

r e v i s t a

MUNDO JAVA®

Artigo publicado
na edição 39

MUNDO JAVA



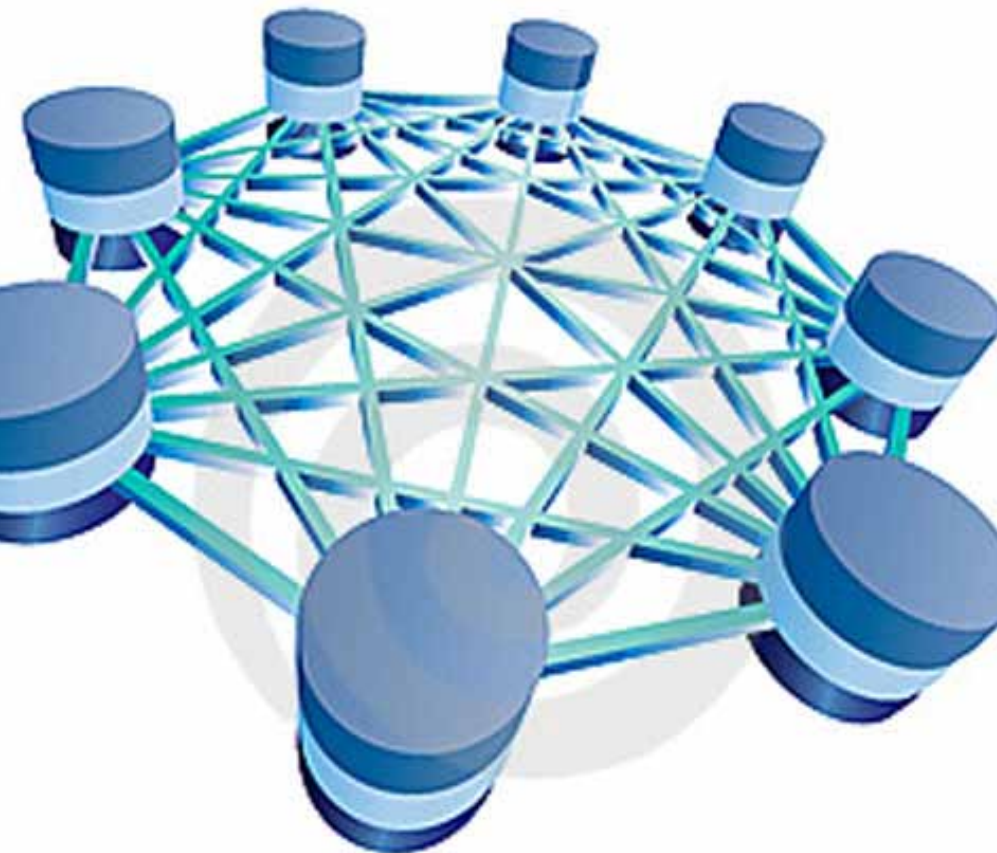
janeiro e fevereiro de 2010



Construindo Stored Procedures com Java no PostgreSQL

Aprenda a escrever stored procedures no PostgreSQL com o poder do Java utilizando sua IDE de programação favorita.

Este artigo dá ênfase a PL/Java, um módulo adicional para o PostgreSQL criado por Thomas Hallgren e mantido atualmente em conjunto com Kris Jurka que possibilita a criação de functions e stored procedures nativas para o banco de dados PostgreSQL em Java. Com PL/Java, é possível escrever qualquer funcionalidade embutida nativamente para o banco de dados PostgreSQL usando sua IDE favorita. Ela possibilita a utilização de recursos para construir aplicações robustas e com funcionalidades escritas em uma linguagem inteligível e “portável” entre vários produtos de gerenciamento de dados que utilizam Java tornando a criação de stored procedures uma tarefa fácil.



Marcelo Costa

(marcelojscosta@gmail.com) é gerente de Infraestrutura e Operações na upLexis Tecnologia, onde atua no desenvolvimento de sistemas para descoberta e tratamento de informação relevante na Web. É bacharel em Ciência da Computação pela Universidade da Amazônia e possui capacitação em Arquitetura de Software, Componentização e SOA pela Unicamp. Trabalha com PostgreSQL desde 2002, quando iniciou sua atuação como gerente e coordenador de equipes nas áreas de projetos de software e banco de dados.



Paulo Crestani

(crestani.jr@gmail.com) é engenheiro de Software na Dextra Sistemas onde atua em projetos de arquitetura e desenvolvimento de sistemas de software com especial ênfase em sistemas Web baseados em tecnologia Java EE. É graduado em Ciência da Computação pela Universidade Estadual de Maringá, mestre em Engenharia Elétrica pela Faculdade de Engenharia Elétrica e de Computação da Unicamp, e especialista em Engenharia de Software pelo Instituto de Computação da Unicamp onde também possui capacitação em Arquitetura de Software, Componentização e SOA. Trabalha com a linguagem Java desde 2003, possuindo as certificações SCJP, SCJD e SCWCD.

O PostgreSQL é considerado um dos sistemas gerenciadores de banco de dados de código livre mais robustos que existe: muitas empresas, apesar da pouca divulgação, utilizam o PostgreSQL como solução para armazenamento de dados em substituição a produtos comerciais aumentando consideravelmente o ROI (do inglês return of investment – retorno sobre o investimento) em armazenamento de dados sem perder funcionalidades ou recursos devido à mudança para um produto de código livre. É muito comum ver na lista de discussão nacional (pgbr-geral) no yahoo groups chamadas de prestadores de serviços para órgãos como o Ministério da Saúde, Caixa Econômica Federal, Dataprev e Correios solicitando profissionais experientes em PostgreSQL para integração com Java.

O PL/Java é uma linguagem procedural compatível com a especificação SQL 2003, o que significa que você pode escrever stored procedures em PL/Java e usar estes mesmos códigos (compilados ou em código-fonte) com qualquer gerenciador de banco de dados que obedeça à especificação SQL 2003 (incluindo Oracle, DB2, Sybase, Firebird, entre outros) alterando apenas pequenas funcionalidades particulares de cada SGBD (sistema gerenciador de banco de dados). Uma das grandes vantagens de escrever suas stored procedures em PL/Java é ter acesso aos dados armazenados no banco de dados mais rapidamente e com isso adicionar vantagem a um dos requisitos não-funcionais mais problemáticos que existem para qualquer software, ou seja, performance, especificamente no acesso aos dados armazenados no banco de dados. Além disso, quando uma stored procedure é escrita em PL/Java, um código java normal é escrito, esse código é transformado em bytecode, ou seja, o código-fonte é compilado, transformado em um arquivo .class e esse arquivo é então transformado no formato de um java archive (um arquivo .jar). Feito isso, esse java archive pode ser carregado no banco de dados.

SQL (Structured Query Language) é uma linguagem de pesquisa para sistemas gerenciadores de banco de dados (SGBD). Muitos produtores criaram variações da linguagem apesar de ter sido desenvolvida pela IBM. Foi necessário, então, padronizar a linguagem pela ANSI (American National Standards Institute — Instituto Nacional Americano de Padronização), em 86 e pela ISO (International Organization for Standardization — Organização Internacional para Padronização) em 87. SQL foi revisto em 92 (SQL-92), em 99 SQL:1999 (SQL3) e 2003 SQL:2003. O SQL:1999 usa expressões regulares, queries recursivas e triggers. Além de tipos não-escalados e características de orientação objeto. O SQL:2003 possui características ao XML, sequências padronizadas e colunas com valores de autogeneralização.

Stored Procedures são coleções de comandos em SQL para execução no banco de dados. Elas existem para tornar nossas atividades mais fáceis quando necessitamos manipular dados. Imagine stored procedures como ferramentas desenhadas para realizar uma tarefa de manipulação de dados bem definida, como, por exemplo, calcular a raiz quadrada de um número ou transformar letras minúsculas em maiúsculas. Stored procedures podem: reduzir o tráfego na rede, melhorar a performance, criar mecanismos de segurança etc.

