Banco de Dados – Atividade Hands-on em Laboratório - 08
HSQLDB - Driver JDBC - Prof. Dr. Aparecido Freitas

1. Introdução

Até o momento trabalhamos com aplicações que fazem uma única conexão ao banco de dados.

Mas o que acontece em aplicações web ou cliente servidor onde temos dezenas, centenas ou até mesmo milhares de usuários acessando o serviço simultâneamente?

Se mantivermos uma única conexão o tempo todo, assim que o primeiro usuário acessa e enquanto ele executa suas queries, o segundo usuário espera.

Se um terceiro usuário fizer uma requisição ele terá que esperar o término das requisições do primeiro e do segundo usuário.

Dessa maneira vamos enfileirando todas as requisições com uma única conexão para atendê-las. É similar a uma fila do banco: diversas pessoas vão entrando na fila e só existe um único caixa para atender, uma única conexão para atender. Enquanto o primeiro não termina, todos esperam.

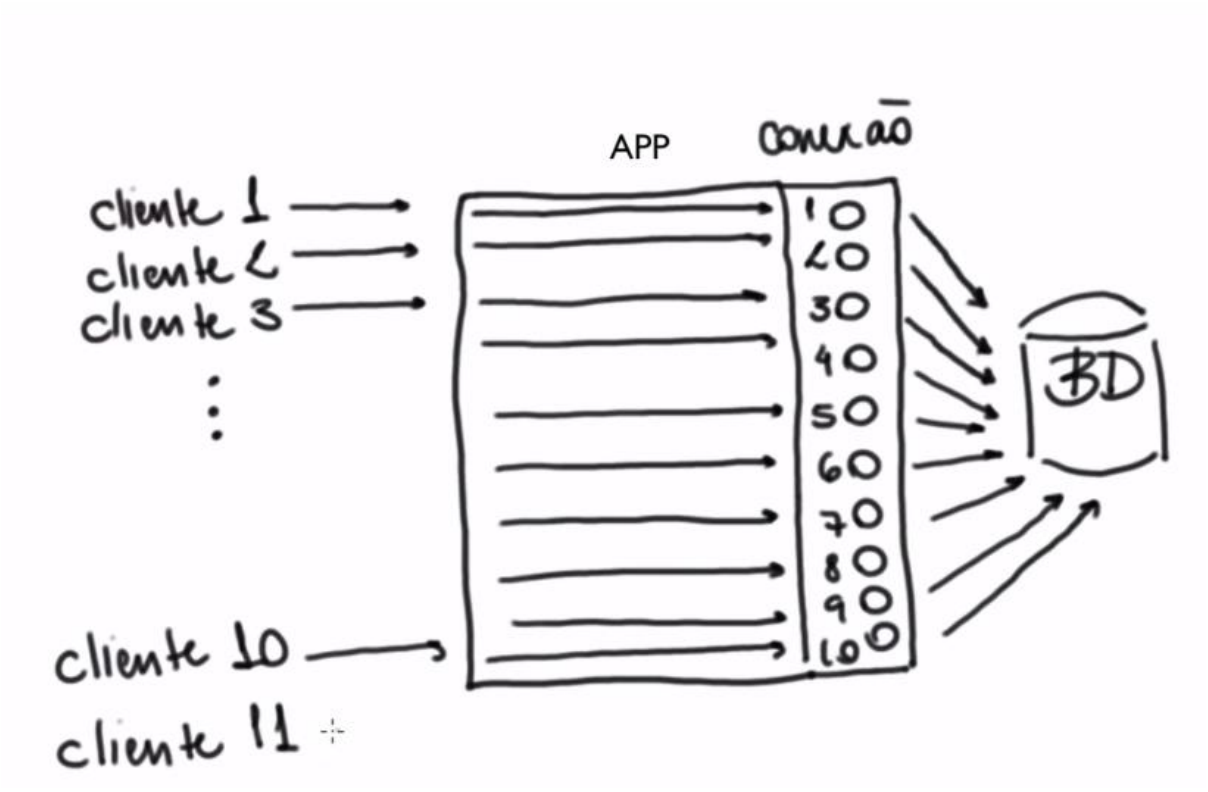
Claramente, em geral, essa não é uma boa solução. Mas e se abrirmos e fecharmos uma nova conexão para cada novo usuário/requisição?

