Desenvolvendo Aplicações de Banco de Dados

Gustavo Maciel Dias Vieira[1] **, Vanessa Orsi Pazeto**[1]

[1]Instituto de Computação – UNICAMP

Abstract. *This paper is a short summary of the most frequently used technologies for database applications construction. It is based on the Chapter 6 of the book Database Management Systems.*

Resumo. *Este artigo fará um breve resumo das tecnologias mais usadas atualmente para construir aplicações de bancos de dados, tomando como base o Capítulo 6 do livro Database Management Systems.*

1. Introdução

Bancos de dados relacionais apareceram como uma solução para o problema de gerenciar um volume grande de informação de forma independente da aplicação. Os sistemas gerenciadores de banco de dados atuais, baseados no modelo relacional e que usam SQL como principal interface de acesso, englobam soluções muito eficientes para problemas recorrentes na manutenção de um grande volume de informações, como gerenciamento de *buffers*, geração de índices, transações, etc. Estes componentes passaram a ocupar um papel fundamental no desenvolvimento das aplicações modernas de médio e grande porte.

No entanto, banco de dados relacionais não foram concebidos para aceitar a execução de lógica procedural nativamente. A interface padrão de acesso à dados, a linguagem SQL, não possui construções procedurais e é orientado a operações sobre conjuntos de tuplas. Isto não se revelou uma limitação para as primeiras aplicações a usarem estes sistemas, pois estas se limitavam a coletar, armazenar e gerar relatórios de grandes volumes de dados. Porém, as aplicações atuais exigem um nível de complexidade muito grande, o que implica em grandes aplicações procedurais e/ou orientadas a objetos acessando o banco de dados como seu repositório de informações persistentes. Este é o modelo usualmente encontrado nas aplicações web e difundido por soluções como J2EE e .NET.

Estas novas classes de aplicações exigem interfaces eficientes entre a aplicação e o sistema gerenciador de banco de dados. Existem várias maneiras de fazer esta ligação, desde interfaces de programação (APIs) até procedimentos que executam internamente ao SGBD passando por ferramentas de mapeamento objeto relacional. Este artigo fará um breve resumo das tecnologias mais usadas atualmente para construir aplicações de bancos de dados, tomando como base o Capítulo 6 do livro Database Management Systems [Ramakrishnan and Gehrke 2003].

2. Aplicações e Banco de Dados

Quando falamos de sistemas gerenciadores de banco de dados é importante observar que estes são sistemas extremamente especializados que fazem apenas um coisa de forma muito eficiente: armazenar grandes volumes de dados. Muitos avanços em termos de estruturas de dados para armazenagem secundária, indexação, recuperação de dados, tolerância a falhas, acesso concorrente, entre outros estão representados nos sistemas usados atualmente. Além disso, o modelo relacional apresenta uma forma muito atraente para modelar e acessar dados de forma flexível e ao mesmo tempo eficiente.

Não é surpresa que no cerne da maioria das aplicações modernas de médio e grande porte exista um sistema gerenciador de banco de dados. Porém, nem todos os inúmeros aspectos de uma aplicação são modeláveis usando técnicas de modelagem da dados, nem todas as operações exigidas das aplicações pode ser implementado internamente ao SGBD. Em especial, o comportamento procedural das aplicações possui requisitos de modelagem e de implementação totalmente distintos daqueles utilizados apenas para dados.

Segundo Jim Gray [Gray 2004], a origem deste desencontro remonta à linguagem COBOL e a sua separação forçada entre a definição de dados e da lógica procedural, que ele compara ironicamente a irmãos gêmeos separados no parto e que lutam para se reencontrar. É certo que a independência os dados é uma qualidade em um sistema mas, em essência, mesmo mantendo dados e operações independentes não é possível separá-los. Grandes avanços na construção de sistemas complexos vem ocorrendo em sistemas que empregam técnicas de independência de dados que não forçam uma separação artificial entre lógica e dados, como por exemplo a orientação à objetos.

2.1. Diferença de Impedância

De qualquer forma, não nos resta escolha, para construirmos aplicações temos que fazer a ponte entre a lógica e o banco de dados. Esta ligação por sua vez expõe um problema conhecido por diferença de impedância. Esta diferença está relacionada às divergências na forma como as aplicações funcionam e como banco de dados relacionais funcionam e se apresenta de duas formas:

Procedural: Os bancos de dados trabalham apenas com conjuntos de tuplas, enquanto que a maioria das linguagens procedurais trabalha apenas com elementos individuais.

