 2009.

- *Driver* ODBC

O *Core* (núcleo) é o principal subprojeto do Firebird. Nele são desenvolvidos o *engine*, o servidor para acesso cliente-servidor, a biblioteca cliente em linguagem C e ferramentas básicas para administração, manutenção e desenvolvimento.

2.2 Modos de Operação

O *engine* do Firebird pode ser executado em quatro modos diferentes: *Classic*, *SuperServer*, *SuperClassic* e *Embedded*.

O modo *Classic* foi o primeiro modo suportado pelo InterBase, devido à falta da existência de sistemas operacionais com suporte a *threads* de execução na época de sua criação. Neste modo, para cada conexão cliente é iniciado um processo servidor distinto para gerenciá-la. Cada conexão possui *caches* distintos para metadados e páginas de dados. Através de comunicação interprocesso o Firebird coordena os acessos e gravações aos bancos de dados para a correta operação. Este modo pode ser melhorado, através de um gerenciador de *locks* distribuído, para que futuramente o Firebird funcione em *clusters*.

O modo *SuperServer* funciona com um processo único, fazendo uso de *threads* para o gerenciamento de conexões simultâneas. Cada banco de dados possui um *cache* único para metadados e páginas de dados compartilhados por todas as conexões a este banco. Teoricamente, este modo deveria ser mais escalável que o *Classic*, devido a *threads* consumirem menos recursos que processos, a não redundância de *caches* e a ausência de comunicação interprocesso. Porém, devido a detalhes de implementação na sincronização para acesso aos *caches*, este modo ainda não funciona bem em máquinas com múltiplos processadores. Um banco de dados usado por um processo *SuperServer* não pode ser acessado simultaneamente por outro processo que use este ou outro modo.

O modo *SuperClassic* foi introduzido na versão 2.5 do Firebird com o objetivo de melhorar incrementalmente a escalabilidade do mesmo. Este modo funciona com um processo único e múltiplos *threads* (como o *SuperServer*) mas com *caches* distintos para cada conexão (como o *Classic*), unindo assim as vantagens destes dois diferentes modos. Este modo é temporário e tende a desaparecer na versão 3.0, pois são planejadas[1] melhorias a sincronização de acessos aos *caches* do modo *SuperServer*, tornando-o desnecessário.

Diferente dos outros modos, o modo *Embedded* funciona em uma biblioteca, e não em uma aplicação. Esta biblioteca pode ser carregada por aplicações para acessar bancos de dados locais sem a necessidade de instalação do servidor Firebird. Até a versão 2.1, o modo *Embedded* é baseado no modo *SuperServer* na plataforma Windows e baseado no modo *Classic* nas outras plataformas. A partir da versão 2.5 este modo é baseado no Super*Classic* em todas as plataformas, permitindo assim o acesso a um determinado banco de dados por várias aplicações (processos) diferentes.

Devido a detalhes de implementação e diferenças entre as plataformas, a partir da versão 2.5, no Windows, os servidores *Classic* e *SuperClassic* são implementados em uma mesma aplicação e em duas aplicações diferentes nas outras plataformas. No Windows, o *engine* é compilado diretamente na aplicação e nas outras plataformas as duas aplicações fazem uso da biblioteca *Embedded*.

2.3 A Linguagem PSQL

O Firebird possui uma linguagem de programação estruturada chamada Procedural SQL (PSQL). A linguagem PSQL suporta comandos para declaração de variáveis, tomada de decisões (*if-then-else*), controle de fluxo (*while*), tratamento de exceções, demarcação de

1 <http://www.firebirdsql.org/index.php?op=devel&sub=engine&id=roadmap_2009&nosb=1>. Acesso em 08 mar. 2009.

transações, notificação de eventos, execução de procedimentos e funções, manipulação de cursores SQL e execução de comandos SQL. O uso da linguagem PSQL é suportado em três diferentes contextos: em *stored procedures*, em *triggers* e em blocos de código executados pela aplicação cliente através do comando *EXECUTE BLOCK*. Comandos PSQL são escritos entre os comandos *BEGIN* e *END*, podendo ser precedidos por declarações (comando *DECLARE*) de variáveis. A figura 1 apresenta a execução de um bloco de código PSQL que retorna os dez primeiros termos da sequencia de Fibonacci.

```
SQL> execute block returns (n integer)
CON> as
CON>   declare termos integer = 10;
CON>   declare n1 integer = 0;
CON>   declare n2 integer = 1;
CON> begin
CON>   n = 0;
CON>
CON>   if (termos >= 1) then
CON>      suspend;
CON>
CON>   if (termos >= 2) then
CON>   begin
CON>      n = 1;
CON>      suspend;
CON>   end
CON>
CON>   while (termos >= 3) do
CON>   begin
CON>     n = n1 + n2;
CON>     suspend;
CON>
CON>     n1 = n2;
CON>     n2 = n;
CON>     termos = termos - 1;
CON>   end
CON> end!

           N
============
           0
           1
           1
           2
           3
           5
           8
          13
          21
          34
```

Figura 1: Execução de um bloco de código PSQL.

2.4 Tipos de Dados

No Firebird, toda coluna, variável ou parâmetro possui um tipo de dado. O tipo de dado limita os valores que podem ser atribuídos a estes itens. A escolha de um tipo de dado adequado é uma importante consideração com relação a tráfego de rede, economia em disco e tamanho de índices (Borrie, 2004). Conversões são realizadas automaticamente quando se atribui uma expressão de um tipo a um outro diferente. Se o usuário desejar realizar uma conversão explícita entre dois tipos, pode-se usar o operador *CAST(expressão AS tipo).*

O Firebird suporta os seguintes tipos de dados:

- SMALLINT – Número inteiro de 16 *bits*;
- INTEGER – Número inteiro de 32 *bits*;
- BIGINT – Número inteiro de 64 *bits*:
- FLOAT – Número ponto flutuante de 32 *bits*:
- DOUBLE PRECISION – Número ponto flutuante de 64 *bits*;
- NUMERIC (precisão, escala) – Número em ponto fixo com uma determinada precisão mínima e escala;
- DECIMAL (precisão, escala) - Número em ponto fixo com uma determinada precisão mínima e escala;
- DATE – Data (dia, mês e ano);
- TIME – Horário (horas, minutos, segundos e fração de segundos);
- TIMESTAMP – Data e horário;
- CHAR (comprimento) – *String* de caracteres de determinado comprimento. No caso de uma *string* de comprimento menor ser atribuída a este tipo, o comprimento é complementado com espaços;
- VARCHAR – *String* de caracteres com um determinado comprimento máximo;

- BLOB – Texto ou *bytes* de comprimento desconhecido.

Os tipos *SMALLINT*, *INTEGER* e *BIGINT* representam números com sinal, variando apenas a precisão. São armazenados na forma nativa da plataforma, geralmente na notação "complemento de dois". Os tipos *FLOAT* e *DOUBLE PRECISION* representam números na notação "ponto flutuante" nativa da plataforma, divididos em mantissa e expoente, variando a precisão. Os tipos *NUMERIC* e *DECIMAL* são funcionalmente idênticos e armazenam números sob forma inteira (de 16 a 64 bits, dependendo da precisão) com uma determinada escala (quantidade de dígitos após o ponto decimal). Os tipos *DATE* e *TIME* armazenam, respectivamente, datas e horários. O tipo *TIMESTAMP* armazena data e horário simultaneamente. Os tipos *CHAR* e *VARCHAR* armazenam *strings* de até 32KB variando o tratamento referente ao comprimento declarado e ao comprimento atribuído. O tipo *BLOB* armazena textos ou *bytes* de comprimento desconhecido e virtualmente ilimitado.

Além dos tipos predeterminados, o Firebird suporta a criação de domínios pelo usuário. Um domínio é uma declaração em que é atribuído um tipo de dado a um determinado nome. Este nome pode então ser usado como um tipo de dado. O tipo de dado de um domínio pode ser alterado, refletindo automaticamente a mudança em todos os itens que usem este nome como tipo de dado.

