Desenvolvimento de aplicações de Banco de Dados.

Marcílio da Silva Oliveira[1],
[1]Laboratório de Inovação em Software – UNICAMP / Ci&T
Instituto de Computação, Universidade Estadual de Campinas – SP/ Brasil.
E-mail: marcilio@ccuec.unicamp.br

Resumo

A evolução das tecnologias traz consigo novas necessidades. As aplicações utilizando banco de dados não são uma exceção e à medida que as aplicações se tornam mais dinâmicas e inovadoras, o tempo é cada vez mais cruel para os desenvolvedores. Além disso, as partes da arquitetura do sistema devem acompanhar seu ritmo de desenvolvimento. Muitas vezes, o banco de dados funciona como o coração do sistema, e seu bom funcionamento é imprescindível, e é pensando nisso que cada vez mais o desenvolvimento de aplicações de banco de dados exige uma maior atenção e dedicação. Este trabalho descreve um pouco sobre como podemos fazer a integração da aplicação com sua base de dados, dando uma atenção especial a uma API Java, o Java Data Base Connectivity (JDBC) e os drivers JDBC. Além disso, também citamos sobre como o SQL é embutido no código da aplicação, sobre a utilização de cursores e de Stored Procedures, dando uma atenção especial também à inovação de se utilizar Java como uma linguagem procedural, que foi proposta pela Oracle e considerada por gigantes no mercado como a própria IMB e o banco de dados PostgreSQL. Enfim, este trabalho dá uma visão geral do desenvolvimento de aplicações de banco de dados.

Abstract

The evolution of Technologies triggers new necessities. Applications which use a database are not an exception anymore, and as they become more dynamic and innovative, the time is something more and more cruel to developers. Besides that, parts of the system architecture must follow the rhythm of the system's development as whole. Many applications have the database as its heart, so its well-functioning is of utmost importance. For that reason, the development of database applications demands a lot of attention and dedication. This paper/article gives a slight view on how to integrate the application with the database, giving special attention to a Java API, Java Database Connectivity (JDBC) and JDBC drivers. In this article we will also be discussing how SQL can be embedded in the application's code, the utilization of cursors and stored procedures. All that, giving special attention to the use of Java as a procedural programming language, which is an innovation and has been proposed by Oracle and considered by giants in the market like IBM and the database PosgreSQL. Thus, this article gives the reader a general view of the development of database applications.

1. Introdução

Não é raro encontrarmos sistemas computacionais nos quais o banco de dados é o "coração" da aplicação e o seu bom funcionamento é imprescindível para o sucesso de todo o sistema. Com tamanha importância e com as necessidades cada vez maiores sobre quesitos como desempenho, usabilidade, produtividade, segurança entre outros, surgiram e surgem cada vez mais novas técnicas de implementação de bancos de dados, assim como novas tecnologias e suas respectivas ferramentas, visando facilitar a vida do desenvolvedor e satisfazer as exigências do usuário.
Há uma preocupação, em especial, com a forma com que as aplicações são desenvolvidas, a maneira em que a aplicação interage com o banco de dados e como integrar a linguagem "nativa"

da aplicação com a linguagem do BD e suas ações. Muitos se empenharam e ainda se empenham em busca de soluções para melhorar a comunicação entre as diferentes camadas da aplicação. Além dos interesses comerciais específicos, há uma grande demanda por parte de outras áreas de pesquisa e de atividades econômicas por soluções para seus problemas de armazenamento e gerência de dados[1].
A evolução das ferramentas também tem auxiliado muito o desenvolvimento das aplicações com banco de dados. Tais ferramentas fornecem pacotes de desenvolvimento com interfaces padronizadas, que propiciam, muitas vezes, um maior nível de abstração para o tratamento de conexões, consultas, definição e manipulação de dados e outras ações típicas de banco de dados.
Além disso, muitas das tecnologias e técnicas mais antigas persistem e, definitivamente, são inúmeras as possibilidades de implementação da base de dados. A escolha deve ser baseada nas necessidades correntes, e com tamanho leque de opções, fica mais fácil encontrar tecnologias e ferramentas compatíveis e que possibilitem a implementação de aplicações de banco de dados que satisfaçam as necessidades.
Aplicações que utilizam SGBDs para gerenciar dados executam processos separados para conectar com o SGBD. Uma vez que uma conexão é estabelecida, comandos SQL podem ser usados para inserir, apagar ou modificar dados. Consultas SQL podem ser usadas para restaurar dados desejados, mas, mas é necessário exibir uma importante diferença de como o sistema de banco de dados "vê" os dados e como uma aplicação desenvolvida em linguagens como C++ ou Java vêem os dados: o resultado de uma consulta no banco de dados é um conjunto (ou coleção) de registros, mas linguagens estruturadas com Java, por exemplo, não possui um tipo de dado correspondente. Esta "falta de combinação", conhecida como *impedance mismatch,* entre as linguagens é resolvida através de construções adicionais SQL que possibilitam às aplicações reconhecer e acessar um conjunto e interagir com um registro por vez.
O crescimento de Java como uma linguagem popular de desenvolvimento, especialmente para aplicações para *WEB*, fez com que o acesso ao SGBD através do código Java se tornasse um tópico particularmente importante. Neste trabalho, falaremos sobre o *Java Data Base Connectivity* (JDBC), uma interface de programação que nos permite executar consultas SQL de um programa Java e utilizar os resultados desta consulta dentro do programa. JDBC tem as mesmas funcionalidades de consultas estáticas SQL, porém é mais simples programar em Java utilizando o JDBC.
Geralmente é bastante conveniente executar código de aplicação no servidor de banco de dados, ao invés de simplesmente recuperar dados executar a lógica da aplicação em processos separados. *Stored Procedures* possibilitam que a lógica da aplicação seja armazenada e executada no servidor.
O objetivo deste trabalho é descrever algumas formas de integração entre a linguagem nativa da aplicação com o banco de dados. Esclarecer como "problemas de impedância" são resolvidos. Será dada uma atenção especial ao JDBC explicando suas principais características e realizando uma comparação entre seus tipos de *driver.* Além disso, também é comentado sobre as linguagens procedurais, destacando a iniciativa da Oracle de utilizar Java como uma linguagem procedural.
Inicialmente, veremos como pode ser feito o acesso ao banco de dados através da aplicação. Na seção 3, é feita uma introdução sobre o JDBC, destacando uma descrição mais detalhada dos quatro tipos de *drivers* JDBC. E, por fim, nas seções 4 e 5, é dado um *overview* de SQLJ e *Stored Procedures.*