Como funciona a integração da PL/Java com o PostgreSQL

Uma stored procedure PL/Java, quando executada, cria um processo na Java Virtual Machine (JVM) que interage diretamente com o processo principal do servidor PostgreSQL. Dessa forma, a JVM executa o bytecode da stored procedure carregado no servidor de banco de dados. O bytecode é executado pela JVM do SO como explicado posteriormente possibilitando um tempo de resposta extraordinário. Uma stored procedure em PL/Java representa um método de uma classe Java e, uma vez que são métodos Java, podemos usar estruturas de controle Java, classes Java, tipos de dados Java e funções escritas em Java sem nenhum problema; e o mais importante: sem precisar aprender a sintaxe de uma linguagem nativa de um banco de dados (PL/pgSQL, PL/SQL, PL/PSM etc.). No entanto, é importante lembrar que as stored procedures são construídas obedecendo ao padrão SQL 2003. A PL/Java permite executar de uma forma padronizada a passagem de parâmetros e o retorno de valores. Tipos complexos e configurações são passados usando a classe padrão do JDBC ResultSet. Esse módulo foi projetado com o objetivo de permitir ao banco de dados todo o poder que a linguagem Java propõe de forma que a lógica do negócio possa ser totalmente armazenada no banco de dados, sendo executada de forma mais ágil uma vez que a interação com os dados armazenados no banco de dados é realizada em uma linguagem nativa ao servidor de banco de dados visto que não necessita de um driver como o JDBC funcionando como uma camada a mais de acesso as informações.

Junto com o módulo PL/Java, um driver JDBC é fornecido. Este driver interage diretamente com as rotinas SPI (Server Programming Interface) internas do PostgreSQL sendo fundamental para o reuso de stored procedures. Uma questão importante a observar é que o módulo PL/Java não possui uma java virtual machine própria, ele faz uso da java virtual machine instalada no sistema operacional em uso. PL/Java foi especificamente construído e otimizado para a melhor performance possível e executa apenas o processo que o iniciou em uma chamada do PL/Java, sendo interrompida caso ocorra uma quebra de violação da transação.

SPI (Server Programming Interface — Interface de Programação do Servidor) é uma forma de dar ao usuário a habilidade de executar consultas SQL dentro de funções C definidas pelo próprio usuário. SPI é um conjunto de funcionalidades existentes e que possuem uma interface de acesso ao parser, planejador de consultas e executor do servidor de banco de dados, além de poder gerenciar também alguns aspectos da memória. A maioria das linguagens procedurais utilizadas pelo PostgreSQL executam de várias formas comandos SQL a partir de stored procedures, muitas dessas facilidades são baseadas em rotinas de SPI. Observe que se um comando chamado a partir de uma instrução SPI falhar, o controle de transação não retorna para a procedure que o chamou. Neste caso, a transação ou subtransação na procedure será retornada a condição anterior a existência do problema (rolled back).



Para instalar a infraestrutura necessária para este artigo (PostgreSQL e biblioteca libjvm.so), acesse o site da revista MundoJava e leia o tutorial de instalação do PostgreSQL e configuração do Java no Ubuntu Linux.

Escrevendo Stored Procedures e Triggers em Java

Escrever stored procedures e triggers com o suporte do módulo PL/Java resume-se ao uso da conhecida API JDBC, com algum suporte da API da PL/Java. Em geral, métodos de acesso a banco utilizando a API JDBC resumem-se a uma rotina básica de obtenção de uma conexão, montagem e execução de uma consulta, iterar sobre os resultados da consulta no caso de operações de seleção, e finalmente fechar a conexão obtida inicialmente.

É importante afirmar que não obstante a facilidade oferecida pelo módulo PL/JAVA em termos de utilização da sintaxe Java para a construção de stored procedures e triggers portáveis entre bancos de dados distintos, sua adoção deve ser considerada com cautela no desenvolvimento de sistemas. Conforme mencionado anteriormente, a razão principal para a utilização de código executável dentro de um servidor de banco de dados está relacionada ao ganho de performance oferecido por essa alternativa. Por outro lado, a codificação e a manutenção de stored procedures e triggers é uma opção sempre mais custosa, em termos de manutenção de código, que a codificação de consultas diretamente no código da aplicação, como, por exemplo, com a utilização de JDBC, e especialmente quando se considera frameworks de infraestrutura de alto nível como Java Persistence API ou Hibernate.

Em geral, a utilização de recursos de manipulação de dados através de stored procedures não é apenas útil, mas mandatória quando um problema de desempenho é detectado. Em aplicações de médio e grande porte, com forte requisito de disponibilidade de dados e acesso concorrente a grandes volumes de dados, este tipo de situação ocorrerá fatalmente, mas isto não justifica a adoção de stored procedures como mecanismo padrão de acesso a dados. Ao invés disto, é fundamental a utilização de ferramentas de identificação de perfil de desempenho da aplicação (profilers), capazes de identificar com precisão os pontos do sistema que constituem gargalos de acesso a dados e, portanto, onde a utilização das técnicas apresentadas a seguir tornam-se necessárias. Assim, o uso de PL/JAVA como um padrão não é recomendado, mas sua utilização como técnica de melhoria de desempenho em pontos de gargalo da aplicação é uma alternativa bastante atraente e quase certamente necessária, uma vez que conforme descrito no artigo o objetivo é escrever stored procedures em java.

Dito isso, os exemplos apresentados a seguir demonstram a utilização da API JDBC e da API fornecida pelo módulo PL/JAVA para a construção de stored procedures e triggers utilizando linguagem Java. Estes exemplos, embora simples, por questões didáticas, apresentam conceitos que podem ser fácil e imediatamente estendidos a consultas mais complexas, que justifiquem a utilização de stored procedures em detrimento à construção de consultas diretamente no código da aplicação.

Os exemplos apresentados nesta seção desenvolvem diversas operações de seleção e atualização de informações de um banco de dados de informações bancárias utilizando linguagem Java. O processo de implementação e testes de cada exemplo é descrito desde a codificação até a implantação e execução das operações como stored procedures em um banco de dados PostgreSQL.

O modelo de banco de dados utilizado pelos códigos de exemplo é uma base de informações bancárias simples contendo quatro tabelas conforme mostra-

do na figura 1 e seu script de instalação (banuario.sql) pode ser baixado no site da revista MundoJava. O projeto Java usado para a manipulação de dados nestas tabelas consiste em quatro classes: ProcedimentosBancarios, DadosCliente, DadosDebito, IteradorDeClientes, além da interface DatabaseConstants. Cada uma destas classes será descrita juntamente com o exemplo que a utiliza. Por hora, importa saber que todos os métodos exportados na forma de stored procedures ou triggers são definidos na classe ProcedimentosBancarios. Estes métodos devem ser declarados como públicos e estáticos.

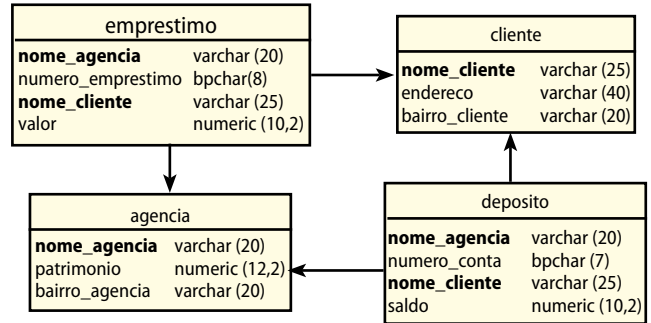


Figura 1. Modelo de banco de dados exemplo.