2.5 Funções

Funções são rotinas armazenadas em um banco de dados. Elas podem ser chamadas por *stored procedures*, *triggers*, outras funções ou diretamente em instruções SQL. As funções possuem zero ou mais parâmetros e um valor de retorno. No Firebird, as funções são implementadas em linguagens nativas (como C ou Pascal) da plataforma e disponibilizadas ao *engine* sob a forma de bibliotecas. Elas então devem ser declaradas no banco de dados

especificando-se o nome do módulo (*MODULE_NAME*) e o ponto de entrada (*ENTRY_POINT*). *MODULE_NAME* especifica o nome da biblioteca que implementa a função e *ENTRY_POINT* especifica o nome da função definida dentro da biblioteca. Uma função pode realizar qualquer ação possível baseada nos parâmetros recebidos, porém possui sérias limitações, pois não pode consultar o banco de dados que a chamou sem que antes realize uma nova conexão que será tratada como uma conexão independente. A figura 2 apresenta a declaração de uma função que recebe dois parâmetros do tipo *INTEGER* e retorna um valor do tipo *DOUBLE PRECISION*.

```
SQL> DECLARE EXTERNAL FUNCTION div
CON>     INTEGER, INTEGER
CON>     RETURNS DOUBLE PRECISION BY VALUE
CON>     ENTRY_POINT 'IB_UDF_div' MODULE_NAME 'ib_udf';
```

Figura 2: Declaração de uma função.

2.6 *Stored Procedures*

Stored procedures são procedimentos armazenados em um banco de dados. São particularmente úteis pois permitem que operações no banco de dados sejam disponibilizadas a aplicações externas sem expor qualquer detalhe interno do banco de dados (Silberschatz, Korth, Sudarshan, 2001). Elas podem ser chamadas por outras *stored procedures*, por *triggers* ou diretamente pela aplicação cliente. *Stored procedures* possuem zero ou mais parâmetros de entrada e zero ou mais parâmetros de saída. Se a quantidade de parâmetros de saída for zero, nenhum registro será retornado. Se houver algum parâmetro de saída, zero ou mais registros podem ser retornados. No Firebird, as *stored procedures* são divididas em duas categorias: executáveis e selecionáveis.

Procedures executáveis são chamadas com o comando *EXECUTE PROCEDURE*. Elas retornam um ou nenhum registro, de acordo com a existência ou não de parâmetros de saída. *Procedures* selecionáveis são chamadas com o comando *SELECT*. Este tipo de

procedure funciona como uma tabela, retornando zero ou mais registros. Uma *procedure* selecionável necessita de no mínimo um parâmetro de saída. A figura 3 apresenta o mesmo algoritmo da figura 1 sob a forma de uma *procedure* selecionável. A variável *termos* foi substituída por um parâmetro de entrada com o mesmo nome. A figura 4 apresenta a execução desta *procedure* retornando os dez primeiros termos da sequencia.

```
SQL> create or alter procedure fibonacci (termos integer)
CON>   returns (n integer)
CON> as
CON>   declare n1 integer = 0;
CON>   declare n2 integer = 1;
CON> begin
CON>   n = 0;
CON>
CON>   if (termos >= 1) then
CON>      suspend;
CON>
CON>   if (termos >= 2) then
CON>   begin
CON>      n = 1;
CON>      suspend;
CON>   end
CON>
CON>   while (termos >= 3) do
CON>   begin
CON>     n = n1 + n2;
CON>     suspend;
CON>
CON>     n1 = n2;
CON>     n2 = n;
CON>     termos = termos - 1;
CON>   end
CON> end!
```

Figura 3: Criação de uma *stored procedure* selecionável em linguagem PSQL.

```
SQL> select * from fibonacci(10);

           N
============
           0
           1
           1
           2
           3
           5
           8
          13
          21
          34
```

Figura 4: Execução de uma *stored procedure* selecionável.

2.7 *Triggers*

Triggers (gatilhos) são procedimentos armazenados chamados pelo SGBD na ocorrência de um determinado evento. *Triggers* são mecanismos úteis para alertar humanos ou iniciar certas tarefas automaticamente quando alguma condição é satisfeita (Silberschatz, Korth, Sudarshan, 2001). No Firebird, os *triggers* são escritos em linguagem PSQL e são divididos em três categorias: *triggers* DML, *triggers* de banco de dados e *triggers* DDL.

Os *triggers de t*odas as categorias possuem as seguintes características: nome, status, sequência e corpo. O nome é o identificador do objeto dentro do banco de dados. O status (*ACTIVE* ou *INACTIVE*) indica se o *trigger* está ativo ou inativo. A sequência indica a ordem de chamada do *trigger* em relação a outros *triggers* aplicáveis ao mesmo evento. O corpo é um bloco de código PSQL onde são implementadas as ações que o *trigger* irá realizar.

Os *triggers* DML são o tipo mais comum de *trigger*. Além das características comuns aos triggers, eles possuem um nome de tabela, um ou mais eventos e uma fase. O nome da tabela indica a tabela que estará associada à execução dos eventos. Os eventos (*DELETE*, *INSERT* ou *UPDATE*) indicam os tipos de operações na tabela que irão disparar o *trigger*. A fase indica o momento do disparo. Os triggers de um determinado tipo de evento são disparados durante a execução de comandos DML como *DELETE, INSERT* e *UPDATE*. O comando SQL *MERGE* pode disparar *triggers* de *INSERT* e *UPDATE*, dependendo das ações realizadas. A fase (*BEFORE* ou *AFTER*) indica o momento em que o *trigger* será disparado. *Triggers BEFORE* são disparados antes da modificação e *AFTER* após as modificações. O Firebird dispara os *triggers* antes ou após a exclusão, alteração ou inserção de cada registro.

A linguagem PSQL possui extensões a serem usadas no corpo de *triggers* DML. Estas extensões incluem as variáveis lógicas *DELETING, INSERTING* e *UPDATING*, que indicam a operação que disparou o *trigger*. Também são disponibilizadas as variáveis *NEW* e *OLD*. Estas variáveis possuem um tipo de dado compatível com a tabela associada ao *trigger*.

Através delas é possível acessar o valor novo e anterior de cada coluna. A figura 5 apresenta um exemplo de *trigger* DML que mantém automaticamente os valores das colunas DATA_CADASTRO e DATA_ALTERACAO para a tabela CLIENTES, usando o *status* padrão *ACTIVE* e a sequencia padrão zero.

```
SQL> create trigger clientes_biu
CON>   before insert or update on clientes
CON> as
CON> begin
CON>   if (inserting) then
CON>      new.data_cadastro = current_date;
CON>
CON>   new.data_alteracao = current_date;
CON> end!
```

Figura 5: Criação de um *trigger* DML.

Os *triggers* de banco de dados são disparados em eventos relacionados a conexões dos usuários ao banco de dados. Estes eventos são: *ON CONNECT, ON DISCONNECT, ON TRANSACTION START, ON TRANSACTION COMMIT* e *ON TRANSACTION ROLLBACK*. O evento *ON CONNECT* ocorre assim que um usuário se conecta ao banco de dados, sendo útil principalmente para rejeição de conexões ou gravação de *logs*. O evento *ON DISCONNECT* ocorre após o usuário solicitar a desconexão ao banco, sendo útil principalmente para registro de horário de desconexões ou da inexistência de desconexão, podendo indicar problemas de segurança ou problemas de infraestrutura. O evento *ON TRANSACTION START* ocorre logo após o início de uma transação. Os eventos *ON TRANSACTION COMMIT* e *ON TRANSACTION ROLLBACK* ocorrem respectivamente antes do *commit* e *rollback* de transações, sendo úteis para implementação de lógicas de negócio e integridade lógica de dados.

Os *triggers* DDL são associados à execução de comandos DDL. Assim como os *triggers* DML, eles possuem um ou mais eventos e uma fase, porém não possuem tabela associada. Os posíveis eventos de *triggers* DDL são: *CREATE TABLE, ALTER TABLE, DROP TABLE, CREATE PROCEDURE, ALTER PROCEDURE, DROP PROCEDURE, CREATE FUNCTION, ALTER FUNCTION, DROP FUNCTION, CREATE TRIGGER,*

ALTER TRIGGER, DROP TRIGGER, CREATE EXCEPTION, ALTER EXCEPTION, DROP EXCEPTION, CREATE VIEW, ALTER VIEW, DROP VIEW, CREATE DOMAIN, ALTER DOMAIN, DROP DOMAIN, CREATE ROLE, ALTER ROLE, DROP ROLE, CREATE SEQUENCE, ALTER SEQUENCE, DROP SEQUENCE, CREATE USER, ALTER USER, DROP USER, CREATE INDEX, ALTER INDEX, DROP INDEX, CREATE COLLATION, DROP COLLATION, ALTER CHARACTER SET, CREATE PACKAGE, ALTER PACKAGE, DROP PACKAGE, *CREATE PACKAGE BODY*, *DROP PACKAGE BODY* e *ANY DDL STATEMENT*. Com exceção do evento *ANY DDL STATEMENT*, os eventos podem ser combinados com a palavra *OR*. Detalhes sobre o disparo do *trigger* podem ser consultados com a função interna *RDB$GET_CONTEXT*. Os parâmetros desta função são: *namespace* e variável. O *namespace* deve ser sempre *DDL_TRIGGER* e a variável pode ser *DDL_EVENT*, *OBJECT_NAME* ou *SQL_TEXT*. A variável *DDL_EVENT* retorna o tipo de evento que disparou o *trigger*. *OBJECT_NAME* retorna o nome do objeto que causou o disparo. *SQL_TEXT* retorna o texto do comando executado que causou o disparo.