Orientada à Objetos: O modelo relacional não suporta diretamente noções OO como herança, polimorfismo e agregação.

A diferença de impedância não é uma barreira intransponível para a integração de aplicação e banco de dados e pode ser vencida como veremos nas próximas seções. Mesmo assim, a existência desta diferença cria dificuldades para a integração sem costuras destes dois componentes e deve sempre ser levada em consideração ao se planejar e construir tais sistemas.

2.2. Modelos de Integração

Considerando que conseguimos resolver este problema de acoplamento, ainda resta uma decisão de cunho arquitetural para conseguir a integração do banco de dados à aplicação; onde colocar a lógica procedural. De forma geral, existem dois pontos possíveis de acoplamento: fora do espaço de endereçamento do SGBD ou dentro do espaço de endereçamento do SGBD. A Figura 1 apresenta as duas abordagens e tecnologias que fazem uso delas.

Acesso fora do SGBD é feito usualmente via uma interface de programação (API) de acesso. Cada banco de dados define a sua própria API e portabilidade de acesso entre banco de dados diferentes é alcançado por uma camada que embrulha a API original. Existem APIs de acesso a banco de dados em praticamente todas as linguagens modernas, desde C até Java passando por Pascal, Perl, Python, entre outras. O acesso dentro do

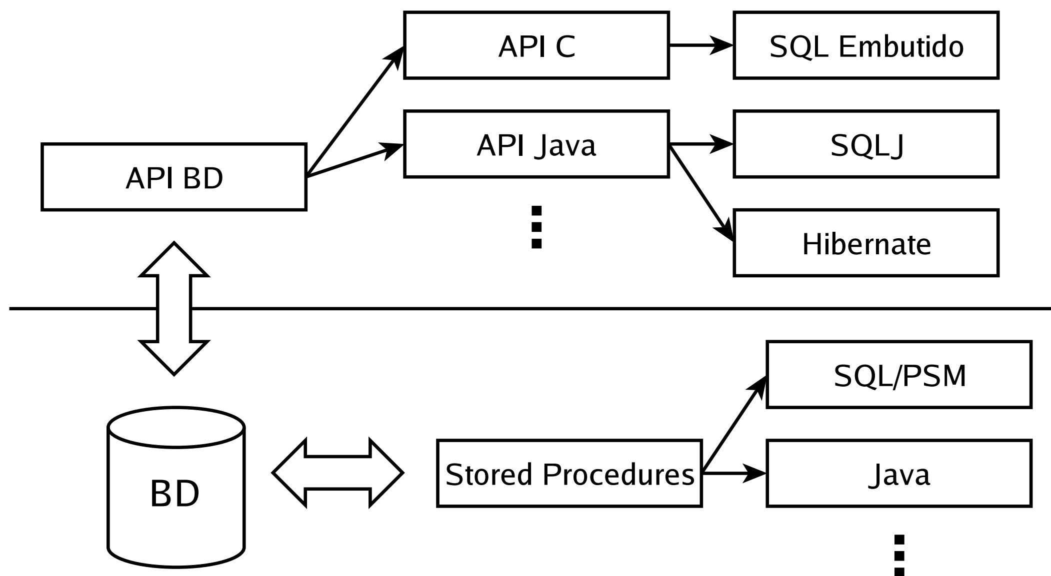

Figura 1. Modelos de integração.

SGBD é feito por um mecanismo chamado *stored procedures*, que permitem a execução de trechos de código procedural diretamente no processo do SGBD. Esta abordagem permite aumentar o poder do SGBD sem mudar muito as suas características de funcionamento.

No restante deste resumo, vamos descrever as seguintes tecnologias:

SQL Embutido: Um pré-processador para API na linguagem C.
JDBC: Uma API na linguagem Java.
SQLJ: Um pré-processador para API na linguagem Java.
SQL/PSM e Java: Usados como *stored procedures*

3. SQL Embutido

3.1. Visão Geral

O SQL Embutido foi definido no padrão SQL-92 como uma forma de acessar bancos de dados SQL a partir de uma linguagem alvo (*host language*). O SQL Embutido é definido para a linguagem C/C++ e permite intercalar comandos SQL e instruções normais da linguagem. Isto é alcançado por meio de um pré-processador independente do compilador que transforma instruções SQL marcadas com sintaxe semelhante a do pré-processador C em chamadas de API. Uma vez feita esta transformação, o código pode ser compilado usando um processador C comum.

SQL Embutido apresenta uma forma mais amigável e limpa de acessar a API C do banco de dados, representando de forma mais direta as chamadas SQL. Como é um padrão definido pelo mesmo comitê que definiu a linguagem SQL, o SQL Embutido permite acesso uniforme a diferentes APIs de vários fabricantes de bancos de dados.