As classes, uma vez compiladas, podem ser empacotadas em um arquivo jar antes de serem carregadas no servidor de banco de dados. É possível carregar classes individuais no servidor de banco de dados, porém, empacotar classes em um arquivo jar é altamente aconselhável. Ao final do artigo, será realizada uma explicação detalhada de como fazer o deployment de uma forma simples e fácil.

Para criar o arquivo funcionalidades-bancarias.jar, basta ir ao diretório em que os arquivos compilados (.class) se encontram e executar o comando da Listagem 1.

Listagem 1. Criação do arquivo funcionalidades-bancarias.jar.

```
# jar cf funcionalidades_bancarias.jar ProcedimentosBancarios.class
IteradorDeClientes.class DadosCliente.class DadosDebito.class
DatabaseConstants.class
```

IDEs como o Eclipse dão suporte à criação de arquivos jar com suporte gráfico (vide <http://www.fsl.cs.sunysb.edu/~dquigley/cse219/index.php?it=eclipse&tt=jar&pf=y>), entretanto, em projetos profissionais de software, as gerações de empacotamentos jar, war e ear são normalmente automatizadas por ferramentas de gestão de configuração de projetos de software como ant e maven e a descrição destes recursos foge do escopo deste artigo.

Listagem 2. Comando para carga do arquivo jar no banco de dados.

```
banco=# SELECT sqlj.install_jar ( 'file:///home/paulo/projetos/pljava/
workspace/PLJava/bin/funcionalidades_bancarias.jar', 'Banco', true );
```

Todos os comandos neste artigo são executados no banco de dados como usuário postgres conforme descrito no tutorial de instalação da infraestrutura.

O comando sqlj.install_jar (Listagem 2) instala o arquivo .jar dentro do repositório do servidor, mais especificamente na tabela sqljar_entry,

e associa a ele o nome simbólico Banco. O primeiro parâmetro indica a URL que representa o arquivo .jar a ser carregado, o segundo representa o nome simbólico associado ao jar e o terceiro parâmetro é um valor lógico que indica ao servidor se o descritor contido no arquivo jar deve ser processado ou não. Mais adiante será apresentado o papel do descritor MANIFEST.MF para facilitar o processo de instalação do arquivo jar e suas funções no banco de dados. Eventualmente é necessário alterar uma versão de jar instalada no servidor por outra mais recente. Para isto, basta executar o comando `replace_jar` conforme apresentado na Listagem 3. Esse comando (Listagem 3) possui a mesma sintaxe do comando `install_jar`.

Listagem 3. Comando para alteração de arquivos jars instalados.

```
banco=# SELECT sqlj.replace_jar( 'file:///home/paulo/projetos/pljava/
workspace/PLJava/bin/funcionalidades_bancarias.jar', 'Banco', true );
```

Além de instalar o arquivo jar, é preciso que suas classes estejam no classpath do esquema de banco de dados ao qual se quer associar as stored procedures. Sem este passo, a JVM não conseguirá localizar as classes e métodos do arquivo jar carregado anteriormente. Para associar as classes ao classpath do esquema, é preciso executar o comando da Listagem 4 no cliente do servidor de banco de dados:

Listagem 4. Comando para definição do classpath.

```
banco=# SELECT sqlj.set_classpath( 'public', 'Banco' );
```

O comando da Listagem 4 coloca o jar associado ao banco de dados Banco no classpath do esquema público (public). Note que o comando `set_classpath` usa os nomes simbólicos dos arquivos jar que já devem ter sido importados, e não seus nomes físicos. É possível acrescentar múltiplos arquivos ao classpath de um esquema passando ao comando uma lista de nomes simbólicos separados por ":" conforme mostra a Listagem 5. `banco=# SELECT sqlj.set_classpath('public', 'Banco:Empresa');`

Listagem 5. Comando para definição do classpath a ser utilizado.

```
banco=# SELECT sqlj.set_classpath( 'public', 'Banco:Empresa' );
```

Uma vez executados estes passos, é possível começar a criar e executar no PostgreSQL functions e triggers definidas no arquivo jar.

Interagindo com banco de dados

Uma maneira muito simples de interagir com o banco de dados é através da seleção de um valor escalar em uma tabela. O método `saldo()` apresentado na Listagem 6 faz exatamente isto utilizando a API JDBC. Dado o nome de um cliente e o nome de uma agência, o método obtém uma conexão JDBC (linha 6), define e executa a query de seleção (linhas 8 a 13) e obtém e retorna o saldo do cliente na agência informada (linhas 14 a 16). Note que a conexão com o banco é fechada em um bloco `finally` para evitar que permaneça aberta caso uma exceção seja disparada.

Para montar a consulta foi utilizada uma instância de `PreparedStatement`, que permite a predefinição da query a ser executada e a definição

Listagem 6. Método `saldo()`. Um exemplo de interação com o servidor de banco de dados PostgreSQL.

```
1 public static double saldo(String nomeCliente, String nomeAgencia)
2     throws SQLException {
3     Connection connection = null;
4     PreparedStatement statement = null;
5     try {
6         connection = DriverManager
7             .getConnection(DatabaseConstants.defaultURL);
8         String query = new StringBuilder("SELECT * FROM
9             deposito ")
10            .append("WHERE nome_cliente = ? and nome_
11            agencia = ?");
12         statement = connection.prepareStatement(query.toString());
13         statement.setString(1, nomeCliente);
14         statement.setString(2, nomeAgencia);
15         ResultSet resultSet = statement.executeQuery();
16         resultSet.next();
17         double saldo = resultSet.getFloat("saldo");
18         return (saldo);
19     } finally {
20         statement.close();
21         connection.close();
22     }
```

de parâmetros, o que é feito nas linhas 11 e 12.

Para criar no PostgreSQL a função que mapeia o método `saldo()`, basta executar o comando da Listagem 7. O comando `CREATE FUNCTION` cria uma função `saldo` no banco de dados que aceita como parâmetros dois atributos do tipo TEXT e retorna um valor do tipo DOUBLE PRECISION. Todos estes são tipos nativos do PostgreSQL. Em seguida, `CREATE FUNCTION` associa a função e os tipos PostgreSQL com o equivalente em Java. Para isto, define-se a assinatura do método, incluindo o nome da classe a que pertence e os tipos de seus parâmetros. Neste caso, `java.lang.String` que mapeia para o tipo TEXT do PostgreSQL. Por fim, a linguagem da função é definida como sendo Java.

Listagem 7. Comando para criação da função que mapeia o método `saldo()`.

```
banco=# CREATE FUNCTION saldo( TEXT, TEXT ) RETURNS DOUBLE
PRECISION

banco=# AS 'ProcedimentosBancarios.saldo( java.lang.String, java.lang.
String )'

banco=# LANGUAGE java;
```

Observe que, caso a assinatura da função seja modificada em versões posteriores do jar, a função original precisará ser eliminada para que a nova função possa ser carregada novamente com o comando `CREATE FUNCTION` do PostgreSQL. No PostgreSQL, para eliminar uma função previamente criada, executa-se o comando `DROP FUNCTION` como mostrado na Listagem 8. Lembre-se de declarar também os tipos retornados.

Listagem 8. Comando para eliminar uma função previamente criada.

```
banco=# DROP FUNCTION saldo( TEXT, TEXT );
```

Uma vez criada a função no banco de dados, é possível executá-la e observar o resultado conforme a listagem 9.