2.8 Jaybird

O Jaybird é um *driver* JDBC/JCA para o Firebird, completamente compatível com a especificação JDBC 2.0 (Borrie, 2004). Um *driver* JDBC é uma implementação da API de conectividade a banco de dados da plataforma Java. O Jaybird funciona como *driver* JDBC tipos 2 e 4. Um *driver* JDBC tipo 2 necessita carregar uma biblioteca nativa para se comunicar com o SGBD. O tipo 2 pode ser usado para comunicação com servidores remotos e locais, porém seu principal uso é para conexões com servidores *embedded*. Um *driver* JDBC tipo 4 é um *driver* “puro Java” que pode conectar-se a servidores remotos e locais através do protocolo TCP/IP. O tipo 4 não pode conectar-se a servidores *embedded*.

Capítulo 3

3. Java

3.1 A Linguagem Java

Java é uma linguagem de programação de propósito geral, concorrente, baseada em classes, orientada a objetos, especialmente desenvolvida para ter o mínimo possível de dependências de implementação. Ela permite que o desenvolvedor de aplicativos escreva um programa uma vez e o execute em qualquer lugar na Internet (Gosling *et al.*, 2005). Java é uma linguagem baseada em padrões e independente de plataforma. Java possui o suporte a acesso à banco de dados e computação distribuída em seu núcleo (Reese, 2000).

Java resolve o problema de independência de plataforma através da sua biblioteca de classes padrão, do *bytecode* (arquivo *class*) e da JVM. Sua biblioteca de classes provê uma abstração aos serviços fornecidos pelo SO, como leitura e gravação de arquivos e interfaceamento com dispositivos e periféricos. O *bytecode* é a saída produzida pelo compilador Java, tendo como entrada o código fonte. O *bytecode* é um código binário, porém, ao invés de ser destinado ao hardware, ele é destinado a JVM.

3.2 A Máquina Virtual Java (JVM)

A JVM é o principal componente de tecnologia responsável pela independência de plataforma e SO de Java, o pequeno tamanho de códigos compilados e a habilidade de proteger os usuários de códigos maliciosos (Lindholm, Yellin, 1999).

A JVM não tem conhecimentos sobre a linguagem Java, sendo compatível com qualquer linguagem que produza *bytecode* de acordo com sua especificação, como as linguagens Java, Groove e Scala. Ela tem como entrada o *bytecode*, e geralmente o compila sob demanda para a plataforma em que está sendo executada. O *bytecode* é verificado para assegurar que ele satisfaça a especificação da JVM, já que o mesmo pode ser proveniente de fontes não confiáveis, como a Internet.

3.3 Tipos de Dados

A linguagem Java é uma linguagem fortemente tipada, o que significa que toda variável e expressão possuem um tipo conhecido em tempo de compilação. Os tipos da linguagem Java são divididos em duas categorias: tipos primitivos e tipos referenciais (Gosling *et al.*, 2005).

Os tipos primitivos são:

- *byte* - Número inteiro de 8 *bits*;
- *short* - Número inteiro de 16 *bits*;
- *int* - Número inteiro de 32 *bits*;
- *long* - Número inteiro de 64 *bits*;
- *float* - Número ponto flutuante de 32 *bits*;
- *double* - Número ponto flutuante de 64 *bits*;
- *char* – Caractere Unicode em UTF-16;

- *boolean* – Tipo booleano: valores *false* e *true*;
- *void* – Ausência de retorno.

O tipo *void* pode ser usado apenas como tipo de retorno em métodos, indicando que o método não retornará informações. Os tipos primitivos possuem equivalentes tipos referenciais, conforme mostra a tabela 1.

Tabela 1: Tipos primitivos e seus equivalentes tipos referenciais.

Tipo primitivo	Tipo referencial
byte	java.lang.Byte
short	java.lang.Short
int	java.lang.Integer
long	java.lang.Long
float	java.lang.Float
double	java.lang.Double
char	java.lang.Character
boolean	java.lang.Boolean
void	java.lang.Void

Os tipos referenciais são capazes de armazenar uma referência a um objeto compatível ou o valor *null*. Eles são divididos em três tipos: classes, interfaces e arrays (Gosling *et al.*, 2005). Os tipos arrays armazenam referências a vetores de tipos primitivos ou referenciais com um número de dimensões pré-determinado.

3.4 Classes e Interfaces

Classes e interfaces são criadas através de uma declaração e criam um novo tipo referencial. Cada classe é implementada como uma extensão ou subclasse de uma única classe existente, podendo também implementar uma ou mais interfaces (Lindholm, Yellin, 1999). Uma classe é um modelo para a instanciação de objetos de seu tipo. Já uma interface é um contrato que

declara suas possíveis operações, porém não as implementa. Uma interface pode conter declarações de constantes e estender uma ou mais interfaces.

Através de classes e interfaces Java alcança um de seus principais valores: a orientação a objetos. Um objeto é uma instância de uma classe e possui estado e comportamentos. O estado é mantido pelos *fields* (variáveis de instância) declarados em sua classe e subclasses. Seus comportamentos são definidos pelos métodos de sua classe e subclasses. As interfaces implementadas pela classe e subclasses de um objeto definem os contratos que serão respeitados.

3.5 Métodos

Um método declara um trecho de código executável que pode ser chamado, passando um número fixo de parâmetros (Gosling *et al.*, 2005). Um método possui um tipo de retorno e uma lista opcional de parâmetros. Os parâmetros podem ser de qualquer tipo Java, com exceção de *void*, que pode ser usado apenas para o tipo de retorno. Além dos parâmetros e tipo de retorno, a declaração de um método pode conter diversos modificadores e também a cláusula *throws*, que indica as exceções checadas que podem ser lançadas pelo método.

No *bytecode* um método é representado por sua assinatura. A assinatura de um método é composta por seu nome e o tipo de cada um de seus parâmetros. Java permite a declaração de vários métodos com o mesmo nome em uma classe desde que tenham diferentes assinaturas. Neste caso, diz-se que o método foi sobrecarregado.

Um método pode ser um método estático ou um método de instância. Um método estático é declarado com o modificador *static* e não tem acesso ao estado (variáveis) da instância do objeto, e sim, apenas ao estado da classe. Um método de instância tem acesso ao estado da instância do objeto bem como ao estado da classe.

3.6 Exceções

Quando um programa viola as semânticas da linguagem Java, por exemplo, tentando acessar um elemento fora dos limites de um *array*, a JVM avisa sobre o erro ao programa através de uma exceção. Programas também podem lançar exceções explicitamente, usando o comando *throw* (Gosling *et al.*, 2005). O lançamento de uma exceção quebra o fluxo normal de execução do programa levando-o a um bloco *catch* que seja capaz de tratá-la.

Uma exceção é definida estendendo-se a classe *java.lang.Throwable*. A classe *Throwable* possui informações como a mensagem de erro, o *stack trace* de execução do momento de criação da instância e uma *causa*, que pode ser usada para criar um encadeamento de exceções. Java disponibiliza algumas classes estendidas de *Throwable*, como *java.lang.Exception*, *java.lang.Error* e *java.lang.RuntimeException*, que podem ser usadas diretamente pelo programador. As classes *Exception* e *Error* estendem *Throwable* e a classe *RuntimeException* estende *Exception*.

As exceções são divididas em duas categorias: não-checadas (*unchecked)* e checadas (*checked)*. As exceções não-checadas são *RuntimeException*, *Error* e todas que as estendam. Uma exceção não-checada pode ser lançada sem a necessidade de que o método a trate (com *catch*) ou que declare seu possível lançamento com a cláusula *throws*. Todas as outras exceções são checadas. Uma exceção checada precisa ser tratada no método que a lance ou o método precisa declarar seu possível lançamento através da cláusula *throws*. Quando um método chama outro método que tenha declarado exceções checadas, a possibilidade de lançamento destas exceções é transferida ao método chamador, obrigando-o a tratá-las ou declará-las.

3.7 Java Native Interface (JNI)

Java Native Interface é um importante recurso da plataforma Java. Aplicativos que usam JNI podem incorporar código nativo escrito em linguagens de programação como C e C++. JNI permite que programadores obtenham vantagem da plataforma Java sem ter de abandonar o investimento feito em código legado (Liang, 1999). Através de JNI é possível misturar código nativo e código Java de duas maneiras diferentes: incorporando código nativo a aplicativos Java e incorporando a JVM a aplicativos nativos.