2. Acessando o banco de dados através da aplicação.

Nesta seção, é descrito como comandos SQL podem ser executados de dentro de uma linguagem nativa de programação, como C ou Java. As declarações podem se referir às variáveis locais (incluindo variáveis especiais, usadas para retorno de status), aumentando ainda mais a integração entre a linguagem nativa e o SQL. Para isso, deve existir uma declaração para conectar ao banco de dados correto. Uma vez que uma conexão é estabelecida, comandos SQL podem ser usados para inserir, apagar ou modificar dados.
Existem duas abordagens de integração: Embutir o SQL na linguagem (ex: SQL embutido, SQLJ), e criar ou utilizar uma API especial para chamar os comandos SQL, como por exemplo, o JDBC.

O comando SQL dentro da linguagem nativa em uma aplicação é chamado SQL Embutido. Os detalhes a respeito de SQL embutido também dependem da linguagem nativa. Apesar de capacidades similares serem suportadas por uma variedade de linguagens de programação, a sintaxe varia algumas vezes.

2.1. SQL embutido

O uso de SQL embutido permite a execução de comandos SQL junto à linguagem de programação. Expressões SQL (não declarações), podem ser usadas sempre que uma declaração na linguagem nativa é habilitada.
Há duas complicações que devemos ter sempre em mente: Primeiramente, o tipo reconhecido pelo SQL pode não ser reconhecido pela linguagem nativa, ou um tipo da linguagem nativa pode não ser reconhecido pelo SQL. Para isso, é utilizado *casting* de dados, antes de passar os dados entre as duas linguagens (o SQL, assim como outras linguagens, possui operadores para converter valores de um tipo para valores de outro tipo). A segunda complicação é que as relações SQL possuem um conjunto de registros, sem limite no número de registros, por exemplo. Tradicionalmente, nenhuma estrutura de dados existe em linguagens estruturais como C++ e Java que possa endereçar ou receber o resultado das consultas (embora agora exista a STL – *Standard Template Library*). O SQL fornece o **cursor** para dar suporte a isso.
A declaração de variáveis é similar a como elas são declaradas em linguagens estruturadas, porém, elas devem ser declaradas entre os comandos `EXEC SQL BEGIN DECLARE SECTION` e `EXEC SQL END DECLARE SECTION`. Por exemplo:

```
EXEC SQL BEGIN DECLARE SECTION
char placa[8].
float ano;
EXEC SQL END DECLARE SECTION
```

Figura 1. Exemplo de declaração de variáveis em SQL embutido.

O SQL-92 padrão reconhece duas variáveis especiais para reportar erros, `SQLCODE` e `SQLSTATE`. `SQLCODE` é a mais antiga das duas, é definida para retornar um valor negativo quando ocorre um erro condicional acontece, sem especificar qual erro um valor negativo particular representa. `SQLSTATE`, introduzida no SQL-92, associa valores predefinidos com alguns erros mais comuns, introduzindo uma maior uniformidade para como os erros são reportados. Uma das duas variáveis deve ser declarada. Em C, por exemplo, `SQLCODE` é do tipo `long` e o `SQLSTATE` é `char[6]`.
Comandos SQL podem ser facilmente criados (ou declarados) dentro do código da linguagem nativa. Em C, tais expressões devem ser prefixadas por `EXEC SQL`. Como um exemplo simples, a figura 2 exibe um código de SQL embutido que insere uma linha cujas colunas têm valores definidos por variáveis, em uma tabela Carro:

```
EXEC SQL
INSERT INTO Carro VALUES (:placa, :ano);
```

Figura 2. Exemplo de comando em SQL embutido.

Outras opções, como o comando `WHENEVER` para facilitar a utilização do SQL embutido, mas, não serão detalhados neste trabalho. O objetivo aqui é apenas dar uma visão superficial do SQL embutido e suas utilizações e limitações.