Listagem 9. Resultado da função criada que utiliza o método saldo().

```
banco=# select saldo('Jose Antonio', 'Paulista');

saldo
-----
 500
(1 row)
```

Retornando múltiplos valores

A PL/Java define por convenção três formas as quais pode-se retornar múltiplos valores de uma consulta. Se o objetivo for retornar uma lista de valores escalares, a PL/Java define que a função Java que realiza a consulta deve retornar uma instância de `java.util.Iterator`. Entretanto, se o objetivo for retornar uma lista de valores complexos, como registros, então a PL/Java fornece duas alternativas de retorno: as interfaces `ResultSetHandle` e `ResultSetProvider`. Nesta seção, será ilustrado um exemplo de retorno de valores escalares. Nas seções seguintes serão exploradas as interfaces `ResultSetHandle` e `ResultSetProvider`.

Na Listagem 10, em termos gerais, o método `clientesComEmprestimoEmAgencia()` lista todos os clientes com empréstimos em uma determinada agência e sua implementação é semelhante ao método `saldo()`. Uma conexão é estabelecida (linhas 6 e 7), e a consulta é montada e executada (linhas 9 a 13). Entretanto, dado o nome de uma agência, o método lista todos os seus clientes o que implica retornar múltiplos valores ao SGBD. Para isso, a PL/Java convencionou que o método Java que realiza a consulta retorne uma instância da classe `java.util.Iterator` contendo os valores da consulta. Assim, um `ArrayList` (linha 14) é criado e preenchido com valores obtidos do `ResultSet` retornado pela consulta (linhas 15 a 17). Por fim, o `Iterator` é obtido do `ArrayList` (linha 18).

O próximo passo é criar uma função no PostgreSQL e executá-la. Como pode ser observado na Listagem 11, o tipo de retorno definido para a função agora é `SETOF TEXT`, que corresponde ao `java.util.Iterator<String>` retornado pelo método Java. O resultado da execução da função para a agência Paulista pode ser observado por meio do `select` executado na Listagem 11 em que, nesse exemplo, dois clientes são identificados.

Um ponto fraco da abordagem ilustrada antes é que o iterador tem de ser completamente preenchido na busca antes de ser retornado. Para consultas que retornem grandes volumes de dados, esta é uma solução que degrada sobremaneira o desempenho do sistema. Uma alternativa é construir o iterador sob demanda. O método `clientesDaAgencia()` apresentado na Listagem 12 não retorna um `Iterator` diretamente, mas uma instância de `IteradorDeClientes`, delegando a esta classe a realização da consulta. Este método retorna todos os clientes de uma agência, por definição, aqueles que possuam ou um empréstimo ou um depósito na agência.

Retorna todos os clientes de uma agência, por definição, aqueles que possuam ou um empréstimo ou um depósito na agência.

O código de `IteradorDeClientes` pode ser visto na Listagem 13. Esta classe implementa a interface `java.util.Iterator<String>`, de forma que quando o método `clientesDaAgencia()` da classe `ProcedimentosBancarios` retorna um `IteradorDeClientes`, está retornando um `Iterator`.

Listagem 10. Retornando valores escalares.

```
1 public static Iterator<String> clientesComEmprestimoEmAgencia(
2     String nomeAgencia) throws SQLException {
3     Connection connection = null;
4     PreparedStatement statement = null;
5     try {
6         connection = DriverManager
7             .getConnection(DatabaseConstants.defaultURL);
8         StringBuilder query = new StringBuilder(
9             "SELECT nome_cliente FROM emprestimo ")
10            .append(" where nome_agencia LIKE ? ");
11         statement = connection.prepareStatement(query.toString());
12         statement.setString(1, nomeAgencia);
13         ResultSet resultSet = statement.executeQuery();
14         List<String> result = new ArrayList<String>();
15         while (resultSet.next()) {
16             result.add(resultSet.getString(" nome_cliente"));
17         }
18         return (result.iterator());
19     } finally {
20         statement.close();
21         connection.close();
22     }
23 }
```

Listagem 11. Criação da função que faz a chamada ao método criado na Listagem 10.

```
banco=# CREATE FUNCTION clientesComEmprestimoEmAgencia( TEXT )
RETURNS SETOF TEXT

AS 'ProcedimentosBancarios.clientesComEmprestimoEmAgencia( java.lang.
String )' LANGUAGE java;

banco=# select clientesComEmprestimoEmAgencia('Paulista');

clientescomemprestimoemagencia
-----
Joao Carlos
Jose Antonio
(2 rows)
```

Listagem 12 – método clientesDaAgencia().

```
1 public static Iterator<String> clientesDaAgencia(String
nomeAgencia) {
2     return (new IteradorDeClientes(nomeAgencia));
3 }
```

O construtor da classe `IteradorDeClientes` armazena o nome da agência a ser consultada em uma variável de instância. Como implementa a interface `Iterator`, `IteradorDeClientes` deve implementar os métodos `hasNext()`, `next()` e `remove()`. Em particular, o método `remove()` não é chamado pelo banco de dados e portanto sua implementação simplesmente dispara uma `UnsupportedOperationException`. Quando o banco de dados acessa o método `next()`, este busca pelo próximo elemento da consulta. Para isso, procura obter o `ResultSet` que representa o resultado da busca através de uma chamada ao método utilitário privado `getResultSet()`.

Listagem 13. Implementação da interface java.util.Iterator<String>.

```

1 public class IteradorDeClientes implements Iterator<String> {
2     private String nomeAgencia = "";
3     private ResultSet resultSet = null;
4     private Connection connection;
5     private PreparedStatement statement;
6
7     public IteradorDeClientes(String nomeAgencia) {
8         this.nomeAgencia = '%' + nomeAgencia.replace(" ", "%")
9         + "%";
10    }
11    private void tearDown() {
12        try {
13            statement.close();
14            connection.close();
15        } catch (SQLException e) {
16            new RuntimeException(e);
17        }
18    }
19    private ResultSet getResultSet() throws SQLException {
20        if (resultSet == null) {
21            connection = DriverManager
22                .getConnection(DatabaseConstants.defaultURL);
23            StringBuilder query = new StringBuilder(
24                "SELECT DISTINCT c.nome_cliente FROM cliente AS c,
25                deposito AS
26                d, emprestimo AS e " .append(" WHERE (c.nome_cliente =
27                d.nome_cliente) AND (d.nome_agencia LIKE ?) ")
28                .append(" OR
29                (c.nome_cliente = e.nome_cliente) and (d.nome_agencia
30                LIKE ?) ");
31            statement = connection.prepareStatement(query.toString());
32            statement.setString(1, nomeAgencia);
33            statement.setString(2, nomeAgencia);
34            resultSet = statement.executeQuery();
35        }
36        return resultSet;
37    }
38    @Override
39    public boolean hasNext() {
40        try {
41            return !(getResultSet().isLast());
42        } catch (Exception e) {
43            return (false);
44        }
45    }
46    @Override
47    public String next() {
48        try {
49            getResultSet().next();
50            return (getResultSet().getString("nome_cliente"));
51        } catch (Exception e) {
52            try {
53                getResultSet().close();
54            } catch (SQLException e1) {
55                tearDown();
56                throw new RuntimeException(e1);
57            }
58            throw new NoSuchElementException("Sem mais clientes");
59        }
60    }
61    @Override
62    public void remove() {
63        throw new UnsupportedOperationException();
64    }
65 }

```

Quando este método é acessado pela primeira vez, `ResultSet` é `null`, de forma que `getResultSet()` constrói e executa a consulta aos clientes da agência utilizando o nome da agência armazenado pelo construtor. O resultado é armazenado em uma instância singleton de `ResultSet` que é então retornada ao método `next()`. O método `next()` acessa e retorna o próximo elemento do `ResultSet` singleton. O método `hasNext()` retorna `false` quando o último elemento do `ResultSet` tiver sido lido e retornado. O próximo passo é criar a função no PostgreSQL e invocá-la. O resultado desse processo é apresentado na Listagem 14.