A abordagem adotada pelo SQL Embutido apresenta algumas vantagens e desvantagens em relação ao uso direto da API. A principal delas é que os detalhes da API não são visíveis, permitindo assim que o código torne-se mais legível. Também é possível fazer

alguma verificação sintática da linguagem SQL em tempo de compilação. Por outro lado, é necessário realizar um passo extra de compilação e o programa resultante, apesar de muito próximo, não é C/C++, o que pode quebrar ferramentas de desenvolvimento. Um outro problema encontrado é que as mensagens de erro geradas durante a compilação podem não ser relacionadas diretamente ao texto do programa antes do pré-processamento.

3.2. Principais Construções

Como é construído como uma linguagem de substituição de macros, o SQL Embutido possui a aparência de macros usados em C. O Exemplo 1 mostra as construções mais comuns da linguagem. O primeiro comando estabelece a conexão com o banco de dados, o segundo comando delimita a declaração de variáveis e o terceiro comando executa uma consulta SQL.

Exemplo 1 Principais Construções do SQL Embutido

```
EXEC SQL CONNECT
EXEC SQL BEGIN (END) DECLARE SECTION
EXEC SQL <statement>;
```

As consultas SQL podem se referir a variáveis locais do programa C, tanto para obter valores usados na consulta quanto para retornar resultados das mesmas. Porém, para tal é necessário que as declarações das variáveis que serão acessadas em SQL estejam delimitadas como no Exemplo 2.

Exemplo 2 Declaração de Variáveis

```
EXEC SQL BEGIN DECLARE SECTION
char c_sname[20];
long c_sid;
short c_rating;
float c_age;
EXEC SQL END DECLARE SECTION
```

Dentro do comando SQL, o acesso às variáveis locais é feito prefixando as mesmas com dois pontos (`:`), produzindo consultas como esta:

```
EXEC SQL SELECT sname INTO :c_sname FROM Sailors
         WHERE sid = :c_sid;
```

Além das variáveis definidas pelo usuário, existem duas variáveis pré-definidas para indicar condições de erro:

SQLCODE: Se for negativa indica que um erro ocorreu.
SQLSTATE: Contém o código do erro encontrado.

3.3. Cursores e Processamento de Consultas

Normalmente uma consulta não retorna apenas uma tupla, mas o conjunto delas que correspondem às condições especificadas na cláusula `WHERE`. Este é um caso da diferença de impedância procedural e tem como solução o mecanismo de cursores. De forma geral,

um cursor é uma forma de percorrer de forma iterativa um conjunto qualquer de tuplas, executando o processamento para cada uma delas. Como a diferença de impedância procedural é comum a todas as linguagens procedurais, o mecanismo de cursor aparece não só no contexto do SQL Embutido.

Um cursor é declarado a partir de uma relação ou uma consulta. Na declaração de um cursor, é possível usar uma cláusula especial chamada `ORDER BY` para controlar a ordem na qual as tuplas serão retornadas. O Exemplo 3 mostra a declaração de um cursor que retorna os nomes dos marinheiros que reservaram um barco vermelho, em ordem alfabética.

Exemplo 3 Declaração de um Cursor

```
EXEC SQL DECLARE sinfo CURSOR FOR
    SELECT S.sname FROM Sailors S, Boats B, Reserves R
    WHERE S.sid=R.sid AND R.bid=B.bid AND B.color='red'
    ORDER BY S.sname
```

Uma vez declarado, o cursor pode ser aberto e ter as suas tuplas acessadas uma por uma. O comando `FETCH` é usado para ler os valores que compõem a tupla e colocá-los em variáveis locais. O Exemplo 4 mostra um exemplo mais completo onde um cursor é declarado e todas as tuplas resultantes da sua execução são impressas.

Exemplo 4 Usando `FETCH` para Acessar um Cursor

```
char SQLSTATE[6];
EXEC SQL BEGIN DECLARE SECTION
char c_sname[20]; short c_minrating; float c_age;
EXEC SQL END DECLARE SECTION
c_minrating = random();
EXEC SQL DECLARE sinfo CURSOR FOR
    SELECT S.sname, S.age FROM Sailors S
    WHERE S.rating > :c_minrating ORDER BY S.sname;
do {
  EXEC SQL FETCH sinfo INTO :c_sanem, :c_age;
  printf("%s is %d years old\n", c_sname, c_age);
} while (SQLSTATE != '02000');
EXEC SQL CLOSE sinfo;
```

3.4. SQL Dinâmico

Em muitas aplicações, as consultas SQL nem sempre são conhecidas em tempo de compilação. Um exemplo destas aplicações seria uma ferramenta de relatórios onde o usuário pode entrar a sua própria consulta SQL. Para atender a esta classe de aplicações, o SQL Embutido permite que as consultas sejam construídas em tempo de execução usando mecanismo de SQL Dinâmico. Na prática, isto permite o acesso direto à API do banco de dados, mas ainda de forma portável. O Exemplo 5 mostra o uso do SQL Dinâmico.