2.2. Cursores

O maior problema em "embutir" comandos SQL em uma linguagem como C é o chamado problema de *impedância* ou "falta de combinação", ocorrida devido ao SQL utilizar conjunto de registros para retornar consultas, enquanto as linguagens como C não suportam uma abstração a

conjunto de registros. A solução é prover um mecanismo que nos permite acessar um registro da relação por vez. Este mecanismo é o cursor.
Um cursor pode ser declarado em qualquer relação ou qualquer consulta SQL (pois toda consulta SQL retorna um conjunto de registros). Uma vez que um cursor é declarado, nós podemos executar algumas ações sobre ele: abri-lo (***open*** - o que "posiciona" o cursor imediatamente antes da primeira linha); "consumir" a próxima linha (***fetch***); mover o cursor (***move*** – mover para a próxima linha, para após a *n-ésima* linha, para a primeira linha, para a linha anterior, etc.); ou fechar o cursor (***close***). Desta forma, o cursor nos permite essencialmente acessar uma linha (ou registro) em uma tabela através da posição do cursor em relação a uma linha e ler os seus dados. Podemos utilizar uma cláusula especial, chamada `ORDER BY`, nas consultas que serão acessadas através de um cursor, para controlar a ordem na qual as tuplas serão retornadas. Vale lembrar que a cláusula `ORDER BY`, que ordena as tuplas da resposta, é somente permitida no contexto de um cursor. Além disso, podemos também modificar ou excluir uma linha apontada pelo cursor.
Geralmente precisamos abrir o cursor se o comando SQL é um `SELECT` (ex: consulta). De qualquer forma, nós podemos evitar abrir um cursor se o resultado contém uma única linha. Comandos `INSERT`, `DELETE` e `UPDATE` geralmente não requerem cursor, ainda que algumas variantes do `DELETE` e `UPDATE` utilizem cursores.
Como exemplo, a figura 3 exibe uma consulta de carros com o ano de fabricação superior a 2000.

```
DECLARE newCars CURSOR FOR
SELECT C.numplaca, C.anocarro
FROM carro C
WHERE C.anocarro > 2000;
```

Figura 3. Declaração de um cursor, e atribuição de uma consulta ao cursor.

Esta consulta retorna uma coleção de tuplas, não apenas uma. A solução é usar o cursor. Neste caso, usamos o cursor `newCars`. Posteriormente, poderemos executar operações sobre o cursor, como:

⇒ `OPEN newCars;`
⇒ `FETCH newCars INTO :placa, :ano;`
⇒ `CLOSE newCars;`

Os cursores também possuem uma série de propriedades que podem ser validadas no momento da declaração do cursor. Propriedades como `READ ONLY`, `INSENSITIVE`, `SCROLL`, dentre outras, dão maior poder e flexibilidade aos cursores. Uma descrição de tais propriedades pode ser encontrada em [1].

2.3. SQL Dinâmico

O SQL embutido é uma forma boa e eficiente para realizar consultas nos dados de dentro da linguagem nativa, porém *strings* de consulta nem sempre são conhecidas em tempo de compilação (por exemplo: planilhas, gráficos que precisam acessar o *DataSource* do SGBD). É pra essas situações que precisaríamos de um pouco mais de flexibilidade para lidar com os comandos SQL. SQL provê algumas facilidades isso. O SQL dinâmico permite a construção de declarações SQL através de variáveis da linguagem nativa. No exemplo ilustrado na figura 4, são mostrados dois comandos chave para execução de comandos com SQL dinâmico, `PREPARE` e `EXECUTE`:

```
String  sqlQuery = "DELETE FROM Carro WHERE ano < 1980";
EXEC SQL PREPARE prepared FROM :sqlQuery;
EXEC SQL EXECUTE prepared;
```

Figura 4: Execução de um comando SQL utilizando SQL Dinâmico.

A primeira linha declara a variável `sqlQuery` e inicializa esta variável com o *string* representando o comando SQL. A segunda linha compila o comando e atribui o valor retornado à variável `prepared`. E a terceira linha executa o comando.
Muitas situações sugerem a utilização de SQL dinâmico. De qualquer forma, podemos notar que a composição do comando SQL assim como sua recompilação acontece em tempo de execução, causando um certo *"overhead"*. Comandos em SQL embutido só podem ser compostos em tempo de compilação e são re-executados sem *overhead*. Conseqüentemente, fica claro que devemos limitar a utilização de SQL dinâmico a situações nas quais ele é essencial.
Existem mais coisas a saber sobre SQL dinâmico. Por exemplo, como passar parâmetros da linguagem nativa para comandos SQL, o que torna esta opção de integração aplicação / banco de dados ainda mais poderosa.

3. Introdução ao JDBC

Uma alternativa ao SQL embutido é a utilização de *APIs* para banco de dados. Ao invés de modificar o compilador, podemos adicionar bibliotecas com as chamadas ao banco de dados, que são as *APIs*. Fornecem, muitas vezes, uma interface padronizada com procedimentos e objetos pré-definidos com os quais podemos passar *strings* SQL da linguagem nativa e apresentar os resultados de forma mais amigável. JDBC é um bom exemplo, que é uma API da Sun.
SQL embutido possibilita a integração de SQL com uma linguagem de programação. Como descrevemos, um preprocessador de um SGBD específico transforma os comandos do SQL embutido em funções chamadas de dentro da linguagem nativa. Os detalhes desta transformação variam de acordo com o SGBD, e o código pode ser compilado para rodar em diferentes SGBDs. O executável final geralmente roda apenas em um SGBD específico.
Ao contrário do SQL embutido, o JDBC possibilita um único executável acessar diferentes SGBDs sem recompilação.