Retornando múltiplos registros

A PL/Java suporta duas alternativas para retorno de tipos complexos como registros em uma busca: através do retorno de uma instância de `ResultSetHandle` ou através do retorno de uma instância de `ResultSetProvider`. O exemplo da Listagem 16 explora a alternativa com `ResultSetHandle` e o exemplo da Listagem 18 aborda a alternativa com `ResultSetProvider`. Assim como no exemplo da Listagem 13, embora a classe `ProcedimentosBancarios` exporte o método `dadosCliente()` para o banco de dados, não é ela quem realiza a consulta. Ao invés disto, o método `dadosCliente()` delega o trabalho para uma instância da classe `DadosCliente`, que implementa a interface `ResultSetHandle`. O propósito de `dadosCliente()` é retornar o registro completo do cliente dado seu nome conforme mostra a Listagem 15.

Listagem 14. Criação da função de invocação `clientesDaAgencia`.

```

banco=# CREATE FUNCTION clientesDaAgencia(TEXT) RETURNS SETOF
TEXT AS 'ProcedimentosBancarios.clientesDaAgencia(java.lang.String)'
LANGUAGE java;

```

```

banco=# SELECT clientesDaAgencia('Paulista');
clientesdaagencia
-----
Joao Carlos
Jose Antonio
Renan Souza
(3 rows)

```

Listagem 15. Método `DadosCliente`.

```

1 public static ResultSetHandle dadosCliente(String nomeCliente) {
2     return new DadosCliente(nomeCliente);
3 }

```

O código da classe `DadosCliente` é apresentado na Listagem 16. Assim como na classe `IteradorDeClientes` do exemplo anterior, o construtor de `DadosCliente` armazena o parâmetro da consulta. Neste caso, o nome de um cliente em uma variável de instância para uso posterior. A interface `ResultSetHandle` exige a implementação de dois métodos: `close()` e `getResultSet()`. O método `close()` é chamado pelo banco de dados quando todos os registros do `ResultSet` retornado já tiverem sido lidos, servindo para fechar a conexão com o banco. Já o método `getResultSet()` realiza a consulta com base no parâmetro armazenado pelo construtor e retorna o resultado da busca na forma de um `ResultSet`.

Finalmente é necessário criar a função no banco de dados e invocá-la. Neste caso, o processo de criação da função deve ser antecedido pela criação do tipo de dados que será retornado pela função. Como agora o tipo de retorno é um registro, este precisa ser criado no servidor de banco de dados, o que é feito por meio do comando `CREATE TYPE` na Listagem 17.

Listagem 16. Classe DadosCliente().

```

1 import java.sql.Connection;
2 import java.sql.DriverManager;
3 import java.sql.PreparedStatement;
4 import java.sql.ResultSet;
5 import java.sql.SQLException;
6
7 import org.postgresql.pljava.ResultSetHandle;
8
9 public class DadosCliente implements ResultSetHandle {
10
11     private String nomeCliente;
12     private Connection connection;
13     private PreparedStatement statement;
14
15     public DadosCliente(String nomeCliente) {
16
17         this.nomeCliente = '%' + nomeCliente.replace(" ", "%") + "%";
18     }
19
20
21     @Override
22     public void close() throws SQLException {
23         statement.close();
24         connection.close();
25     }
26
27     @Override
28     public ResultSet getResultSet() throws SQLException {
29         connection =
30             DriverManager.getConnection(DatabaseConstants.
31             defaultURL);
32         StringBuilder query =
33             new StringBuilder("SELECT * FROM cliente AS c ")
34             .append(" WHERE c.nome_cliente LIKE ? ");
35         statement = connection.prepareStatement(query.toString());
36         statement.setString(1, nomeCliente);
37         return statement.executeQuery();
38     }
39 }

```

Retornar registros com suporte da interface `ResultSetHandle` é muito útil quando os valores registros retornados são lidos diretamente de uma consulta a uma tabela do banco de dados, mas há situações em que é desejável construir o próprio `ResultSet`, como, por exemplo, em situações em que o `ResultSet` possua campos com valores calculados. Neste caso, a interface `ResultSetProvider` é a alternativa a ser utilizada. O exemplo na Listagem 18 mostra como retornar um `ResultSet` personalizado com o suporte de `ResultSetProvider`. Assim como nos exemplos anteriores, o método exportado para o banco de dados, neste caso, `debitosClientes()`, é definido na classe `ProcedimentosBancarios`, mas esta delega a consulta para a classe `DadosDebito`, que implementa a interface `ResultSetProvider`. O método `debitosClientes()` identifica todos os clientes com depósitos e empréstimos no banco e calcula seu débito potencial com o banco, que é representado pela diferença entre seu saldo e o valor de seu empréstimo. Como o banco pode possuir muitos clientes, a consulta possui um parâmetro que regula a quantidade máxima de registros a serem retornados (`maxRecords`). Quando o valor de `maxRecords` é negativo, todos os registros encontrados são retornados.

O código da classe `DadosDebito` é apresentado na Listagem 19. Ela mantém um `ResultSet` como variável de instância para armazenar os resultados da busca. Quando o construtor é invocado, ele recebe o parâmetro `maxRecords` e realiza a consulta no banco. O `ResultSet` com

os resultados da consulta é armazenado em uma variável de instância. A interface `ResultSetProvider` exige a implementação de dois métodos: `assignRowValues()` e `close()`. O método `close()`, assim como no caso da implementação de `ResultSetHandle`, é chamado quando todos os registros do `ResultSet` gerado pela consulta são lidos. O banco de dados chama `close()` para fechar a conexão aberta pelo construtor da classe `DadosDebito`. O método `assignRowValues()` recebe um `ResultSet` externo e o número da linha daquele `ResultSet` que será atualizada. Nesse caso, o método transmite para o `ResultSet` recebido como parâmetro os dados correspondentes à linha no `ResultSet` armazenado internamente. Neste exemplo, o mapeamento entre o `ResultSet` interno e aquele recebido como parâmetro é direto, linha por linha, mas isso não é necessário. Os campos do `ResultSet` externo podem ser preenchidos como uma combinação de campos do `ResultSet` interno ou mesmo calculados.

A execução da função `debitoCliente` é similar à execução de `dadosCliente` apresentada no exemplo anterior. É necessário criar o tipo complexo `debitoCliente` antes de se criar a função no PostgreSQL. Os comandos na Listagem 20 ilustram três cenários de consulta, com diferentes valores para o número máximo de registros retornados. Pode-se notar que quando o parâmetro é -1, todos os registros são retornados.

Listagem 17. Criação de tipo de dados no banco de dados, criação da função de invocação e exemplo de resultado de uma chamada para a função criada.

```

banco=#CREATE TYPE dadosCliente
banco=#AS ( nome_cliente TEXT, endereco TEXT, bairro_cliente TEXT);
banco=#CREATE FUNCTION dadosCliente( TEXT ) RETURNS dadosCliente
banco=#AS 'ProcedimentosBancarios.dadosCliente( java.lang.String )'
LANGUAGE java;
banco=# SELECT dadosCliente( 'Jose Antonio' );
          dadoscliente
-----
("Jose Antonio", "Av. sul N 900", "Asa Leste")
(1 row)

```

Listagem 18. Implementação da interface `ResultSetProvider`.

```

1 public static ResultSetProvider debitosClientes(int maxRecords) {
2     return new DadosDebito(maxRecords);
3 }

```

Implementando Triggers em Java

Uma trigger é uma função executada em resposta a um evento ocorrido sobre uma tabela. Os eventos que disparam uma trigger podem ser inserções, atualizações ou remoções de registros. A PL/Java dá suporte à criação de triggers em linguagem Java, bastando para isto que o método a ser invocado como uma trigger respeite a convenção de ser público, estático, não possuir valor de retorno (`void`) e receber como parâmetro um atributo do tipo `TriggerData`.