O uso mais convencional de JNI é para a incorporação de código nativo a aplicativos Java. Para fazer este uso, o desenvolvedor deve gerar uma biblioteca dinâmica de código nativo exportando funções segundo as especificações JNI. O código Java deve carregar esta biblioteca através do método *loadLibrary* da classe *java.lang.System*. Os códigos Java e nativo são ligados, pela JVM, através de métodos declarados como *native* e seu nome correspondente exportado em uma biblioteca nativa.

A incorporação da JVM a um aplicativo nativo é feita da seguinte forma. O aplicativo deve carregar a biblioteca nativa da JVM, que contém a função *JNI_CreateJavaVM*. Através desta função, carrega-se a JVM, podendo passar parâmetros como uma execução independente com o utilitário *java*. Para cruzar a fronteira entre código nativo e código Java, uma *thread* nativa precisa ser ligada a JVM. Esta ligação deve ser feita através da função JNI *AttachCurrentThread*. Após a execução do código Java a *thread* pode ser desligada da JVM através da função JNI *DetachCurrentThread*. De acordo com experimentos realizados, diversas chamadas as funções *AttachCurrentThread* e *DetachCurrentThread* torna-se um processo lento. Obtém-se um desempenho melhor chamando *AttachCurrentThread* na primeira oportunidade necessária e adiando a chamada a *DetachCurrentThread* para o momento de encerramento da *thread*.

3.8 JDBC

A API JDBC é uma API Java para acessar virtualmente qualquer tipo de informação em formato tabular. A API consiste de um conjunto de classes e interfaces escritas na linguagem de programação Java que provê uma API padrão para ferramentas e desenvolvedores e possibilita a criação de aplicativos de banco de dados inteiramente na linguagem Java (Fisher, Ellis, Bruce, 2003). JDBC disponibiliza ao desenvolvedor classes para gerenciamento de conexões e transações, criação e execução de comandos SQL e consulta a metadados.

Através de JDBC um programa pode se comunicar com um SGBD SQL, lendo e atualizando informações úteis para seu funcionamento. Cada SGBD possui suas particularidades, como API e tipos de dados próprios. Um *driver* JDBC é o componente responsável por abstrair estas diferenças para a API JDBC.

Para fazer uso de JDBC um programa precisa, em primeiro lugar, carregar um *driver* JDBC. Um *driver* JDBC é carregado quando sua classe é carregada. Por exemplo, para carregar o *driver* do Firebird, é necessário carregar a classe *org.firebirdsql.jdbc.FBDriver* com o método *Class.forName*. Ao ser carregado, um *driver* JDBC registra um prefixo de URL, fazendo com que as URL passadas ao método *DriverManager.getConnection* iniciadas por este prefixo sejam redirecionadas ao *driver*. Este método retornará um objeto *Connection*, fornecido pelo *driver*, e a partir deste objeto o aplicativo interage com o *driver* e o SGBD.

3.9 *ClassLoaders*

Em Java, o carregamento de classes e interfaces é o processo de procurar a forma binária de uma classe ou interface com um nome particular, as vezes criando-a em tempo de execução, mas tipicamente carregando-a de um arquivo previamente criado ao compilar seu código fonte. (Gosling *et al.*, 2005)

A classe *ClassLoader* e suas subclasses são as responsável por este processo de carregamento. Subclasses de *ClassLoader* podem implementar políticas de carregamento diferentes, como buscar classes na Internet ou em um banco de dados, e repassar pedidos para outro *ClassLoader,* conhecido como *ClassLoader* pai.

Após o processo de carregamento é iniciado o processo de ligação da classe ou interface a JVM. Este processo transforma a forma binária da classe ou interface em uma estrutura em memória e a integra a JVM. Três diferentes atividades estão envolvidas no processo de ligação: verificação, preparação e resolução de referências simbólicas (Gosling *et al.*, 2005). A verificação faz a validação das declarações e códigos presentes, de acordo com a especificação da JVM. A preparação envolve a alocação de espaço para constantes e variáveis estáticas bem como a inicialização com seus valores *defaults*. A resolução de referências é o processo de carregar, se necessário, novas classes e interfaces que foram usadas e apontá-las nos locais que contenham referências simbólicas a elas.

Uma importante propriedade dos *ClassLoaders* é o isolamento em relação a outros *ClassLoaders*. Uma classe ou interface com um determinado nome pode ser carregada por mais de um *ClassLoader* e a JVM irá tratá-las como classes ou interfaces totalmente distintas. Seus objetos *Class* serão distintos e suas variáveis estáticas terão valores independentes. A atribuição de uma instância carregada em um *ClassLoader* a um tipo com o mesmo nome resolvido por outro *ClassLoader* lançará a exceção *ClassCastException*.

3.10 Segurança

Discussões referentes ao modelo de segurança de Java frequentemente giram em torno da idéia de *sandbox* (caixa de areia). A idéia por trás deste modelo é a de que quando você permite que um aplicativo rode na sua máquina, você quer providenciar um ambiente onde o

aplicativo rode confinado a certos limites. Você pode decidir deixar o aplicativo ter acesso a certos recursos do sistema, mas em geral, você quer ter certeza que o aplicativo fique confinado a *sandbox*. (Oaks, 2001)

A arquitetura de segurança de Java é baseada em *plugins*, possibilitando a instalação de gerenciadores de segurança que implementem um sistema de segurança específico. O gerenciador de segurança é o componente responsável pela checagem de permissões baseado em um contexto de segurança. Quando um método solicita a checagem de uma permissão, o gerenciador de segurança lança uma exceção *SecurityException* caso o contexto de segurança não possua a permissão.

Um programa Java iniciado através da linha de comando roda sem um gerenciador de segurança. O gerenciador de segurança padrão pode ser instalado iniciando-se a JVM com o parâmetro *-Djava.security.manager*. O gerenciador de segurança padrão trabalha com um arquivo de políticas de segurança. Por padrão, este arquivo é procurado em um diretório da instalação da JVM, porém seu local pode ser redefinido com o parâmetro *-Djava.security.policy*. Através do arquivo de políticas de segurança é possível definir diferentes permissões para códigos provenientes de diferentes locais ou com diferentes assinaturas digitais.

Capítulo 4

4. Desenvolvimento

4.1 Arquitetura

A integração da linguagem Java com o SGBD Firebird foi realizada através do desenvolvimento de dois módulos. O primeiro módulo, chamado de *External Engines*, é uma extensão ao *core* do Firebird, para suportar funções, *stored procedures* e *triggers* externos, independentes da linguagem. O segundo módulo é o *plugin* FB/Java, uma extensão ao projeto Jaybird, que faz a ligação entre o *core* e o código Java do usuário.

A figura 5 apresenta um diagrama mostrando o sistema de execução de rotinas de banco de dados do Firebird com as extensões desenvolvidas neste trabalho. A área destacada no quadro pontilhado mostra o novo suporte a rotinas externas e o *plugin* de integração com Java.