3.1. Arquitetura JDBC

A utilização do JDBC pode ser tanto para arquiteturas duas camadas ou três camadas. O JDBC possui quatro componentes básicos em sua arquitetura:

i) **Aplicações:** A aplicação é desenvolvida em Java, e, de dentro do código é responsável por iniciar terminar as conexões com o banco de dados, submeter as declarações SQL, além de utilizar os cursores e outras iterações com o banco de dados.
ii) **"*Driver Manager*"** ou **Gerenciador de *Driver:*** É responsável por carregar o *driver* JDBC que será utilizado.
iii) ***Driver:*** Componente que conecta a fonte de dados, transmite as requisições e retorna os resultados obtidos e traduz estes resultados assim como os códigos de erro.
iv) ***"DataSource"*** ou **fonte de dados:** Processa as declarações SQL e retorna os resultados.

Dependendo da aplicação e do tipo de solução, alguns diferentes cenários são possíveis, os quais não serão detalhados neste trabalho. A seguir, a figura 5 ilustra a arquitetura de funcionamento do JDBC, retratando como ele funciona como uma camada intermediária entre a aplicação e o banco de dados. O JDBC pode estar na máquina do cliente ou em um servidor intermediário, dependendo a arquitetura da aplicação.

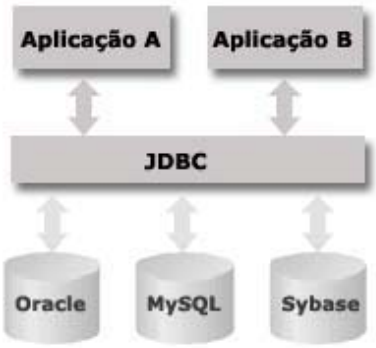

Figura 5: Arquitetura JDBC.

3.2. Classes e interfaces

JDBC é uma coleção de classes e interfaces Java que possibilita acesso ao banco de dados de programas escritos na linguagem Java. JDBC contém métodos para se criar conexões a *DataSources* remotos, executar comandos SQL, examinar conjuntos de resultados a partir de comandos SQL executados, gerenciamento de transações, verificação de exceções dentre outros.
Resumidamente, podemos ilustrar os passos para submeter uma consulta a um *DataSource* e receber os resultados da seguinte forma:

i) **Carregar o *driver*:** O primeiro passo na conexão ao *DataSource* é carregar o *driver* JDBC correspondente. No JDBC, o *driver* é gerenciado pela classe `Drivermanager`, que mantém uma lista de todos os *drivers* carregados.

ii) **Conectar ao *DataSource***: Uma sessão com um *DataSource* é iniciado com a criação de um objeto `Connection`, que identifica uma sessão lógica com o *DataSource*; conexões múltiplas com um mesmo programa Java podem se referir a diferentes *DataSources* ou a um mesmo *DataSource*. Estabelecer esta conexão não é uma operação simples e tampouco "barata", uma vez que envolve vários passos, como estabelecer uma rede de conexão com o *DataSource*, autenticar e alocar recursos.

iii) **Executar o comando SQL**: O JDBC suporta três diferentes formas de executar os comandos SQL: `Statement` (para declarações SQL tanto estáticas quanto dinâmicas), `PreparedStatement` (declarações semi-estáticas) e `CallableStatement` (chamada de *Stored Procedures*) , sendo que o `Statement` é a classe base para as duas outras classes.

O `PreparedStatement` é uma classe pré-compilada, com declarações SQL parametrizadas. A estrutura do comando é fixa e os valores dos parâmetros são definidos em tempo de execução. Um exemplo de utilização do `PreparedStatement` é mostrado na figura a seguir:

```
String sql= "INSERT INTO Carro VALUES(?,?)";
PreparedStatment pstmt=con.prepareStatement(sql);
pstmt.clearParameters();
pstmt.setString(2,placa);
pstmt.setInt(1,ano);

int numRows = pstmt.executeUpdate();
```

Figura 6: Exemplo de utilização do `PreparedStatement`.

Dentre as três opções, o `PreparedStatement` é a mais utilizada.
Para executar o comando, podemos utilizar o `executeQuery()` ou o `executeUpdate()`. O `executeQuery()` é utilizado se o comando retorna algum dado, como um SELECT. O JDBC tem seu próprio mecanismo de cursor que é o objeto `ResultSet`.

O `executeQuery()` retorna um objeto `ResultSet`, que é similar a um cursor. Ele permite uma série de operações sobre os dados retornados do banco de dados. Algumas das suas principais operações são listadas na tabela 1:

Tabela 1. Funções básicas de um `ResultSet`.

FUNÇÃO	DESCRIÇÃO
`previous()`	Move uma linha para trás
`absolute(int num)`	Move para a linha com o número especificado
`relative(int num)`	Move para frente (ou para trás) a quantidade de linhas especificada
`first()`	Move para a primeira linha
`last()`	Move para a última linha

A seção a seguir faz uma descrição dos quatro tipos de *driver* JDBC, analisando as principais vantagens (prós) desvantagens (contras) de cada *driver*.