Exemplo 5 SQL Dinâmico

```
char c_sqlstring[ ] =
    {"DELETE FROM Sailors WHERE rating > 5"};
EXEC SQL PREPARE readytogo FROM :c_sqlstring;
EXEC SQL EXECUTE readytogo;
```

4. JDBC

4.1. Visão Geral

A abordagem de mesclagem com a linguagem alvo adotada pelo SQL Embutido tem como objetivo esconder a complexidade da API de acesso ao banco de dados subjacente. Uma outra forma de tentar resolver o mesmo problema seria tornar a API mais simples de usar. É esta saída que o padrão JDBC adota para acesso a bancos de dados a partir da linguagem Java.

O JDBC utiliza as características de orientação à objetos da linguagem Java para definir uma API de fácil uso. Ao contrário de outras APIs, JDBC é um padrão que independe do banco de dados sendo usado. O mesmo código JDBC funciona com qualquer banco de dados, sem recompilação, desde que um *driver* apropriado seja fornecido pelo fabricante do SGBD. Esta portabilidade é atingida por meio da arquitetura em camadas do JDBC, composta por quatro componentes:

Aplicação: Inicia e termina conexões, envia declarações SQL usando uma API genérica.

Gerenciador de *driver*: Carrega o driver JDBC apropriado.

***Driver*:** Conecta à fonte de dados, transmite requisições e retorna resultados e códigos de erro usando a API específica do banco de dados.

Fonte de dados: Processa as declarações SQL e retorna os resultados.

Destes componentes o mais simples de entender é a fonte de dados. Este componente nada mais é que o SGBD que queremos acessar a partir da aplicação. Usando JDBC, o acesso a esta fonte de dados é feita seguindo os seguintes passos que envolvem os componentes da arquitetura:

1. Carregar o driver JDBC.
2. Conectar à fonte de dados.
3. Executar consultas SQL.

No restante desta seção serão apresentados os componentes da arquitetura e como eles são empregados para realizar o acesso ao banco de dados.

4.2. *Drivers* JDBC

Um *driver* JDBC é uma camada de tradução entre a API genérica e a API ou protocolo específicos empregados pelo SGBD. Existem vários tipos de *drivers* JDBC, que variam de acordo com a forma como eles fazem esta tradução:

Ponte (Tipo I): Traduz as funções JDBC chamadas por uma outra API não nativa, como por exemplo ODBC (*JDBC-ODBC bridge*).

Tradutor direto para a API nativa não Java (Tipo II): Acessa a API nativa da fonte de dados diretamente, usando uma biblioteca não portável.

Ponte em rede (Tipo III): Envia comandos pela rede para um servidor, que traduz as requisições JDBC para a API específica do SGBD.

Tradutor direto para a API nativa em Java (Tipo IV): Converte as funções JDBC para acesso direto ao banco de dados, sem camadas intermediárias.

O mecanismo de *drivers* é umas das características mais atraentes do JDBC. Permitindo não só portabilidade de aplicações entre bancos de fornecedores diferentes, mas o acesso simultâneo a vários SGBDs.

Os *drivers* de JDBC são classes Java e a sua carga é controlada pelo gerenciador de *drivers*. Existem duas formas para configurar o gerenciador de *drivers* e instrui-lo a carregar um *driver* específico, por exemplo `oracle.jdbc.driver.Oracledriver`. A primeira é de forma programática no código Java, com a seguinte instrução:

```
Class.forName("oracle.jdbc.driver.Oracledriver");
```

A segunda é na linha de comando usada para invocar a máquina virtual Java:

```
-Djdbc.drivers=oracle.jdbc.driver.Oracledriver
```

4.3. Conexões com o Banco de Dados

Uma vez que o *driver* correto está carregado, resta estabelecer a conexão ao banco de dados. Toda a interação com o SGBD acontece através de conexões, sendo cada conexão um identificador de uma sessão lógica.