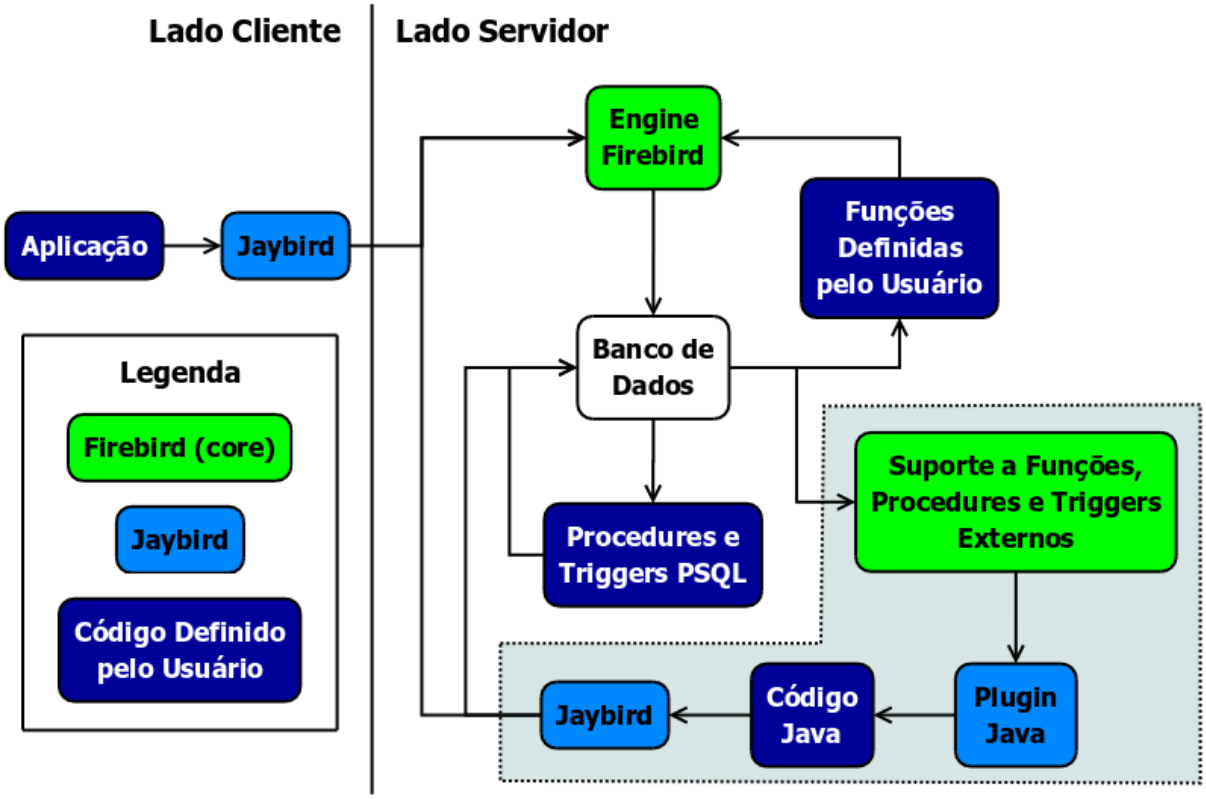

Figura 5: Arquitetura de execução de rotinas no Firebird

4.2 O Módulo *External Engines*

O Firebird possui um módulo para execução de funções externas, porém, este módulo possui alguns inconvenientes:

- As funções trabalham em um contexto externo ao banco de dados. Isto significa que uma função não pode ler ou gravar informações pela mesma conexão do banco de dados que a chamou. Uma função pode abrir outras conexões ao banco, porém, isto pode causar problemas de visibilidade de dados e *deadlocks* com a transação da conexão original.
- As chamadas de funções externas são realizadas com a convenção de chamadas C (*cdecl*). Isto impossibilita a integração de plataformas gerenciadas, como Java, que necessitam de metainformações (como quantidade e tipos dos

parâmetros) sobre as chamadas. Outro importante recurso que se torna inviável é a criação de rotinas genéricas que possam, por exemplo, receber um número variante de parâmetros.

Além destes inconvenientes, o Firebird não possui suporte a *stored procedures* e *triggers* externos. Com base nestes problemas, na necessidade de integração com diferentes linguagens e plataformas e visando uma API de fácil extensão e uso, foi criado o módulo *External Engines*.

O desenvolvimento do módulo *External Engines* foi feito em camadas, e possui três API: a API comum, a API de *plugins* e a API de *external engines*. A API comum é um esforço de desenvolvimento de uma nova API cliente para o Firebird, em linguagem C++. A API comum, por enquanto, possui apenas os elementos úteis a API *external engines*. A API de *plugins* visa unificar o sistema de extensões ao Firebird. Hoje o Firebird possui algumas API não integradas para diferentes tipos de extensões. A API de *external engines* é a responsável pela integração de funções, *stored procedures* e *triggers* externos ao Firebird.

Estas novas API foram desenvolvidas em linguagem C++, porém nem todos os recursos de C++ foram usados. Códigos compilados com diferentes compiladores C++ geralmente são incompatíveis, devido a falta de padronização da ABI (*Application Binary Interface*) e alguns mecanismos de execução. Todas as classes foram definidas como classes abstratas (isto é, apenas com funções *puramente virtuais*), exceções não podem ultrapassar os limites da API e objetos não podem ser alocados por uma camada e desalocados em outra.

O modelo de classes abstratas é conhecidamente portável entre diferentes compiladores, pois é o núcleo da arquitetura COM (*Component Object Model*) da Microsoft. Uma classe abstrata é representada em uma estrutura simples em memória, chamada *vtable*, que possui o endereço de cada método da classe. Esta estrutura pode facilmente ser representada em outras linguagens, tornando possível o uso direto das classes sem a

necessidade de criação de *bindings*. Na plataforma Windows, os métodos usam a convenção de chamadas *stdcall*, padrão deste sistema operacional, através da definição *FB_CALL*.

O trecho seguinte de código mostra a declaração de uma classe abstratas da API comum.

```
class Attachment : public Disposable
{
public:
    virtual Handle FB_CALL getHandle(Error* error) const = 0;

    virtual const char* FB_CALL getUserName() const = 0;
    virtual const char* FB_CALL getDatabaseName() const = 0;
};
```

O suporte a exceções na linguagem C++ depende de mecanismos de tempo de execução não padronizados, que os fabricantes de compiladores implementam de maneiras distintas. Isto torna inviável o uso de exceções em uma API. Este problema poderia ser solucionado da mesma maneira que é feito em linguagens sem suporte a exceções, como C, retornando valores de sucesso ou erro nas funções, porém esta técnica não foi utilizada. Ao invés disso, foi criada uma classe para representação de erros, e toda função que possa gerar erros recebe uma instância desta classe como parâmetro. O trecho seguinte apresenta a classe *Error*.

```
class Error
{
public:
    virtual bool FB_CALL addCode(int32 code) = 0;
    virtual bool FB_CALL addString(const char* str, uint strLength) = 0;
};
```

Através da classe *Error* uma função pode adicionar códigos e mensagens de erros que podem ser checados após a execução da função. O código que chama as funções da API pode passar uma instância de uma classe concreta de *Error* que lance uma exceção C++ pelo destrutor da classe. Isto é possível, pois, neste caso, a exceção não estará cruzando os limites da API.

Assim como as exceções, o suporte a alocação dinâmica de memória em C++ não é padronizado. Um objeto não pode ser alocado com o operador *new* em um código compilado por um compilador e desalocado com o operador *delete* em um código compilado com outro

compilador. Este problema é solucionado na API com o uso da classe *Disposable*, apresentada a seguir.

```
class Disposable
{
public:
    virtual void FB_CALL dispose(Error* error) = 0;
};
```

A classe *Disposable* é uma classe base para outras classes, e possui o método abstrato *dispose*, que deve ser sobrescrito na mesma camada de código que cria instâncias do tipo concreto com o operador *new*. A implementação do método *dispose*, em geral, deve ser feita da seguinte maneira.

```
void FB_CALL dispose(Error* error)
{
    delete this;
}
```

O método *dispose* encapsula o uso do operador *delete* e possibilita a liberação de memória de uma instância de classe criada por um compilador diferente.

4.3 Disponibilização de Código Java no Banco de Dados

O primeiro passo necessário para disponibilizar rotinas em Java em um banco de dados é a instalação do *plugin* FB/Java no banco. A instalação consiste na criação de alguns objetos de banco de dados necessários ao funcionamento do *plugin*, e pode ser realizada através do utilitário *fbjava-deployer*, da seguinte maneira:

```
fbjava-deployer.sh --database servidor:banco.fdb --user sysdba \
    --password materkey --install-plugin
```

O utilitário *fbjava-deployer* é uma aplicação Java que faz uso da classe *org.firebirdsql.jrt.Deployer*, portanto esta classe pode ser usada diretamente por outras aplicações, sem a necessidade de chamar uma aplicação externa. A classe *Deployer* faz a instalação do *plugin* executando o arquivo de script *install.sql* previamente armazenado

dentro do arquivo JAR do Jaybird. O arquivo *install.sql* é disponibilizado junto ao *plugin*, porém seu uso direto não é recomendado.

Após a instalação do *plugin*, arquivos JAR podem ser adicionados ao banco de dados. As classes e *resources* disponibilizados em um banco de dados ficam isolados dos itens presentes em outros bancos de dados, pois cada banco possui um *classloader* distinto. Este isolamento funciona de maneira similar ao isolamento de diferentes aplicações web em um servidor de aplicações. A disponibilização do arquivo JAR é feita da seguinte maneira com o utilitário *fbjava-deloyer*:

```
fbjava-deployer.sh --database servidor:banco.fdb --user sysdba \
    --password materkey --install-jar /home/user/arquivo.jar arquivo
```

O primeiro parâmetro da opção *--install-jar* identifica o arquivo a ser carregado e o segundo parâmetro dá um nome a este arquivo. Através deste nome, um arquivo disponibilizado no banco pode ser atualizado ou removido, respectivamente, das seguintes maneiras:

```
fbjava-deployer.sh --database servidor:banco.fdb --user sysdba \
    --password materkey --replace-jar /home/user/arquivo.jar arquivo

fbjava-deployer.sh --database servidor:banco.fdb --user sysdba \
    --password materkey --remove-jar arquivo
```

O utilitário *fbjava-deployer* é uma aplicação cliente, portanto os arquivos lidos com as opções --install-jar e --replace-jar são buscados, por padrão, na máquina local. Estas opções, porém, além de nomes de arquivos, permitem o uso de URL, por exemplo, *http://servidor.com/arquivo.jar*.