3.3. *Drivers* JDBC

Para conectar com diferentes bancos de dados, o JDBC requer *drivers* para cada banco de dados. Os *drivers* JDBC estão divididos entre quatro tipos ou níveis. Cada tipo define uma implementação do *driver* JDBC aumentando o alto nível de independência de plataforma, desempenho e administração do processo. Os quatro tipos são:

⇒ Tipo 1: JDBC-ODBC *Bridge*.
⇒ Tipo 2: API-Nativa / parcialmente *driver*-Java
⇒ Tipo 3: *Network Bridge* / *driver*-Java.
⇒ Tipo 4: Protocolo Nativo / *driver*-java.

A seguir, os quatro tipos de *drivers* são discutidos separadamente.

a) TIPO 1:

Prove acesso JDBC via um ou mais *driver* ODBC (*Open DataBase Connectivity*). Traduz as chamadas JDBC em chamadas ODBC e as envia para o *driver* ODBC. Desta forma, o driver ODBC, deve estar presente na máquina do cliente. Utilizado para ambientes não Java. Uma ilustração deste tipo de *driver* é feita pela figura 7:

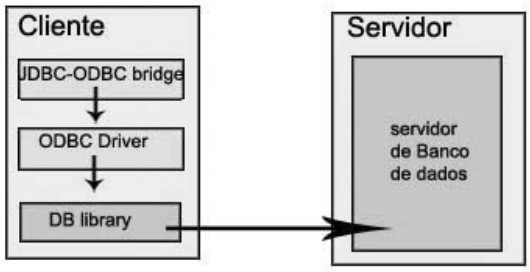

Figura 7: Tipo1, JDBC-ODBC Bridge.

✓ **Prós:**

Este tipo apresenta uma boa oportunidade para estudar JDBC. É mais útil em companhias que já possuam drivers ODBC instalados nos clientes. Pode ser talvez a única forma de acessar alguns *endpoints* (banco de dados desktop), quando estes *endpoints* utilizam windows.

O *JDBC-ODBC bridge* permite acesso a grande maioria dos bancos de dados, desde que os driver de banco de dados ODBC esteja disponível. Pode ser mais útil para empresas que tenha um driver ODBC sempre instalado nos clientes.

✓ **Contras:**
O tipo 1 não indicado para aplicações em grande escala, uma vez que o desempenho cai à medida que as chamadas JDBC trafegam através da ponte para o *driver* ODBC. Os resultados são reenviados de volta pelo processo reverso. Considerando o tópico desempenho, o tipo 1 pode não ser o mais indicado para aplicações de grande escala.
O Driver ODBC e a interface nativa precisa ser estar instalada no cliente. Então, alguma possível vantagem de utilizar aplicações Java no ambiente interno é perdida.
Além disso, ao utilizar o tipo 1, o usuário é limitado pelas funcionalidades do driver ODBC.

b) TIPO 2:
Converte chamadas JDBC em chamadas especifica de um banco de dados como SQL Server, Informix, Oracle, entre outros. O tipo 2 comunica diretamente com o servidor de banco de dados, então ele requer que alguns códigos estejam presentes no cliente. Uma ilustração deste tipo de *driver* é feita pela figura 8:

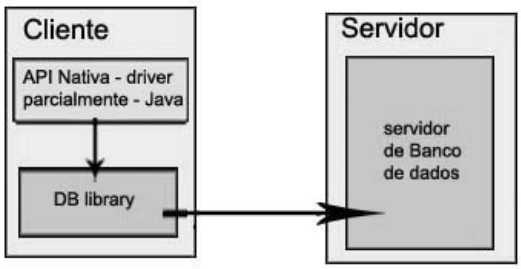

Figura 8: Tipo 2, API-Nativa/ Driver parcialmente Java

✓ **Prós:**
Geralmente oferece uma performance significativamente melhor do que o *JDBC-ODBC bridge*.

✓ **Contras:**
O usuário precisa estar certo de que o driver JDBC está em cada cliente.
A biblioteca do banco de dados deve ser carregada em cada cliente. Conseqüentemente, o tipo 2 não pode ser usado para a internet. O tipo 2 possui pior desempenho que os tipos 3 e 4.
Melhor para ambientes controlados, como em uma intranet. Onde você conhece todos os clientes que participam da rede.

c) TIPO 3:
As requisições do BD JDBC são passadas através da rede ara um servidor "*middle-tier*". O servidor "*middle-tier*" então traduz a requisição (direta ou indiretamente) para o especifico banco de dados nativo para passar a diante a requisição para o servidor com o banco de dados. Se o servidor "*middle-tier*" é escrito em Java, pode-se usar os drivers JDBC do tipo 1 ou tipo 2 para fazer isto. Uma ilustração deste tipo de *driver* é feita pela figura 9:

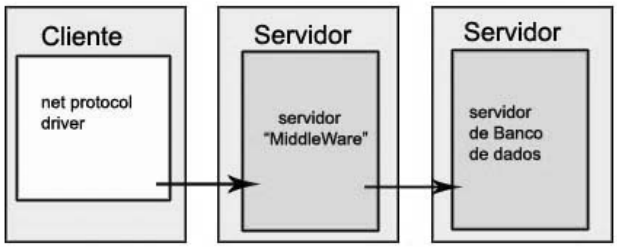

Figura 9: Tipo 9, Network bridge.

✓ **Prós:**

Melhor performance que os tipos 1 e 2.

Sua principal vantagem é a escalabilidade.