Para o estabelecimento de uma conexão, é necessário a determinação do *driver* a ser usado e da localização do banco de dados. Isto é feito por meio de uma URL JDBC como esta:

```
jdbc:<driver>:<parâmetros>
```

O Exemplo 6 mostra o estabelecimento de uma conexão JDBC. Uma vez estabelecida a conexão, a mesma é acessada por um objeto da classe `Connection` obtido durante o processo de abertura da conexão. Este objeto possui várias funções, entre ela é possível determinar o nível de isolamento entre as transações, se as transações são confirmadas automaticamente e se a conexão permanece aberta.

Exemplo 6 Estabelecimento de Conexão JDBC

```
String url = "jdbc:oracle:www.bookstore.com:3083";
Connection con;
try {
  con =
    DriverManager.getConnection(url, usedId, password);
} catch (SQLException excpt) { ... }
```

4.4. Declarações SQL

Usando a conexão recém aberta, existem três formas diferentes de executar declarações SQL: `Statement`, `PreparedStatement` e `CallableStatment`. Cada uma destas formas é representada por um objeto que pode ser obtida a partir da conexão e a diferença entre elas deve-se a forma como são utilizadas.

Um `Statement` é usado para declarações SQL executadas apenas uma vez, enquanto que um `PreparedStatement` é usado para declarações SQL usadas várias vezes que são parametrizadas e compiladas no servidor. Por fim, `CallableStatment` é usado para chamar um *stored procedure*. O Exemplo 7 mostra a execução de uma consulta SQL por meio de um `PreparedStatement`.

Exemplo 7 Execução de um `PreparedStatement`

```
String sql = "INSERT INTO Sailors VALUES(?,?,?,?)";
PreparedStatment pstmt = con.prepareStatement(sql);
pstmt.clearParameters();
pstmt.setInt(1, sid);
pstmt.setString(2, sname);
pstmt.setInt(3, rating);
pstmt.setFloat(4, age);
int numRows = pstmt.executeUpdate();
```

A forma como uma declaração é chamada depende da existência ou não de valores de retorno. No Exemplo 7 a consulta não retorna dados, então foi usado o método `executeUpdate` para executar a consulta. Este método retorna apenas o número de registros afetados pela consulta. Caso a consulta retorne dados, ela deve ser executada pelo método `executeQuery`. Este método retorna os dados em um objeto da classe `ResultSet` que tem funcionamento equivalente a um cursor e pode ser processado tupla por tupla, como mostra o Exemplo 8.

Exemplo 8 Um `ResultSet`

```
ResultSet rs=pstmt.executeQuery(sql);
While (rs.next()) {
   ...
}
```

Um ResultSet é um cursor muito poderoso, que oferece, entre outras, as seguintes operações:

`next()`: Move para a próxima linha.
`previous()`: Move uma linha para trás.
`absolute(int num)`: Move para uma linha específica.
`relative(int num)`: Move de forma relativa um certo número de linhas.
`first()` e `last()`: Move para a primeira ou última linhas, respectivamente.

O `ResultSet` é um exemplo de como Java permite que a API seja mais amigável. O cursor representado pelo `ResultSet` na verdade é um tipo nativo de Java que representa um conjunto de elementos. Eles pode ser processados um a um, com semântica de cursor, mas também podem ser manipulados como uma entidade só, definido-se métodos apropriados.

Em cada linha representada no `ResultSet`, é possível resgatar o valor de cada coluna da relação usando métodos apropriados e respeitando a conversão de tipos como exibido na Tabela 1.

Tipo SQL	Classe Java	Método `ResultSet`
BIT	Boolean	getBoolean()
CHAR	String	getString()
VARCHAR	String	getString()
DOUBLE	Double	getDouble()
FLOAT	Double	getDouble()
INTEGER	Integer	getInt()
REAL	Double	getFloat()
DATE	Date	getDate()
TIME	Time	getTime()
TIMESTAMP	TimeStamp	getTimestamp()

Tabela 1. Tipos de Dados em Java e SQL

4.5. Exceções e Alertas

Outra característica bem interessante do JDBC e que os erros são indicados usando o mecanismo padrão de exceções de Java. Desta forma o tratamento dos mesmos é feito de forma bem mais simples que em APIs C. De forma geral, existem duas exceções padrão que podem ser levantadas em caso de erro:

`SQLException`: Representa um erro sério que interrompe a execução a consulta.

`SQLWarning`: Representa um erro menos grave e não é levantada, sua existência deve ser explicitamente testada.