Além do utilitário *fbjava-deployer*, existe outra maneira de manipular arquivos JAR em um banco de dados, através da *package* SQLJ, baseada no padrão ISO SQL/JRT. Com esta *package* é possível instalar, atualizar e remover arquivos usando *procedures* SQL, da seguinte maneira:

```
EXECUTE PROCEDURE SQLJ.INSTALL_JAR('/home/user/arquivo.jar', 'arquivo');

EXECUTE PROCEDURE SQLJ.REPLACE_JAR('/home/user/arquivo.jar', 'arquivo');

EXECUTE PROCEDURE SQLJ.REMOVE_JAR('arquivo');
```

Os procedimentos da *package* SQLJ também são os mesmos usados no *fbjava-deployer*, presentes na classe *org.firebirdsql.jrt.Deployer*. Neste caso, o código será executado dentro do SGBD Firebird, por isso os arquivos serão buscados na máquina servidora.

4.4 Mapeamento de Rotinas

Métodos Java são acessíveis ao banco de dados através do mapeamento de uma rotina interna a uma rotina externa. O mapeamento é feito através de uma declaração com uma especificação de chamada externa. A especificação de chamada consiste da assinatura do método, no seguinte formato:

```
<especificação de chamada> ::=
    <nome qualificado da classe> . <nome do método> (
        [ <tipo> [ { , <tipo> }... ] ]
    ) [ { return | returns } <tipo> ]
    [ ! <informações> ]

<tipo> ::=
    <tipo primitivo> |
    <nome de classe qualificado>
```

Existem dois tipos de mapeamentos suportados: fixo e genérico. Pelo mapeamento fixo, cada parâmetro da rotina declarada no banco de dados deverá ter um parâmetro correspondente na rotina Java. Pelo mapeamento genérico, a rotina Java deverá ter apenas um parâmetro, o *contexto*, independente do número de parâmetros da rotina declarada no banco de dados. O mapeamento genérico pode ser usado com funções, *stored procedures* e *triggers*. O mapeamento fixo pode ser usado com funções e *stored procedures*.

O mapeamento fixo deve ser usado em rotinas com finalidades específicas. Neste caso, o desenvolvedor cria uma função padrão Java que recebe seus parâmetros e devolve ou não uma resposta.

O mapeamento genérico existe para ser usado nos casos em que o desenvolvedor precise criar uma rotina Java sem saber exatamente como esta rotina será chamada, por exemplo, uma *stored procedure* selecionável que retorne o conteúdo de um arquivo XML ou CSV, de forma que cada atributo ou coluna definido no arquivo seja retornado em um parâmetro de saída da *procedure*. Neste caso, o desenvolvedor da rotina em Java delega a responsabilidade da declaração de uma ou mais rotinas no banco de dados para cada requisito da aplicação. Este tipo de mapeamento permite a criação de bibliotecas e *frameworks* passíveis de serem usados por várias aplicações sem a necessidade de reimplementações das rotinas Java. Os *triggers* podem ser definidos apenas com este tipo de mapeamento, pois não seria conveniente a criação de rotinas Java com parâmetros para cada coluna da tabela, já que as tabelas podem facilmente ter colunas adicionadas ou removidas, invalidando assim os mapeamentos.

A parte de informações, na especificação de chamada, é uma string arbitrária que pode ser passada ao método Java e interpretada para fins específicos de cada método.

O *plugin* FB/Java é capaz de chamar apenas métodos estáticos Java, porém não há nenhuma restrição dos recursos de Java durante a execução do método chamado.

4.5 Tipos de Dados

O *plugin* FB/Java trabalha com tipos primitivos e referenciais Java. A tabela 2 apresenta os mapeamentos de tipos suportados. Na tabela, a palavra *"todos"* indica que, por exemplo, um parâmetro Firebird do tipo *VARCHAR* pode ser mapeado a um método Java com parâmetro do tipo *int*, porém, será lançada uma exceção de conversão caso o usuário chame a função passando uma sequência que não seja compatível com o tipo *int*.

Tabela 2: Compatibilidade entre tipos de dados do Java e do Firebird.

Tipo Java	Tipo Firebird Compatível	Observação
byte[]	BLOB, CHAR, VARCHAR	
short	todos	1
int	todos	1
long	todos	1
float	todos	1
double	todos	1
java.lang.Short	todos	
java.lang.Integer	todos	
java.lang.Long	todos	
java.lang.Float	todos	
java.lang.Double	todos	
java.lang.Object	todos	2
java.lang.String	todos	
java.math.BigDecimal	todos	
java.sql.Blob	BLOB	
java.sql.Date	todos	
java.sql.Time	todos	
java.sql.Timestamp	todos	
java.util.Date	todos	
org.firebirdsql.jrt.ExternalResultSet	-	3
org.firebirdsql.jrt.FunctionContext	-	4
org.firebirdsql.jrt.ProcedureContext	-	4
org.firebirdsql.jrt.TriggerContext	-	4

Observações referentes a tabela 2:

1. NULL é convertido para 0 (zero) quando passado a um tipo primitivo;
2. Parâmetros e colunas são convertidos usando o tipo Java padrão;
3. Pode ser usado apenas como tipo de retorno de *procedures*. Alternativamente, o nome qualificado de uma classe que implemente *ExternalResultSet* pode ser usado;
4. Cria um mapeamento genérico. Pode ser usado apenas em métodos com um único parâmetro.

Cada tipo de dado do Firebird possui um tipo Java padrão correspondente. O *plugin* cria ou espera uma instância do tipo padrão em parâmetros do tipo *java.lang.Object* e valores presentes nas classes de contexto. A tabela 3 apresenta o tipo Java padrão para cada tipo Firebird.

Tabela 3: Correspondência entre tipos Firebird e tipos Java padrão.

Tipo Firebird	Tipo Java
SMALLINT	java.lang.Short
INTEGER	java.lang.Integer
BIGINT	java.lang.Long
FLOAT	java.lang.Double
DOUBLE PRECISION	java.lang.Double
CHAR	java.lang.String
VARCHAR	java.lang.String
BLOB	java.sql.Blob
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

As classes *FunctionContext*, *ProcedureContext* e *TriggerContext* são subclasses da classe *org.firebirdsql.jrt.Context* e usam a classe *org.firebirdsql.jrt.Values* para a representação de parâmetros e campos. A figura 6 apresenta a estrutura da classe *Context* e a figura 7 apresenta a estrutura da classe *Values*.

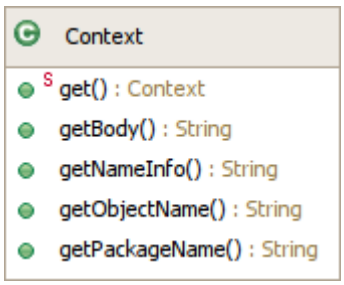

Figura 6: Classe *Context*

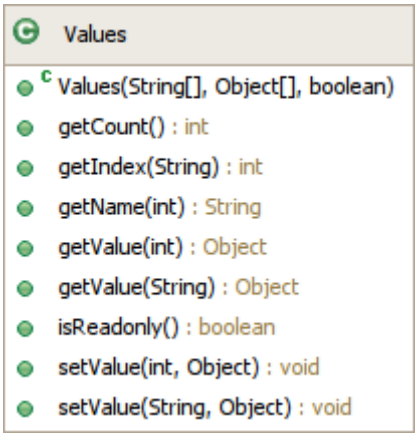

Figura 7: Classe *Values*

4.6 Funções

A criação de funções de banco de dados em Java é feita através do comando *CREATE FUNCTION*. A sintaxe do comando é:

```
CREATE FUNCTION <nome da função> [ ( [ <parâmetros> ] ) ]
    RETURNS <tipo>
    EXTERNAL NAME '<especificação de chamada>'
    ENGINE JAVA
```

Exemplo de criação de função:

```
CREATE FUNCTION get_system_property (
    propriedade VARCHAR(60)
) RETURNS VARCHAR(60)
  EXTERNAL NAME 'java.lang.System.getProperty(java.lang.String)
    return java.lang.String'
  ENGINE JAVA;
```