Este tipo de driver é "*server-based*", então não é necessário de algum banco de dados "*vendor library*" presente nos clientes. Com isso, futuramente, haverá muitas oportunidades para otimizar a portabilidade, o desempenho e a escalabilidade. Além disso, o "*net-protocol*" pode ser designado para fazer do cliente JDBC menor ou mais rápido para carregar. Adicionalmente, o driver tipo 3 tipicamente prove suporte a certas características como *caching*, balanceamento e administração avançada do sistema assim como *logging* e controle.

✓ **Contras:**

O tipo 3 requer o código de um banco de dados específico para ser utilizado na camada do meio. Adicionalmente, o transporte dos dados pode ser demorado, desde que os dados cheguem ao servidor final. Se o "*middle-server*" roda em uma plataforma diferente, o tipo 4 pode trazer maiores benefícios.

d) TIPO 4:

Converte chamadas JDBC dentro de pacotes que são enviados pela rede em um formato proprietário utilizado por banco de dados específicos. Possibilita uma chamada direta entre o cliente e o servidor de banco de dados. Este driver é completamente implementado em Java para alcançar a independência de plataforma. Uma ilustração deste tipo de *driver* é feita pela figura 10:

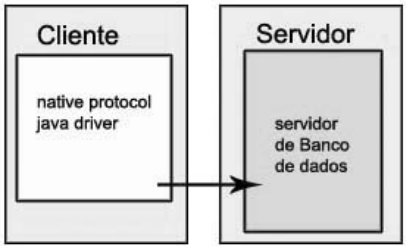

Figura 10: Tipo 4, Protocolo nativo.

✓ **Prós:**

Desde que o driver não tenha que traduzir as requisições para ODBC ou uma outra interface ou passar a requisição por outro servidor, o desempenho tende a ser melhor. Além disso, este tipo de driver ostenta uma melhor performance que os tipos 1 e 2. E também, aqui não é necessário instalar um software especial no cliente ou no servidor. Podem ser "baixados" dinamicamente.

✓ **Contras:**
Com este tipo de driver, os usuários precisam de drivers diferentes para cada banco de dados.

O tipo 1 deve ser considerado apenas para situações transitórias. Para aplicações de grande escala, devemos considerar os tipos 2, 3 ou 4.
Para aplicações em intranets, pode ser mais útil considerar o driver do tipo2. Mas, desde que os tipos 3 e 4 demonstram melhor desempenho que o tipo 2 e a tendência é desenvolver com drivers 100% Java. É melhor então, utilizar os tipos 3 e 4 em situações de intranet também.
Para aplicações na web, deve-se utilizar os drivers de tipo 3 ou 4. O tipo e é melhor para ambientes que precisam prover conectividade a uma maior variedade de SGBDs e banco de dados heterogêneos e que requerem um maior nível de concorrência entre usuários onde o desempenho e a escalabilidade são as maiores preocupações. O tipo 4 é geralmente considerados em estações e grupos de trabalhos.

4. SQLJ

O SQLJ, também conhecido 'SQL-Java', foi desenvolvido pelo *"SQL Group"*, um grupo de desenvolvedores de banco de dados da Sun. O SQLJ foi desenvolvido para complementar a forma dinâmica de criação das consultas em JDBC com um modelo estático. É, todavia, bastante semelhante com o SQL embutido.
Uma das principais preocupações do SQLJ é a simplicidade: uma linguagem concisa que permita embutir comandos SQL estáticos em programas Java. Um exemplo que ilustra bem isso é descrito na figura a seguir:

JDBC:
```
int n;
Statement stmt = conn.prepareStatement
                 ("INSERT INTO Carro VALUES (?)");
stmt.setInt(1,n);
stmt.execute ();
stmt.close();
```

SQLJ
```
int n;
#sql { INSERT INTO Carro VALUES (:n)};
```

Figura11: Comparação entre um comando no JDBC e o mesmo comando em SQLJ

Podemos destacar algumas outras características principais do SQLJ: Ele é considerado um padrão para o SQL embutido em Java, segue os padrões do SQL-92 (SQL-3 no futuro), possui verificação de tipos (estática) e verificação do esquema. Além disso, possui otimização do SQL e portabilidade.
O código SQLJ não pode ser compilado a partir de um compilador Java padrão. É necessário um "SQLJ *Translator*", capaz de gerar o código Java que pode ser compilado normalmente. Através do SQLJ, podemos criar conexões com os *drivers* dos SGBDs, utilizar expressões e variáveis da linguagem nativa, especificamente Java, chamar *stored procedures*, *stored functions,* utilizar iterators, que funcionam como cursores, para acessar os dados resultantes de uma consulta ao banco de dados.

O mapeamento dos tipos de dados Java e SQL é o mesmo utilizado pelo JDBC. O SQLJ provê um conjunto de classes que permite passar uma cadeia de caracteres ou *binary,* como argumento de entrada para um comando SQL.
O SQLJ introduz novos tipos de checagens sobre os comandos SQL, como é mostrado a seguir:

- ***Checagem off-line:***
  São verificados erros de sintaxe dos comandos SQL, sem precisar chegar o banco de dados. Além disso, é capaz de encontrar expressões SQL escritas incorretamente.
- ***Checagem on-line***
  Faz verificações no banco de dados como: Se uma tabela ou coluna foi escrita de forma errada, se uma *stored procedure* foi chamada com um número incorreto de argumentos, se a palavra chave SQL foi utilizada em um lugar inadequado, se a comparação entre dois operadores é inadequada, dentre outras verificações que só podem, de fato, serem realizadas acessando as informações no banco de dados.