Um `TriggerData` contém informações sobre o evento que provocou a invocação da trigger. Assim, por exemplo, é possível descobrir se a trigger foi disparada por uma inserção através do método `isFiredByInsert()`, ou por uma atualização através do método `isFiredByUpdate()` ou de uma remoção pelo método `isFiredByDelete()`.

Listagem 19. Código da classe DadosDebito().

```

1 import java.sql.Connection;
2 import java.sql.DriverManager;
3 import java.sql.ResultSet;
4 import java.sql.SQLException;
5 import java.sql.Statement;
6
7 import org.postgresql.pljava.ResultSetProvider;
8
9 public class DadosDebito implements ResultSetProvider {
10     private int maxRecords;
11     private Connection connection;
12     private Statement statement;
13     private ResultSet resultSet;
14
15     public DadosDebito(int maxRecords) {
16         this.maxRecords = maxRecords;
17         calcularDebito();
18     }
19
20     private void calcularDebito() {
21         try {
22             connection = DriverManager
23                 .getConnection(DatabaseConstants.defaultURL);
24             statement = connection.createStatement();
25             resultSet = statement
26                 .executeQuery("SELECT DISTINCT c.nome_cliente,
27 (e.valor - d.saldo) AS debito FROM cliente AS c "
28 + "INNER JOIN deposito AS d on (d.nome_cliente =
29 c.nome_cliente) "
30 + "INNER JOIN emprestimo AS e on (e.nome_cliente =
31 c.nome_cliente)");
32         } catch (SQLException e) {
33             throw new RuntimeException(e);
34         }
35     }
36     @Override
37     public boolean assignRowValues(ResultSet rs, int row) throws
38     SQLException {
39         if ((maxRecords >= 0) && ((row+1) > maxRecords)) {
40             return (false);
41         }
42         if (resultSet.next()) {
43             rs.updateString(1, resultSet.getString(1));
44             rs.updateDouble(2, resultSet.getDouble(2));
45             return (true);
46         } else {
47             return (false);
48         }
49     }
50     @Override
51     public void close() throws SQLException {
52         statement.close();
53         connection.close();
54     }
55 }

```

Também é possível determinar com que frequência a trigger é invocada, se para cada registro modificado (isFiredForEachRow()) ou uma vez por evento (isFiredForStatement()). Há inúmeros outros métodos interessantes na classe TriggerData para se obter informações sobre a trigger, mas sua descrição pormenorizada foge ao escopo deste artigo.

Por meio de TriggerData é possível ainda obter os registros afetados pelo evento que é responsável pela invocação da trigger. O método getNew() retorna um ResultSet com a linha que contém o novo registro inserido ou

Listagem 20. Cenários de consulta, com diferentes valores para o número máximo de registros retornados.

```

banco=# CREATE TYPE debitoCliente
banco=# AS ( nome_cliente TEXT, debito DOUBLE PRECISION);
banco=# CREATE FUNCTION debitosClientes( INT ) RETURNS SETOF
debitoCliente
banco=# AS 'ProcedimentosBancarios.debitosClientes( int )' LANGUAGE java;
banco=# SELECT debitosClientes(1);
debitosclientes
-----
("Jose Antonio",4500)
(1 row)
banco=# SELECT debitosClientes(-1);
debitosclientes
-----
("Jose Antonio",4500)
("Renan Souza",1310)
(2 rows)
banco=# SELECT debitosClientes(0);
debitosclientes
-----
(0 rows)

```

atualizado na base de dados. Este método só deve ser chamado se a trigger foi disparada por uma inserção (isFiredByInsert() retorna true) ou uma atualização (isFiredByUpdate() retorna true) e se isFiredByEachRow() retorna true, caso contrário, não haverá novo elemento disponível para ser retornado.

De forma semelhante, o método getOld() retorna um ResultSet com a linha que contém o registro antigo removido ou atualizado na base de dados. Este método só deve ser chamado se a trigger foi disparada por uma remoção (isFiredByDelete() retorna true) ou uma atualização (isFiredByUpdate() retorna true) e se isFiredByEachRow() retorna true, caso contrário, não haverá novo elemento disponível para ser retornado.

Após esta breve introdução a TriggerData, o código da Listagem 21 representa um exemplo de implementação de trigger em Java. O método atualizaEmprestimoCliente() é exportado como trigger no banco de dados e invocado sempre que um registro da tabela deposito é atualizado. O propósito do método é realizar uma alteração no valor do empréstimo de um cliente sempre que o valor de seu saldo é atualizado. Neste caso, quando o depósito aumenta, o valor do empréstimo cai 10% do valor do aumento do saldo e quando o saldo cai o valor do empréstimo aumenta 10% do valor da queda do saldo.

Em primeiro lugar, o método retorna sem executar qualquer ação se a trigger não tiver sido disparada com frequência de linha (linha 3). Caso contrário, verifica se a trigger foi disparada por uma atualização de um registro da tabela. Feitas estas verificações iniciais, entre as linhas 10 e 33, o método calcula o novo valor do empréstimo com base na variação do saldo do depósito e atualiza o valor do empréstimo. Para obter as informações sobre a variação do saldo antes e depois da atualização, os métodos getOld() e getNew() de TriggerData são chamados nas linhas 13 e 17, respectivamente. Para obter o valor do empréstimo a ser atualizado, um método utilitário privado (valorEmprestimo()) é usado.

Os parâmetros do método (nome do cliente e nome da agência) são obtidos de um registro de depósito do cliente, previamente obtido com getOld(). O método valorEmprestimo() abre sua própria conexão, busca o registro de empréstimo correspondente ao cliente e à agência, fecha a conexão e retorna o valor de empréstimo encontrado. Por fim, o novo valor de empréstimo é aplicado à tabela emprestimo e a conexão com o banco é fechada.

Listagem 21. Exemplo de implementação de trigger em Java.

```

1 public static void atualizaEmprestimoCliente(TriggerData triggerData)
2     throws SQLException {
3     if (triggerData.isFiredForStatement()) {
4         return;
5     }
6     if (triggerData.isFiredByUpdate()) {
7         Connection connection = null;
8         PreparedStatement statement = null;
9         try {
10
11             //Calcula o novo valor do empréstimo com base na
atualizacao do
12             //saldo.
13             ResultSet resultSetOld = triggerData.getOld();
14             String nomeCliente = resultSetOld.getString("nome_
cliente");
15             String nomeAgencia = resultSetOld.getString("nome_
agencia");
16             Double saldo = resultSetOld.getDouble("saldo");
17             Double novoSaldo = triggerData.getNew().
getDouble("saldo");
18             Double valorEmprestimo =
valorEmprestimo(nomeCliente,
19                 nomeAgencia);
20             Double novoEmprestimo = valorEmprestimo
- ((novoSaldo - saldo) * 0.1);
21
22
23             // Atualiza o valor do empréstimo
24             connection = DriverManager
25                 .getConnection(DatabaseConstants.
defaultURL);
26             StringBuilder query = new StringBuilder("UPDATE
emprestimo ")
27                 .append("set valor = ? WHERE nome_cliente = ?
and nome_agencia = ?");
28             statement = connection.prepareStatement(query.
toString());
29
30             statement.setDouble(1, novoEmprestimo);
31             statement.setString(2, nomeCliente);
32             statement.setString(3, nomeAgencia);
33             statement.executeQuery();
34         } finally {
35             statement.close();
36             connection.close();
37         }
38     }
39 }
40
41 private static double valorEmprestimo(String nomeCliente, String
nomeAgencia)
42     throws SQLException {
43     Connection connection = null;
44     PreparedStatement statement = null;
45     try {
46         connection = DriverManager
47             .getConnection(DatabaseConstants.defaultURL);
48         StringBuilder query = new StringBuilder("SELECT * FROM
emprestimo ")
49             .append("WHERE nome_cliente = ? and nome_
agencia = ?");
50
51         statement = connection.prepareStatement(query.toString());
52         statement.setString(1, nomeCliente);
53         statement.setString(2, nomeAgencia);
54         ResultSet resultSet = statement.executeQuery();
55         resultSet.next();
56         double saldo = resultSet.getFloat("valor");
57         return (saldo);
58     } finally {
59         statement.close();
60         connection.close();
61     }
62 }

```

Após definir os métodos que executarão a trigger, é hora de executar alguns testes. Em primeiro lugar, é preciso criar a função no PostgreSQL que mapeia no método Java implementado e a trigger que invoca esta função. Ambas as definições são descritas nos comandos da Listagem 22.