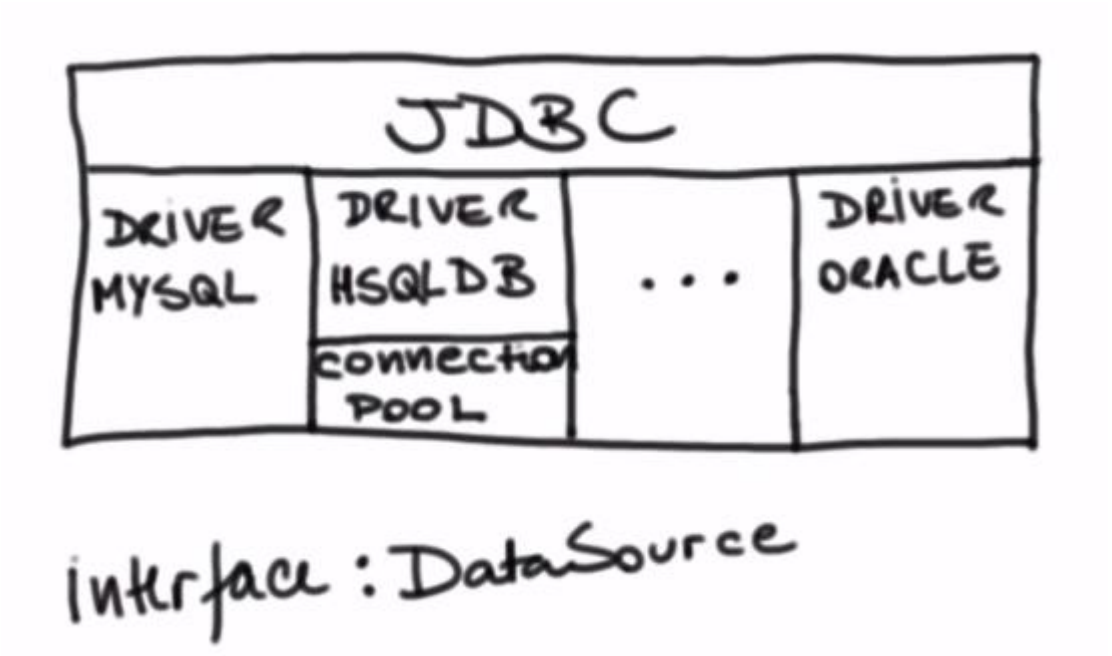
O custo de abrir e fechar uma conexão é alto: é necessário enviar e receber dados de autenticação via TCP para o banco que é remoto. Além disso, se o número de usuários cresce muito, não teremos conexões suficientes no banco para responder pelo desejo dos usuários. O mesmo acontece na fila do banco: criar novos caixas e fechá-los a todo instante que chega um novo cliente é muito custoso, mas seria ótimo, claro.

Que tal o meio termo? Mantemos 5 conexões abertas o tempo todo e a medida que chegam usuários, eles acessam essas 5 conexões. Quando eles vão liberando, o próximo da fila entra e usa a conexão disponível. Desta maneira somos justos (todos serão atendidos na ordem que chegaram), disponibilizamos um número maior de recursos/conexões (não somente uma) mas também não sobrecarregamos com o número de conexões.

Essa abordagem, onde disponibilizamos um número limitado de objetos (conexões), é chamado de **pool**. Temos um pool de conexões, de onde virão os objetos para atender os requerimentos de cada cliente. À medida que os clientes terminam sua requisição, os objetos são devolvidos para esse pool.

Poderíamos implementar nosso pool de conexões mas os drivers do **JDBC** já implementam isso para nós (além de diversas bibliotecas Java), através da interface **DataSource**.