Neste exemplo, é criado um mapeamento de um método padrão da plataforma Java. A função pode então ser chamada como uma função SQL normal, como no exemplo abaixo:

```
SELECT get_system_property('os.name') FROM RDB$DATABASE;
```

O exemplo seguinte apresenta o código de um método Java a ser usado com o mapeamento genérico:

```
package org.firebirdsql.jrt;

import org.firebirdsql.jrt.FunctionContext;
import org.firebirdsql.jrt.Values;

public class FuncTest
{
    public static int sum(FunctionContext context)
    {
        Values values = context.getParameterValues();
        int ret = 0;

        for (int i = values.getCount(); i >= 1; --i)
            ret += (Integer) values.getValue(i);

        return ret;
    }
}
```

Este método faz a soma de todos os parâmetros recebidos e retorna o resultado. A seguir são apresentados dois mapeamentos diferentes, para soma de dois ou quatro valores.

```
CREATE FUNCTION funcSum2 (n1 INTEGER, n2 INTEGER) RETURNS INTEGER
    EXTERNAL NAME 'org.firebirdsql.jrt.FuncTest.sum(
        org.firebirdsql.jrt.FunctionContext) return int'
    ENGINE JAVA;

CREATE FUNCTION funcSum4 (n1 INTEGER, n2 INTEGER, n3 INTEGER, n4 INTEGER)
    RETURNS INTEGER
    EXTERNAL NAME 'org.firebirdsql.jrt.FuncTest.sum(
        org.firebirdsql.jrt.FunctionContext) return int'
    ENGINE JAVA;
```

Estas funções podem então ser chamadas com suas respectivas quantidades de parâmetros:

```
SELECT funcSum2(10, 20), funcSum4(1, 2, 3, 4) FROM RDB$DATABASE;
```

A figura 8 apresenta a estrutura da classe *FunctionContext*, usada em funções com o mapeamento genérico. A classe *FunctionContext* é uma subclasse de *Context*.

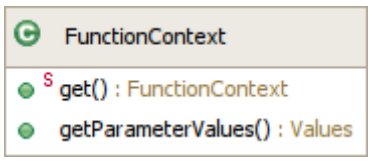

Figura 8: Classe *FunctionContext*

4.7 *Stored Procedures*

A criação de *stored procedures* em Java é feita através do comando *CREATE PROCEDURE*. A sintaxe do comando é:

```
CREATE PROCEDURE <nome da procedure> [ ( [ <parâmetros de entrada> ] ) ]
    [ RETURNS <parâmetros de saída> ]
    EXTERNAL NAME '<especificação de chamada>'
    ENGINE JAVA
```

Exemplo de criação de *procedure*:

```
CREATE PROCEDURE funcionarios RETURNS (
    id INTEGER,
    nome VARCHAR(60)
) EXTERNAL NAME 'org.firebirdsql.example.fbjava.FbJdbc.executeQuery(
      org.firebirdsql.jrt.ProcedureContext)
    return org.firebirdsql.jrt.ExternalResultSet
    !jdbc:postgresql:employee|postgres|postgres'
  ENGINE JAVA;
```

Neste exemplo, é criado um mapeamento de um método Java usando uma especificação de chamada genérica com informações adicionais passadas após o símbolo de exclamação. A *procedure* pode então ser chamada como uma *stored procedure* selecionável, como no exemplo abaixo:

```
SELECT * FROM funcionarios;
```

O trecho seguinte apresenta o código Java do método:

```
package org.firebirdsql.example.fbjava;

import java.sql.Connection;
```

```
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.util.StringTokenizer;
import org.firebirdsql.jrt.ExternalResultSet;
import org.firebirdsql.jrt.ProcedureContext;
import org.firebirdsql.jrt.Values;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class FbJdbc
{
    private static final Logger log = LoggerFactory.getLogger(FbJdbc.class);

    static
    {
        try
        {
            Class.forName("org.postgresql.Driver");
        }
        catch (ClassNotFoundException e)
        {
            log.warn("Cannot load org.postgresql.Driver", e);
        }
    }

    public static ExternalResultSet executeQuery(final ProcedureContext context)
        throws Exception
    {
        return new ExternalResultSet() {
            Values outValues = context.getOutputValues();
            int count = outValues.getCount();
            Connection conn;
            PreparedStatement stmt;
            ResultSet rs;

            {
                StringTokenizer tokenizer = new StringTokenizer(
                    context.getNameInfo(), "|");

                String uri = tokenizer.nextToken();
                String user = tokenizer.hasMoreTokens() ? tokenizer.nextToken() : null;
                String password = tokenizer.hasMoreTokens() ?
                    tokenizer.nextToken() : null;

                conn = DriverManager.getConnection(uri, user, password);
                try
                {
                    StringBuilder sb = new StringBuilder("select ");

                    for (int i = 0; i < count; ++i)
                    {
                        if (i != 0)
                            sb.append(", ");
                        sb.append("x." + outValues.getName(i + 1));
                    }

                    sb.append(" from (" + context.getBody() + ") x");

                    stmt = conn.prepareStatement(sb.toString());
                    rs = stmt.executeQuery();
                }
                catch (Exception e)
                {
                    close();
                    throw e;
                }
            }

            @Override
            public void close() throws Exception
            {
                if (rs != null)
                    rs.close();

                if (stmt != null)
                    stmt.close();
```

```
                if (conn != null)
                    conn.close();
            }

            @Override
            public boolean fetch() throws Exception
            {
                if (!rs.next())
                    return false;

                for (int i = 0; i < count; ++i)
                    outValues.setValue(i + 1, rs.getObject(i + 1));

                return true;
            }
        };
    }
}
```

Este método executa uma consulta por JDBC e retorna os resultados a *procedure* do Firebird. As informações para execução da consulta são obtidas através do método *getNameInfo* da classe *ProcedureContext*.

A figura 9 apresenta a estrutura da interface *ExternalResultSet*, usada para retornar vários registros em *stored procedures* selecionáveis e a figura 10 apresenta a estrutura da classe *ProcedureContext*, usada com o mapeamento genérico. A classe *ProcedureContext* é uma subclasse de *Context*.

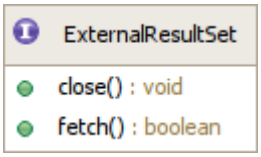

Figura 9: Interface *ExternalResultSet*

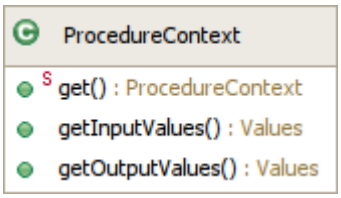

Figura 10: Classe *ProcedureContext*

4.8 *Triggers*

A criação de *triggers* em Java é feita através do comando *CREATE TRIGGER*. A sintaxe do comando é:

```
CREATE TRIGGER <nome do trigger>
    <tipo do trigger>
    EXTERNAL NAME '<especificação de chamada>'
    ENGINE JAVA
```

Exemplo de criação de *trigger*:

```
CREATE TRIGGER funcionarios_log
    AFTER DELETE OR INSERT OR UPDATE
  EXTERNAL NAME 'org.firebirdsql.example.fbjava.FbLogger.info(
    org.firebirdsql.jrt.TriggerContext)'
  ENGINE JAVA;
```

Neste exemplo, é criado um mapeamento para um *trigger* Java que será chamado após exclusões, inserções e atualizações de registros na tabela *funcionarios*. O trecho seguinte apresenta o código Java do método:

```
package org.firebirdsql.example.fbjava;

import org.firebirdsql.jrt.TriggerContext;
import org.firebirdsql.jrt.Values;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class FbLogger
{
    private static final Logger log = LoggerFactory.getLogger(FbLogger.class);
    private static final String NEWLINE = System.getProperty("line.separator");

    public static void info(TriggerContext context)
    {
        String msg = "Table: " + context.getTableName() +
            "; Type: " + getTypeStr(context.getType()) +
            "; Action: " + getActionStr(context.getAction()) +
            valuesToStr(context.getOldValues(), NEWLINE + "OLD:" + NEWLINE) +
            valuesToStr(context.getNewValues(), NEWLINE + "NEW:" + NEWLINE);

        log.info(msg);
    }

    private static String valuesToStr(Values values, String label)
    {
        if (values == null)
            return "";

        StringBuilder sb = new StringBuilder(label);

        for (int i = 1, count = values.getCount(); i <= count; ++i)
            sb.append(values.getName(i) + ": " + values.getValue(i) + NEWLINE);

        return sb.toString();
    }

    private static String getActionStr(int action)
    {
        switch (action)
        {
            case TriggerContext.ACTION_CONNECT:
                return "CONNECT";

            case TriggerContext.ACTION_DISCONNECT:
                return "DISCONNECT";

            case TriggerContext.ACTION_TRANS_COMMIT:
                return "TRANSACTION COMMIT";

            case TriggerContext.ACTION_TRANS_ROLLBACK:
                return "TRANSACTION ROLLBACK";

            case TriggerContext.ACTION_TRANS_START:
                return "TRANSACTION START";
```

```
                case TriggerContext.ACTION_DELETE:
                    return "DELETE";

                case TriggerContext.ACTION_INSERT:
                    return "INSERT";

                case TriggerContext.ACTION_UPDATE:
                    return "UPDATE";

                case TriggerContext.ACTION_DDL:
                    return "DDL";

                default:
                    return null;
            }
        }

        private static String getTypeStr(int type)
        {
            switch (type)
            {
                case TriggerContext.TYPE_AFTER:
                    return "AFTER";

                case TriggerContext.TYPE_BEFORE:
                    return "BEFORE";

                case TriggerContext.TYPE_DATABASE:
                    return "DATABASE";

                default:
                    return null;
            }
        }
    }
```