O Exemplo 9 mostra o uso de exceções e alertas. É interessante observar neste exemplo que os alertas vão verificados duas vezes: uma vez ao criar a expressão da consulta e outra vez depois da execução da mesma. As exceções por sua vez só podem ser lançadas durante a execução da consulta e são tratadas na cláusula `catch`.

Exemplo 9 Exceções e Alertas

```
try {
  stmt = con.createStatement();
  warning = con.getWarnings();
  while(warning != null) {
    warning = warning.getNextWarning():
    ...
  }
  con.clearWarnings();
  stmt.executeUpdate(queryString);
  warning = con.getWarnings();
  ...
}
catch( SQLException SQLe) {
  ...
}
```

4.6. Metadados do Banco de Dados

JDBC permite acesso ainda a informações sobre o próprio sistema de banco de dados através da classe `DatabaseMetaData`. Esta classe permite descobrir quais relações estão definidas no banco, quais as suas colunas e seus tipos de dados, além de outras meta-informações. O Exemplo 10 mostra algumas das informações que podem ser obtidas.

Exemplo 10 Metadados do Banco de Dados

```
  DatabaseMetaData md = con.getMetaData();
  System.out.println("Name:" + md.getDriverName() +
                     "version: " + md.getDriverVersion());
  ResultSet trs = md.getTables(null, null, null, null);
  String tableName;
  While(trs.next()) {
    tableName = trs.getString("TABLE_NAME");
    System.out.println("Table: " + tableName);
    ResultSet crs =
      md.getColumns(null, null, tableName, null);
    while (crs.next()) {
      System.out.println(crs.getString("COLUMN_NAME");
    }
}
```

4.7. Um Exemplo Completo

O Exemplo 11 mostra um programa completo que faz uso do JDBC. Neste exemplo, um `Statement` é usando para fazer a consulta de todos os marinheiros, que são exibidos na tela.

Exemplo 11 Um Exemplo Completo do Uso de JDBC

```
  Connection con =
    DriverManager.getConnection(url, "login", "pass");
  Statement stmt = con.createStatement();
  String query = "SELECT name, rating FROM Sailors";
  ResultSet rs = stmt.executeQuery(query);
  try {
    while (rs.next()) {
      String s = rs.getString("name");
      Int n = rs.getFloat("rating");
      System.out.println(s + "   " + n);
    }
  } catch(SQLException e) {
    System.out.println(e.getMessage() + e.getSQLState() +
                       e.getErrorCode());
  }
```

5. SQLJ

5.1. Visão Geral

Seguindo a mesma linha adotada em relação ao SQL Embutido, o comitê de padronização responsável pelo SQL-99 aumentou o número de linguagens alvo para integração com SQL. A linguagem Java foi uma das escolhidas como linguagem alvo, dando origem ao SQLJ. Desta forma, o SQLJ complementa as consultas puramente dinâmicas do JDBC com um modelo estático.

Seguindo a filosofia de inclusão em uma linguagem alvo, o SQLJ faz pré-processamento, através de um programa de tradução, de consultas SQL embutidas no código do programa em chamadas de uma API de acesso a banco de dados, usualmente JDBC. Desta forma, o pré-processador pode realizar verificações de sintaxe, tipos fortes e consistência das consultas com o esquema de banco de dados. Ou seja, o SQLJ possui as mesmas vantagens e desvantagens em relação ao uso direto do JDBC que o SQL Embutido possui em relação a API C, como descrito na Seção 3.1..

5.2. Principais Construções

Da mesma forma que o JDBC usa características de Java para tornar a API mais amigável, o SQLJ também é mais simples de usar que o SQL Embutido. Dois exemplos disso são o uso das variáveis locais e o tratamento de erros. Em SQLJ não é necessário declarar as variáveis locais que serão acessadas em SQL de forma especial. Qualquer variável pode ser acessada, bastando para isso prefixá-la com dois pontos (`:`).

A única declaração especial que deve ser feita em SQLJ é a de um objeto da classe `Iterator` que recebe o resultado das consultas e que funciona como um cursor sobre o resultado das mesmas. É interessante observar que este cursor é implementado usando um tipo padrão em Java, mais fácil de processar que um `ResultSet`. Todas as expressões SQLJ são prefixadas por `#sql`, que é usado tanto para declarar o iterador quanto para executar as consultas SQL. O Exemplo 12 mostra a declaração de um iterador e a execução de uma consulta, fazendo o paralelo das mesmas operações em JDBC.