Listagem 22. Criação da função no PostgreSQL que mapeia o método Java implementado e a trigger que invoca esta função.

```

banco=# CREATE FUNCTION atualizaEmprestimoCliente() RETURNS TRIGGER
banco=# AS 'ProcedimentosBancarios.atualizaEmprestimoCliente'
LANGUAGE Java;
CREATE FUNCTION
banco=# CREATE TRIGGER atualizaEmprestimoCliente
banco=# AFTER UPDATE
banco=# ON deposito
banco=# FOR EACH ROW
banco=# EXECUTE PROCEDURE atualizaEmprestimoCliente();
CREATE TRIGGER

```

Em seguida, é interessante verificar os registros associados ao cliente Jose Antonio antes da execução da atualização que irá disparar a trigger. No teste apresentado na Listagem 23, o saldo de Jose Antonio será modificado para que se possa verificar como essa alteração afeta o valor do empréstimo deste mesmo cliente através da execução da trigger.

Listagem 22. Verificação dos registros associados ao cliente Jose Antonio utilizando a função e trigger criadas.

```

banco=# select * from empréstimo where nome_cliente = 'Jose Antonio';
nome_agencia | numero_emprestimo | nome_cliente | valor
-----+-----+-----+-----
Paulista    | 80-00-SS          | Jose Antonio | 4845.00
(1 row)
banco=# select * from deposito where nome_cliente = 'Jose Antonio';
nome_agencia | numero_conta | nome_cliente | saldo
-----+-----+-----+-----
Paulista    | 04620-8        | Jose Antonio | 2050.00
(1 row)

```

Note que o valor do depósito é R\$4.845,00 e que o saldo do empréstimo é R\$2.050,00. Agora o valor do depósito do cliente será diminuído em R\$1.000,00. Como informado anteriormente no exemplo, esta atualização disparará a trigger que atualizará o saldo do empréstimo em 10% da variação do saldo de depósito. Neste caso, como o saldo diminuiu, o valor do empréstimo deverá aumentar. O comando que realiza a atualização no depósito do cliente é mostrado na Listagem 23.

Listagem 23. Comando que realiza a atualização no depósito do cliente.

```

banco=# update deposito set saldo = 1050 where nome_cliente = 'Jose
Antonio';
UPDATE 1

```

Para constatar que a trigger foi executada corretamente, as consultas a registros do cliente Jose Antonio executadas acima são repetidas na Listagem 24. Como pode ser visto o saldo de depósito baixou para \$1.050,00, ou seja, \$1.000,00 a menos, a atualização e o valor do empréstimo do cliente subiram para \$4.945,00, \$100,00 mais alto, ou 10% da variação do valor de depósito, conforme descrito na regra de negócio descrita no exemplo.

Listagem 24. Constatação de que a trigger foi executada corretamente.

```
banco=# select * from deposito where nome_cliente = 'Jose Antonio';
nome_agencia | numero_conta | nome_cliente | saldo
-----+-----+-----+-----
Paulista    | 04620-8      | Jose Antonio | 1050.00
(1 row)
banco=# select * from emprestimo where nome_cliente = 'Jose Antonio';
nome_agencia | numero_emprestimo | nome_cliente | valor
-----+-----+-----+-----
Paulista    | 80-00-55      | Jose Antonio | 4945.00
(1 row)
```

Definindo um descritor de implantação

Em todos os exemplos apresentados até o momento um fator que pode ser considerado uma dificuldade comum é a preparação das funções, tipos e triggers no servidor de banco de dados para mapear os métodos escritos em Java. Até o momento, todas estas definições foram feitas diretamente no terminal cliente do banco de dados. Uma maneira mais cômoda de instalar o arquivo jar no servidor é empacotar estas definições dentro do próprio jar, de forma que quando o comando `replace_jar` for executado, todas as definições sejam realizadas automaticamente. Um descritor de implantação é um arquivo que contém uma lista de comandos de instalação e uma lista de comandos de desinstalação, começando com o texto `"SQLActions[] = {"` e terminando com `"}`. O bloco `BEGIN INSTALL ... END INSTALL` delimita a região dos comandos de instalação e o bloco `BEGIN REMOVE ... END REMOVE` delimita a região de comandos de desinstalação. O descritor de implantação para todas as funções, tipos e triggers usados nos exemplos anteriores é apresentado na Listagem 25.

Todo arquivo em formato jar possui um descritor de implantação denominado `MANIFEST.MF` dentro do diretório `META-INF`. Este arquivo pode ser definido manualmente, mas quando nenhum arquivo é fornecido, a ferramenta jar cria um `MANIFEST.MF` padrão, como é o caso do arquivo `funcionalidades_bancarias.jar` em que nenhum descritor de implantação foi definido. Aqui o `MANIFEST.MF`, Listagem 26, será utilizado para definir o descritor de implantação, que corresponde ao arquivo `Banco.ddr`, que contém o código listado acima.

Cada linha em um arquivo manifest significa uma palavra-chave na forma de valores pares. Os primeiros dois pares de linha, representam a versão do arquivo manifest (conforme especificado pela Sun Microsystems) além de uma anotação indicando quem criou o arquivo. Os últimos dois pares de linha, especificam o nome do arquivo e uma anotação que informa ao módulo PL/Java que o respectivo arquivo (`Banco.ddr` neste caso) é um `SQLJDeploymentDescriptor`.

Listagem 25. Descritor de implantação para todas as funções, tipos e triggers usados nos exemplos anteriores.

```
SQLActions[ ] = {
"BEGIN INSTALL
BEGIN PostgreSQL SET search_path TO public END PostgreSQL;
CREATE OR REPLACE FUNCTION saldo( TEXT, TEXT ) RETURNS DOUBLE
PRECISION
AS 'ProcedimentosBancarios.saldo( java.lang.String, java.lang.String )'
LANGUAGE java;
CREATE OR REPLACE FUNCTION clientesComEmprestimoEmAgencia(
TEXT ) RETURNS
SETOF TEXT AS
'ProcedimentosBancarios.clientesComEmprestimoEmAgencia( java.lang.
String )'
LANGUAGE java;
CREATE OR REPLACE FUNCTION clientesDaAgencia( TEXT ) RETURNS
SETOF TEXT
AS 'ProcedimentosBancarios.clientesDaAgencia( java.lang.String )'
LANGUAGE java;
CREATE OR REPLACE TYPE dadosCliente
AS ( nome_cliente TEXT, endereco TEXT, bairro_cliente TEXT);
CREATE OR REPLACE FUNCTION dadosCliente( TEXT ) RETURNS
dadosCliente
AS 'ProcedimentosBancarios.dadosCliente( java.lang.String )'
LANGUAGE java;
CREATE OR REPLACE TYPE debitoCliente
AS ( nome_cliente TEXT, debito DOUBLE PRECISION);
CREATE OR REPLACE FUNCTION debitosClientes( INT ) RETURNS SETOF
debitoCliente
AS 'ProcedimentosBancarios.debitosClientes( int )'
LANGUAGE java;
CREATE FUNCTION atualizaEmprestimoCliente() RETURNS TRIGGER
AS 'ProcedimentosBancarios.atualizaEmprestimoCliente'
LANGUAGE Java;
CREATE TRIGGER atualizaEmprestimoCliente
AFTER UPDATE
ON deposito
FOR EACH ROW
EXECUTE PROCEDURE atualizaEmprestimoCliente();
END INSTALL ",
"BEGIN REMOVE
DROP TRIGGER atualizaEmprestimoCliente ON deposito;
DROP FUNCTION atualizaEmprestimoCliente();
DROP FUNCTION debitosClientes( INT );
DROP TYPE debitoCliente;
DROP FUNCTION dadosCliente( TEXT );
DROP TYPE dadosCliente;
DROP FUNCTION clientesDaAgencia( TEXT );
DROP FUNCTION clientesComEmprestimoEmAgencia( TEXT );
DROP FUNCTION clientesDaAgencia( TEXT );
DROP FUNCTION saldo( TEXT, TEXT );
END REMOVE"
}
```