Como citamos anteriormente, o SQLJ deve ser traduzido pra Java, antes do código ser compilado. A **figura 6** mostra bem o mecanismo de funcionamento o SQLJ.

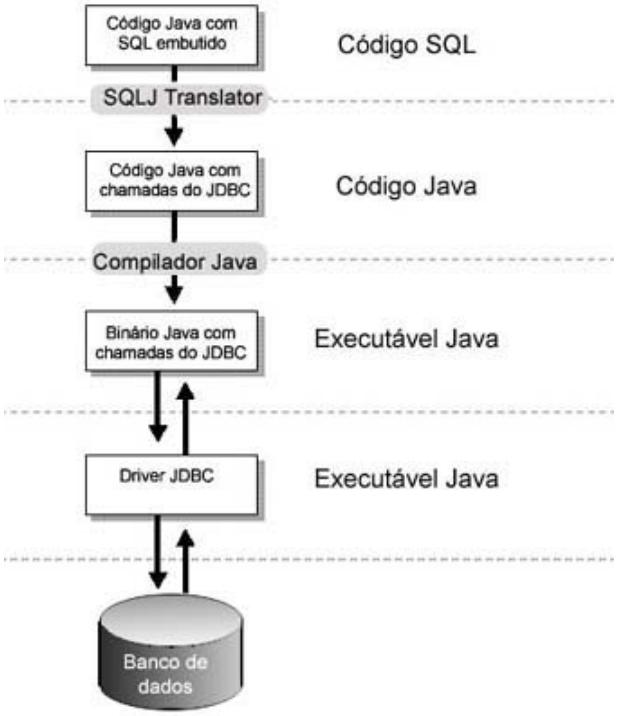

Figura 12: Mecanismo de Funcionamento do SQLJ.

Enfim, o SQLJ fornece aos desenvolvedores Java maior poder e facilidade para lidar com acesso aos dados de dentro do código. Podemos listar várias vantagens, como suporte à checagem *off-line*, o código fica mais fácil de escrever e de ser lido, tem maior performance e é baseado no SQL padrão, enquanto o SQL embutido para outras linguagens geralmente é específico de um fabricante.

5. Stored Procedures

Geralmente, pode ser muito importante executar algumas partes da lógica da aplicação diretamente no espaço de processo no sistema de banco de dados, mais especificamente, no servidor do banco de dados. Executando a lógica da aplicação diretamente no servidor tem muitas vantagens, como minimizar o tráfego de dados entre o servidor e o cliente, enquanto, ao mesmo tempo, utilizar melhor o poder de processamento do servidor.

Quando os comandos SQL são enviados remotamente pela aplicação, os registros resultantes de uma consulta devem ser transferidos do servidor para o cliente, e muitas vezes estes dados não serão considerados na aplicação. E as *stored procedures* podem minimizar tais problemas.

5.1. Conceitos e motivações

Stored procedures, ou procedimentos armazenados, são tarefas (declarações) que estão armazenadas no banco de dados e que são executadas diretamente no servidor de banco de dados.
As *stored procedures* não precisam ser escritas, necessariamente, em SQL. A maioria dos SGBDs permitem aos usuários escrevê-las em uma linguagem mais simples, genéricas, conhecidas como linguagens procedurais, que são mais poderosas que o SQL e permitem a criação de *stored procedures* mais "inteligentes", permitindo a programação de análise condicional e *loops*.
Tudo se passa como se um procedimento armazenado fosse como uma função na linguagem nativa (fazendo uma analogia), pois podemos usar parâmetros de entrada e receber o retorno. Existem vantagens e motivações para a utilização das *stored procedures* podem ser listadas, como:

- Facilidade de manutenção devido à programação modular. Quando você quer alterar uma tarefa só precisar alterar o procedimento armazenado pertinente e não precisa mudar o código do seu aplicativo.
- *Procedures* permitem que o servidor execute operações complexas em seus bancos de dados sem envolver o *software* do cliente.
- Como as *stored procedures* são analisadas e armazenadas na memória do servidor de banco de dados depois da primeira execução, sua execução é mais rápida do que usar instruções SQL, pois essas devem ser analisadas a cada chamada.
- Com as *stored procedures* armazenadas no banco de dados você não precisa enviar instruções SQL para o servidor, com isto o tráfego da rede diminui bastante.
- *Stored Procedures* permitem que dividamos tarefas complexas em módulos menores e mais lógicos e são muito úteis para efetuar tarefas de processamento periódico.
- A segurança dos seus dados também é mais robusta, pois você pode defini-la no nível de usuários atribuindo permissões aos mesmos.

No tópico a seguir, veremos alguns exemplos de linguagens procedurais, dando maior atenção ao novo e desconhecido conceito de "*Java Stored Procedures*", que é uma iniciativa da Oracle em utilizar Java como uma linguagem procedural em seus bancos de dados.