Exemplo 12 Elementos Básicos de SQLJ e JDBC

SQLJ:
```
  #sql iterator Sailors(Int sid, String name);
  Sailors sailors;
  #sql sailors = {
    SELECT sid, name FROM Sailors WHERE rating = :rating
  };
```
JDBC:
```
  PreparedStatement stmt = connection.prepareStatement(
    "SELECT sid, name FROM Sailors WHERE rating = ?");
  stmt.setInt(1, rating);
  ResultSet var = stmt.executeQuery();
```

5.3. Iteradores por Nome e Posicionais

Uma vez que o iterador foi iniciado com o resultado de uma consulta, existem duas formas para processar os valores contidos nele: por nome e por posição.

Processar os iteradores por nome é forma padrão de fazer o processamento de uma lista em Java. Neste modo, acessamos cada coluna pelo seu nome no iterator. O Exemplo 13 mostra um exemplo completo que lista na tela todos os marinheiros que possuem um dada classificação.

Exemplo 13 Iterador por Nome

```
Integer rating;
#sql iterator Sailors(Integer sid, String name);
Sailors sailors;
#sql sailors = {
  SELECT sid, name FROM Sailors WHERE rating = :rating
};
while (sailors.next()) {
  System.out.println(sailors.sid + sailors.name));
}
sailors.close();
```

Iteradores posicionais procuram simular em Java o comportamento dos cursores encontrado no SQL Embutido. Neste caso, são especificados somente o tipo da variável para cada coluna do iterador. Quando a consulta é executada, as posições são preenchidas de acordo com a ordem que as colunas são encontradas na relação. Para acessar individualmente cada coluna é usado o operador `FETCH`. O Exemplo 14 mostra o uso de iteradores posicionais realizando a mesma operação do Exemplo 13.

Exemplo 14 Iterator Posicional

```
Integer sid; String name; Integer rating;
#sql iterator Sailors(Integer, String);
Sailors sailors;
#sql sailors = {
  SELECT sid, name FROM Sailors WHERE rating = :rating
};
while (true) {
  #sql {FETCH :sailors INTO :sid, :name} ;
  if (sailors.endFetch()) { break; }
  System.out.println(sid + name));
}
```

6. *Stored Procedures*

6.1. Visão Geral

Um *stored procedure* é um programa composto por declarações SQL executadas dentro do espaço de processamento do servidor de banco de dados. O resultado final é retornado

à aplicação sem que haja necessidade de transmitir dados intermediários. Uma vez registrado dentro de um SGBD, um *stored procedure* pode ser utilizado por vários usuários sem que haja necessidade de implementar novamente as consultas. Os programadores de aplicação não precisam possuir conhecimento do esquema do banco de dados, uma vez que o acesso ao mesmo é encapsulado dentro dos *stored procedures*.

O uso de *stored procedures* apresenta algumas vantagens como:

- Podem encapsular a lógica da aplicação enquanto permanece opacos para os dados, permitindo que o servidor efetue operações complexas sem envolver o software cliente, reduzindo a complexidade do mesmo.
- Permitem o reuso da lógica da aplicação por diferentes usuários, pois podem ser compartilhados por todas as aplicações cliente que acessam o banco de dados. Desta forma, não é necessário programar a mesma lógica dentro de cada aplicação, pois esta só é programada e testada uma única vez no servidor.
- Evitam o retorno de tuplas uma a uma através de cursores, reduzindo o tráfico na rede.
- Permitem a divisão de tarefas complexas em módulos menores e mais lógicos.
- São bastante úteis para efetuar tarefas de processamento periódico, fornecendo melhor concorrência entre o cliente e o servidor.

6.2. Criação de *Stored Procedures*

Um *stored procedure* é muito semelhante a uma consulta SQL comum, sendo a principal diferença o fato de que *stored procedures* possuem parâmetros, que são tipos válidos em SQL. Há três tipos diferentes de parâmetros:

IN: Argumentos de entrada para o *stored procedure*.
OUT: Argumentos retornados pelo *stored procedure*.
INOUT: Combinação das propriedades de IN e OUT.

O Exemplo 15 mostra a criação de um *stored procedure* que incrementa a classificação de um marinheiro, localizado pelo seu identificador.

Exemplo 15 Criação de um *Stored Procedure*

```
CREATE PROCEDURE IncreaseRating(IN sailor_sid INTEGER,
                                IN increase INTEGER)
  UPDATE Sailors SET rating = rating + increase
    WHERE sid = sailor_sid
```

Em sistemas gerenciadores de banco de dados mais modernos, *stored procedures* não precisam ser escritos apenas em SQL. Na Seção 6.4. será apresentada uma versão procedural de SQL, mas além deste tipo de extensão, é possível definir *stored procedures* em linguagens de uso geral. O Exemplo 16 mostra um *stored procedure* definido em Java.