Este método adiciona a um *log* as alterações efetuadas na tabela. É importante notar que o *trigger* não possui nenhum tratamento específico à tabela *funcionarios*. Por este motivo, o mesmo pode ser usado em quaisquer tabelas, apenas com mapeamentos adicionais.

A figura 11 apresenta a estrutura da classe *TriggerContext*. A classe *TriggerContext* é uma subclasse de *Context*.

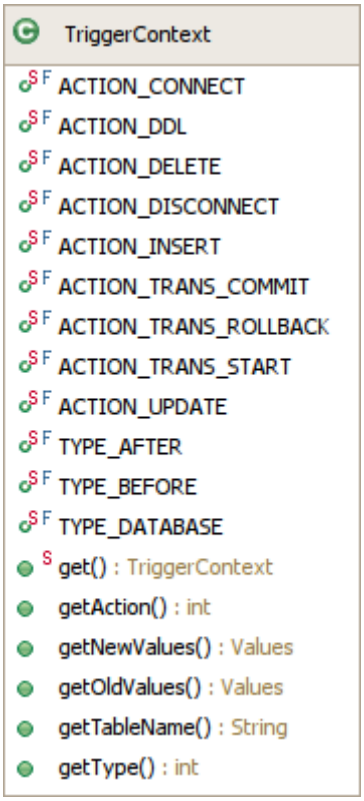

Figura 11: Classe *TriggerContext*

4.9 Acesso ao contexto de transação da aplicação cliente

O *plugin* FB/Java disponibiliza uma maneira das rotinas chamadas acessarem o contexto da transação iniciada pela aplicação cliente. Isto significa que uma rotina em Java pode ler e gravar informações pela mesma transação usada para a sua execução, evitando problemas de visibilidade de dados e *deadlocks* geralmente enfrentados quando UDF escritas em C abrem uma nova conexão para acesso ao mesmo banco de dados.

Para obter este contexto, o desenvolvedor precisa chamar o método *DriverManager.getConnection* passando a URL *jdbc:default:connection*. Através do objeto *Connection* retornado, o desenvolvedor pode executar qualquer método JDBC, com exceção de métodos referentes a início e término de transações, como os métodos *commit*, *rollback* e *setAutoCommit* da classe *Connection*. Esta limitação existe pois as transações são controladas

pela aplicação cliente ou por blocos de código PSQL com transações autônomas ainda em execução.

4.10 Integração entre os Modelos de Segurança Firebird e Java

Da mesma maneira que, a princípio, *applets* rodam com restrições de segurança em um *browser*, o *plugin* FB/Java roda os métodos Java usando o mesmo sistema de segurança desta plataforma: JAAS (*Java Authentication and Authorization Service*).

O administrador de banco de dados Firebird (usuário *SYSDBA*) possui todas as permissões e pode conceder e revogar privilégios adicionais a outros usuários. As permissões são armazenadas e manipuladas em um banco de dados especial para o *plugin*, chamado de *java-security.fdb*. Este banco de dados possui a tabela *PERMISSION*, que possui as seguintes colunas: *USER_NAME, CLASS_NAME, ARG1* e *ARG2*. Na coluna *USER_NAME* é armazenado o nome do usuário de banco de dados que possuirá a permissão, ou a string *PUBLIC*, indicando que todos os usuários a possuirão. Na coluna *CLASS_NAME* é armazenado o nome da classe de permissão Java. Nas colunas *ARG1* e *ARG2* são armazenados os dois possíveis parâmetros da permissão Java.

O *plugin* FB/Java vem previamente configurado com um conjunto de permissões, mostrado na tabela 4.

Tabela 4: Conjunto padrão de permissões do *plugin* FB/Java.

USER_NAME	CLASS_NAME	ARG1	ARG2
SYSDBA	java.security.AllPermission		
PUBLIC	java.util.PropertyPermission	file.separator	read
PUBLIC	java.util.PropertyPermission	java.version	read
PUBLIC	java.util.PropertyPermission	java.vendor	read
PUBLIC	java.util.PropertyPermission	java.vendor.url	read
PUBLIC	java.util.PropertyPermission	line.separator	read
PUBLIC	java.util.PropertyPermission	os.*	read
PUBLIC	java.util.PropertyPermission	path.separator	read

Capítulo 5

5. Conclusão

O objetivo deste trabalho foi integrar o Firebird e a JVM de modo que se tornaria possível a criação de funções, *stored procedures* e *triggers* do Firebird com a linguagem Java. Além disso, estas rotinas deveriam ser fáceis de codificar, deveriam suportar todos os recursos do Firebird e do Java e possibilitariam a criação de bibliotecas reusáveis de código de banco de dados.

Todos estes requerimentos foram alcançados, e este novo recurso mostrou-se bem integrado com os outros recursos e características do Firebird, seu uso mostrou-se simples e seus recursos poderosos.

Para chegar ao resultado final, foram promovidas diversas alterações internas na arquitetura do Firebird. Embora algumas alterações não fossem extremamente necessárias, foi observado que alguns aspectos precisavam ser revistos e melhorados, pois frequentemente causavam problemas de produtividade durante o desenvolvimento e problemas de degradação da qualidade do código existente. Além da integração com Java, foi desenvolvida uma camada para rotinas externas em C++. Esta camada possibilitou um melhor conhecimento das necessidades e a certeza de que do ponto de vista do projeto *core*, as interfaces fossem bem definidas para futuras integrações com mais linguagens e plataformas.

Todas as alterações foram disponibilizadas gratuitamente ao projeto. As alterações do *core* se encontram na árvore principal de desenvolvimento da versão 3.0 do Firebird. As

alterações do Jaybird se encontram em uma árvore paralela que futuramente será integrada a sua versão principal que suportará o Firebird 3.0.

Este trabalho agrega valor ao Firebird e beneficia todos os seus usuários.

Referências Bibliográficas

Bitzer, J., Schröder, P. J. H. The Economics of Open Source Software Development. Holanda: Elsevier B. V. 2006. 281 p.

Borrie, H. The Firebird Book: A Reference for Database Developers. Estados Unidos: Apress. 2004. 1092 p.

Firebird History. http://www.firebirdsql.org/index.php?op=history. Acesso em 30 ago. 2009.

Fisher, M. Ellis, J. Bruce, J. JDBC API Tutorial and Reference, Third Edition. Prentice Hall. 2003. 1280 p.

Freitas, G. A., Paris, R. A. Firebird. (Monografia). Departamento de Sistemas de Informação da Faculdade de Ciências Aplicadas "Sagrado Coração" - UNILINHARES, Linhares, 2007. 39 p.

Gosling, J. *et al.* The Java Language Specification, Third Edition. Addison-Wesley. 2005. 649 p.

Liang, S. The Java Native Interface – Programmer's Guide and Specification. Addison-Wesley. 1999. 303 p.

Lindholm, T., Yellin, F. The Java Virtual Machine Specification, Second Edition. Addison-Wesley. 1999. 473 p.

Oaks, S. Java Security, Second Edition. O'Reilly. 2001. 622 p.

Reese, G. Database Programming with JDBC and Java, Second Edition. O'Reilly. 2000. 328 p.

Silberschatz, A., Korth, H. F., Sudarshan, S. Database System Concepts, Fourth Edition. McGraw-Hill. 2001. 911 p.

Taylor, A. G. SQL for Dummies, 5th Edition. Wiley Publising, Inc. 2003. 408 p.