Listagem 26. `MANIFEST.MF`.

```
Manifest-Version: 1.0
Created-By: 1.6.0_16 (Sun Microsystems Inc.)
Name: Banco.ddr
SQLJDeploymentDescriptor: TRUE
```

Executando novamente a ferramenta jar acrescentando o arquivo de manifesto com a opção `-m` e o descritor de implantação `Banco.ddr` o resultado é o apresentado na Listagem 27.

Para notar o efeito do descritor de implantação, execute as seguintes operações:

- carregue o novo jar e em seguida descarregue-o com o script remove_jar. Isto fará com que todas as definições de funções sejam removidas;
- tente executar a função debitosClientes e note que ocorrerá um erro, pois a função não está mais definida;
- instale novamente o jar e ajuste o classpath do esquema público como descrito anteriormente e tente executar novamente a mesma função;
- veja que agora o resultado esperado é retornado.

Esta sequência de ações é apresentada na Listagem 29.

Listagem 27. Execução do comando jar acrescentando o arquivo de manifesto com a opção -m e o descritor de implantação Banco.ddr.

```
jar cvf funcionalidades_bancarias.jar -m ../resources/manifest.txt
DadosCliente.class DadosDebito.class DatabaseConstants.class
IteradorDeClientes.class ProcedimentosBancarios.class ../resources/Banco.ddr
added manifest
adding: ../resources/manifest.txt(in = 118) (out= 116)(deflated 1%)
adding: DadosCliente.class(in = 1674) (out= 887)(deflated 47%)
adding: DadosDebito.class(in = 1971) (out= 1108)(deflated 43%)
adding: DatabaseConstants.class(in = 210) (out= 163)(deflated 22%)
adding: IteradorDeClientes.class(in = 3084) (out= 1600)(deflated 48%)
adding: ProcedimentosBancarios.class(in = 4769) (out= 2330)(deflated 51%)
adding: ../resources/Banco.ddr(in = 2022) (out= 539)(deflated 73%)
```

Listagem 29. Efeito do descritor de implantação.

```
banco=#SELECT sqlj.replace_jar('file:///home/paulo/projetos/pljava/
workspace/PLJava/bin/funcionalidades_bancarias.jar', 'Banco', true );
replace_jar
-----
(1 row)
banco=# SELECT sqlj.remove_jar('Banco', true);
remove_jar
-----
(1 row)
banco=# SELECT debitosClientes(1);
ERROR: java.lang.ClassNotFoundException: ProcedimentosBancarios
banco=# SELECT sqlj.install_jar('file:///home/paulo/projetos/pljava/
workspace/PLJava/bin/funcionalidades_bancarias.jar', 'Banco', true );
install_jar
-----
(1 row)
banco=# SELECT sqlj.set_classpath('public', 'Banco' );
set_classpath
-----
(1 row)
banco=# SELECT debitosClientes(1);
debitosclientes
-----
("Jose Antonio",3895)
(1 row)
```

Considerações finais

Se você é um programador Java, o módulo PL/Java é uma forma fácil de adicionar funcionalidades embutidas no servidor PostgreSQL. Esse módulo obedece a padrões como o JDBC e a iniciativa do SQLJ. A evolução das tecnologias é um fator que deve sempre ser levado em conta. Práticas atuais no desenvolvimento de software, como o reuso e o baixo acoplamento são cada vez mais levadas em consideração em projetos de software. O baixo acoplamento de um serviço está relacionado com a sua capacidade de ser independente de outros serviços para realizar a sua tarefa e essa é uma das inúmeras utilidades que a PL/Java pode proporcionar. Alguns pesquisadores em seus estudos fazem alguns questionamentos do tipo: "Java Stored Procedure é melhor que PL/pgSQL?". A resposta é: depende da aplicação. Depende do que a aplicação precisa fazer. Ambas as abordagens possuem suas vantagens. O PL/pgSQL é muito semelhante e totalmente compatível com o SQL, porém PL/pgSQL foi desenhada para cenários onde há bastante acesso a dados. Se o seu uso principal for incrementação de inteiros ou cálculos matemáticos, o PostgreSQL oferece outras PLs que executam funções diferentes. Criar stored procedure com PL/Java é ideal para aplicações baseadas em componente e principalmente para modelagem de dados com orientação a objetos. O poder do Java provê simplesmente uma maneira alternativa de desenvolver stored procedures usando a sintaxe do Java. **MU**

Referências

1. GBorg-PostgreSQL related projects, <http://gborg.postgresql.org/project/pljava/projdisplay.php>
2. http://www.amazon.com/PostgreSQL-2nd-Korry-Douglas/dp/0672327562/ref=sr_1_1?ie=UTF8&s=books&qid=1253850481&sr=8-1
3. <http://dee.feg.unesp.br/Disciplinas/SMA6023/Aulas/bancario.sql>
4. <http://osdir.com/ml/db.postgresql.pljava/2005-02/msg00039.html>
5. <http://www.slideshare.net/petereisentraut/postgresql-and-pljava>
6. <http://www.ic.unicamp.br/~geovane/mo410-091/Ch06-DBApp-art.pdf>
7. <http://pgfoundry.org/mailman/listinfo/pljava-dev>
8. <http://en.wikipedia.org/wiki/SQL>
9. Lista nacional de discussão do PostgreSQL <http://listas.postgresql.org.br/pipermail/pgbr-geral/2009-September/017458.html>
10. <http://osdir.com/ml/db.postgresql.pljava/2005-02/msg00039.html>
11. <http://www.fs.lcs.sunysb.edu/~dquigley/cse219/index.php?it=eclipse&tt=jar&pf=y>

EU USO

No banco de dados PostgreSQL tem-se um conjunto bastante variado de linguagens procedurais, sendo a PL/pgSQL sua linguagem mais conhecida, porém além desta linguagem procedural outras tantas podem ser instaladas e utilizadas e uma destas é a PL/Java. Essa possibilidade de interação com outras linguagens que o PostgreSQL possui nos dá a liberdade de testar funcionalidades que em um determinado ambiente comportam-se de uma forma mais atraente que o utilizado ao se utilizar uma linguagem procedural nativa do próprio banco de dados. A PL/Java nos possibilita fazer uso da JVM para tratar dados de uma forma simples tal como escrevemos um programa em Java respeitando as criações de classes, tipos de dados e funções em Java.

Currículo João Paulo Müller da Silva

João Paulo Müller da Silva é administrador de Banco de Dados PostgreSQL e atualmente trabalha na PST Eletronics. Atua também como instrutor de PostgreSQL nos módulos Essencial, Linguagem PL/pgSQL, Administração e Performance Tuning. É instrutor de Oracle no módulo Essencial. Graduado em Ciência da Computação pela UCPel (Universidade Católica de Pelotas) e mestrado em Bioinformática pela UNISINOS (Universidade do Vale do Rio dos Sinos).