6.3. Invocação de *Stored procedures*

A forma de chamar um *stored procedure* depende do ambiente de programação que está sendo usado. O Exemplo 17 mostra como isto é feito para os ambientes apresentados anteriormente neste resumo.

Exemplo 16 *Stored Procedure* em Java

```
CREATE PROCEDURE TopSailors(IN num INTEGER)
LANGUAGE JAVA
EXTERNAL NAME "file:///c:/storedProcs/rank.jar"
```

Exemplo 17 Invocação de *Stored procedures*

SQL Embutido:

```
EXEC SQL BEGIN DECLARE SECTION
int sid; int rating;
EXEC SQL END DECLARE SECTION
EXEC CALL IncreaseRating(:sid,:rating);
```

JDBC:

```
CallableStatement cstmt =
  con.prepareCall("{call ShowSailors}");
ResultSet rs = cstmt.executeQuery();
```

SQLJ:

```
#sql iterator ShowSailors(...);
ShowSailors showsailors;
#sql showsailors={CALL ShowSailors};
```

6.4. SQL/PSM

Como dito anteriormente, a maioria dos SGBDs modernos permitem aos usuários escrever *stored procedures* em uma linguagem procedural. A maioria deles define uma linguagem simples de uso geral derivada do SQL pela adição de elementos procedurais. O padrão SQL/PSM é um representante desta classe de linguagens.

O Exemplo 18 mostra as principais construções encontradas em SQL/PSM. Além de definir procedimentos, é possível definir funções em SQL/PSM que retornam um valor usando o operador `RETURN`. É possível declarar variáveis locais com `DECLARE` e atribuir valores a elas usando-se o operador `SET`. Consultas podem fazer parte das expressões, e ter os seus valores atribuídos a variáveis ou usados em comparações.

Os elementos procedurais do SQL/PSM mais importantes são os controles de fluxo e laços. O controle de fluxo é feito por meio dos operadores `IF THEN ELSE` como exibido no Exemplo 18 e laços são controlados pelo operador `LOOP`. Cursores são usados de forma semelhante aquela do SQL Embutido, mas sem as penalidades de desempenho associadas pois o acesso é feito dentro do servidor de banco de dados.

7. Conclusão

No contexto das aplicações complexas usadas atualmente, o papel dos sistemas gerenciadores de bancos de dados é cada vez mais amplo. Desta forma, a integração dos vários componentes do sistema é vital e inclui a integração fundamental de dados e aplicações. Este artigo apresentou um breve resumo das tecnologias mais usadas atualmente para integrar aplicações e bancos de dados, tomando como base o Capítulo 6 do livro Database Management Systems [Ramakrishnan and Gehrke 2003]. Estas tecnologias foram agru-

Exemplo 18 Construções de SQL/PSM

```
CREATE FUNCTION rate Sailor(IN sailorId INTEGER)
       RETURNS INTEGER
  DECLARE rating INTEGER
  DECLARE numRes INTEGER
  SET numRes = (SELECT COUNT(*) FROM Reserves R
                                WHERE R.sid = sailorId)
  IF (numRes > 10) THEN rating = 1;
                   ELSE rating = 0;
  END IF;
  RETURN rating;
```

padas de acordo com uma classificação simples baseada no ponto de costura arquitetural das mesmas: fora e dentro do espaço de endereçamento do SGBD.

Fora do SGBD o acesso é feito via algum tipo de interface de programação. Como estas interfaces tendem a ser bastante complexas, soluções baseadas em pré-processamento para embutir trechos de SQL em linguagens alvo oferecem um ambiente mais simples de integração. SQL Embutido e SQLJ são soluções baseadas nesta idéia. No entanto, existem outras formas de acessar uma interface de programação. O JDBC é uma API que mostra que não necessariamente a API usada para acessar o banco de dados deve ser complexa.

Dentro do SGBD o mecanismo padrão usado para a implementação de lógica de aplicação são os *stored procedures*. Estes trechos de lógica procedural executam no servidor de banco de dados possibilitando um acesso bem direto aos dados. A maioria dos *stored procedures* é escrito em uma linguagem procedural derivada de SQL, das quais SQL/PSM é um exemplo. No entanto, também é possível escrever *stored procedures* em linguagens de uso geral, como Java e outras.

Referências

Gray, J. (2004). The next database revolution. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 1–4. ACM Press.

Ramakrishnan, R. and Gehrke, J. (2003). *Database Management Systems*. McGraw-Hill Higher Education, third edition.