5.2. Linguagens procedurais

O padrão SQL/PSM é um bom exemplo de linguagem procedural, que é amplamente utilizado por diversos SBGDs.
Como principais construções, o SQL/PSM possibilita a declaração de variáveis locais, retorno de valore de funções, associação de variáveis com parâmetros, construção de *loops* e controle de fluxo dentre outras características. Em *procedures* em SQL/PSM também podemos utilizar consultas como parte das expressões e utilizarmos cursores com maior naturalidade, sem a necessidade do "`EXEC SQL`".
Uma iniciativa, para tornar mais simples e natural a adaptação do banco de dados com modelos orientados a objetos, é a utilização de Java como uma linguagem procedural. Tal iniciativa partiu da Oracle e é vista com bons olhos por outras grandes empresas, como

IBM, com a utilização em alguns de seus produtos, com o DB2. O banco de dados Postgresql também pretende adotar esta iniciativa[8].
Vale lembrar que o "*Java Stored Procedure"* , como já é chamado, não pretende ser um substituto às linguagens procedurais já existentes, a idéia é que ele seja uma nova opção para os desenvolvedores de banco de dados.
Como sempre, inovações trazem consigo alguns questionamentos, e, neste caso, o principal é: "*Java Stored Procedure* é melhor que PL/SQL?". A resposta é, depende da aplicação. Depende do que a aplicação precisa fazer. Ambas as abordagens possuem suas vantagens. O PL/SQL é muito semelhante e totalmente compatível com o SQL e roda mais rápido, geralmente. O *Java Stored Procedure* é ideal para aplicações baseadas em componente e principalmente para bancos com orientação a objetos. Outra questão que principalmente a Oracle vem sempre respondendo é: "Com o suporte a *Java Stored Procedure*, o Oracle deixará de suportar PL/SQL?". Com certeza não. Como dito anteriormente, Java provê simplesmente uma maneira alternativa de desenvolver *Stored Procedures*. A simplicidade do PL/SQL é uma característica que sempre faz dele indispensável, pelo menos por enquanto.
A IBM adotou a idéia de *Java Stored Procedures,* porém, a exemplo da Oracle, não deixa de suportar as linguagens procedurais padrão. Com a ferramenta Stored Procedure Builder (SPB), a IBM facilita ainda mais a vida dos desenvolvedores de banco de dados. O SPB é uma aplicação gráfica que suporta o rápido desenvolvimento de *Stored Procedures* no banco de dados DB2 (IBM). O Stored Procedure Builder permite a seus usuários:

i) Criar novas *Stored Procedures.*
ii) Executar *Stored Procedure* localmente ou remotamente em servidores DB2.
iii) Modificar *Stored Procedures* existentes.
iv) Testar e "debugar" a execução de *procedures* instaladas no servidor de BD.

Além disso, é possível exportar *Stored Procedures* e criar *Java Stored Procedures* de arquivos Java existentes. Além disso, o SPB fornece facilidades gráficas para o desenvolvedor.
Um outro banco de dados que tem se preparado para poder utilizar o *Java Stored Procedure* é o PostgreSQL, através do projeto pljava (Java para postgreSQL)[9]. Este projeto utiliza a JNI (*Java Native Interface*) para possibilitar que *Java Stored Procedure* seja executado em servidores PostgreSQL, assim como *Triggers* e funções.
Enfim, fica clara a tendência de que Java seja realmente adotada como uma linguagem procedural nos principais SGBDs do mercado.

6. Conclusões

Definitivamente, muitos esforços têm sido feitos em busca de maior eficiência e facilidade para se trabalhar com aplicações de banco de dados.
Pudemos observar que muitas técnicas são propostas para se trabalhar com banco de dados, impulsionadas pelas novas tecnologias e suas ferramentas, que tornam cada vez mais amigável a forma de se executar comandos SQL, facilitando assim a iteração entre a aplicação e a base de dados.
Apesar disso, técnicas já bastante usadas, como SQL embutido, persistem. E com o advento de novas idéias, o que podemos esperar é que cada vez mais as opções aumentem, e as alternativas para se desenvolver uma solução tornam-se cada vez mais diversas.
Hoje já podemos contar com ótimas ferramentas de desenvolvimento, APIs e tecnologias. Bons exemplos são: o Stored Procedure Building, JDBC e do *Java Stored Procedure.*
Enfim, embora as necessidades e exigências estejam sempre aumentando, existem tecnologias e ferramentas cada vez mais poderosas, o que diminui o tempo de desenvolvimento, facilitando a padronização e produtividade no desenvolvimento de sistemas.

Referências:

1. Ramakrishnan and Gehrke, *Database Management Systems*, McGraw-Hill, 3rd. edition, 2003 (185-219).
2. An Oracle White Paper, *Unleash the Power of Java Stored Procedures* – Junho, 2002.
3. ORACLE Technology Network, *Database-assisted Web Publishing using Java Stored Procedure* – Setembro, 2002.
4. ORACLE Technology Network, *Is Java better (or faster) than PL/SQL?*– Maio , 1999.
5. Marta L. Q. Mattoso, *Mini-Curso Sobre Banco De Dados*, Programa de Engenharia de Sistemas e Computação - COPPE/UFRJ, 1998.
6. Pratik Patel, *Java Database Programming with JDBC* - The Coriolis Group - Janeiro, 1996.
7. JavaWord.com - *JDBC drivers in the wild - Learn how to deploy, use, and benchmark JDBC driver types 1, 2, 3, and 4* – Julho, 2000.
8. Postgresql, *www.postgresql.org* - Visitado em [06/04]
9. GBorg-PostgreSQL related projects, *http://gborg.postgresql.org/project/pljava/projdisplay.php* – Visitado em [07/04].