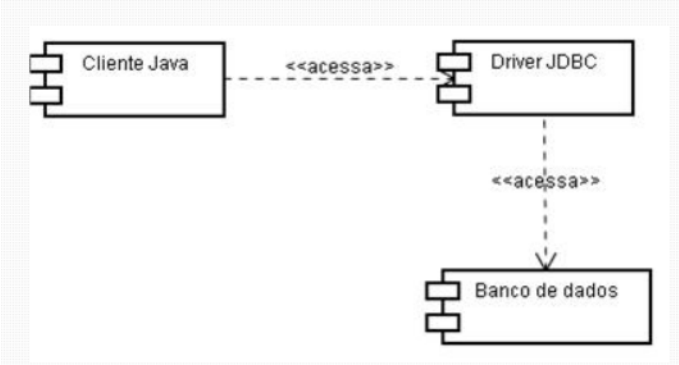




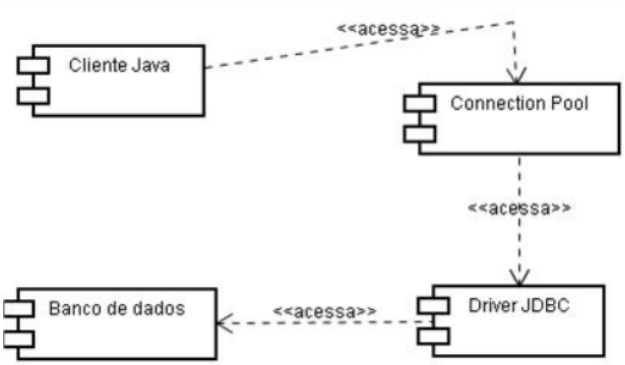
Qualquer aplicação web que acesse bancos de dados precisa estar preparada para receber vários acessos simultâneos de usuários acessando a página e ainda assim acessar o banco de dados usando uma conexão com o banco. Mas o que acontece quando o número de usuário é muito grande? Cada vez que uma requisição é feita, o servidor deve abrir uma conexão com o banco de dados e fechá-la no final da requisição? Além do tempo de latência ser grande ao ficar abrindo e fechando conexões com o banco, deixando o sistema mais lento, isso simplesmente pode deixar a aplicação inutilizável no caso de um número grande de requisições.

Um **connection pool** significaria "piscina de conexões" em português. Basicamente, é uma camada que fica entre o cliente de banco de dados, que faz as conexões com o banco, e o próprio banco. Em aplicações Java, o cliente normalmente é um Servlet ou uma classe Java qualquer e o banco seria representado por uma conexão com o driver **JDBC** para algum servidor de banco de dados.

A idéia dessa camada intermediária é que o cliente possa criar conexões com o banco usando o **connection pool** quase da mesma forma que criaria usando **JDBC** diretamente, de modo que fica transparente para ele como a **conexão** é retornada. O que importa é conseguir uma conexão com o banco de dados para poder realizar as operações desejadas.



Cliente acessando um Banco de dados sem o Connection Pool



Cliente acessando um Banco de Dados com o Connection Pool

Mas o que faz o **pool**? Basicamente, ele **mantém certo número de conexões abertas** com o banco de dados. Quando o cliente **Java** abre uma conexão usando o pool, ao invés de abrir uma nova conexão com o banco usando o driver **JDBC**, este simplesmente pega uma das conexões que ele já mantinha aberta com o banco e a marca como alocada para aquele cliente **Java**.

O cliente Java então usa a conexão normalmente e faz as operações desejadas no banco. Quando o cliente fecha a conexão usando o pool, este não fecha a conexão com o banco. Ao invés disso, mantém a mesma aberta, mas a marca como disponível.

Quando o número de conexões que os clientes abrem usando o **pool** passa do número de conexões que o **connection pool** mantém abertas, o **pool** abre uma nova conexão com o banco de dados, a não ser que tenha atingido um número máximo de conexões reais, caso no qual seria lançada uma exceção.

Esse mecanismo tem duas vantagens chave:

1. Evita a necessidade de ficar abrindo e fechando conexões reais com o banco de dados toda hora, evitando assim o custo ocasionado pelo tempo de abrir e fechar as conexões, tornando o aplicativo razoavelmente mais rápido.

2. Permite servir um número grande de requisições sem necessidade de manter um número tão grande de conexões com o banco. O número de conexões do pool tem que ser tão grande quanto o número de conexões simultâneas.

Qualquer aplicação **JEE** bem feita deve usar pools de conexão. Perceba que, apesar do conceito estar sendo aplicado para conexões com o banco de dados, poderia perfeitamente ser aplicado para outros tipos de conexão, como acessos a EJBs, servlets, etc.

Vamos criar uma classe chamada **Database** que irá implementar a funcionalidade de um pool de conexões.

```
public class Database {

    private DataSource dataSource;

    Database() {
        JDBCPool pool = new JDBCPool();
        pool.setUrl("jdbc:hsqldb:hsql://localhost:59999/db");
        pool.setUser("SA");
        pool.setPassword("");
        this.dataSource = pool;
    }
    Connection getConnection() throws SQLException {
        Connection connection = dataSource.getConnection();
        return connection;
    }

}
```

Nessa implementação utilizamos a interface DataSource. Vamos agora testar essa implementação, construindo uma classe chamada **TestaListagem**.

```
package br.maua;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class TestaListagem {

	public static void main(String[] args) throws SQLException {

		Connection connection = new Database().getConnection();

		Statement statement = connection.createStatement();
		boolean resultado = statement.execute("select * from Produtos");
		ResultSet resultSet = statement.getResultSet();

		while (resultSet.next()) {

			int id = resultSet.getInt("id");
			String nome = resultSet.getString("nome");
			String descricao = resultSet.getString("descricao");
			System.out.println("id=" + id + ", nome=" + nome + ", descricao=" +
descricao);
		}

		resultSet.close();
		statement.close();
		connection.close();
	}


}
```