HOLOS
ISSN 1807 - 1600

UM ESTUDO COMPARATIVO ENTRE BANCO DE DADOS ORIENTADO A OBJETOS, BANCO DE DADOS RELACIONAIS E FRAMEWORK PARA MAPEAMENTO OBJETO/RELACIONAL, NO CONTEXTO DE UMA APLICAÇÃO WEB

M. M. A. OLIVEIRA[*], D. G. CARLOS, A. R. V. O. SOUSA e A. F. CASTRO
Universidade Federal Rural do Semi-Árido
marisergio@gmail.com[*]

Artigo submetido em novembro/2012 e aceito em fevereiro/2015

DOI: 10.15628/holos.2015.1153

RESUMO

A escolha da técnica de persistência de dados é um fator importante para o êxito de um sistema computacional, estando diretamente ligada a quesitos como integridade, confiabilidade, facilidade de manutenção e, sobretudo, o desempenho. O presente artigo realiza um estudo comparativo entre o uso de banco de dados relacional (BDR), banco de dados relacional com o uso de frameworks para mapeamento objeto/relacional e banco de dados orientado a objeto (BDOO), tendo como objetivo principal comparar o desempenho dos SGBDs no contexto de aplicações web. Para esta investigação foram utilizados: o MySQL como BDR, Hibernate como framework, DB4o como BDOO, Glassfish como servidor web e Java como plataforma.

PALAVRAS-CHAVE: **Banco de dados, Framework, Orientação a Objetos.**

A COMPARATIVE STUDY OF OBJECT-ORIENTED DATABASE, RELATIONAL DATABASE AND FRAMEWORK FOR MAPPING OBJECT-RELATIONAL, IN THE CONTEXT TO A WEB APPLICATION

ABSTRACT

The choice of technique of persistence of data is a factor important for the success of a system computational, being directly connected the questions like integrity, confiability, facility of maintenance and, especially, the performance. The present article performs a study comparative between the use of relational database (BDR),relational database with use of frameworks for mapping object/relational and object-oriented database (BDOO), having like main goal compare the performance of SGBDs in the context of applications web. For this investigation were used: MySQL like BDR, Hibernate like framework, DB4o like BDOO, Glassfish like server web and Java like platform.

KEYWORDS: **Database, Framework, Object Orientation.**

HOLOS
ISSN 1807 - 1600

1 INTRODUÇÃO

O enorme volume de informações em que o mundo computacional, nos dias atuais, está inserido suscita uma atenção na escolha de uma técnica de persistência de dados adequada, pois a mesma é um fator importante para o êxito de um sistema computacional, estando diretamente ligada a quesitos como integridade, confiabilidade, facilidade de manutenção e, sobretudo, o desempenho. As vantagens oferecidas por cada uma das técnicas de persistências disponíveis e a necessidade da aplicação devem ser levadas em consideração no momento da definição do projeto. É importante também atentar para as tendências, pois, à medida que o tempo passa, esse volume de informações deve aumentar e, associado a este fato, está uma maior exigência dos usuários, principalmente no tocante à velocidade do tráfego dos dados. Nesse sentido, foram investigadas as seguintes técnicas de persistência: banco de dados relacional, banco de dados orientado a objetos e banco de dados relacional utilizando um framework para o mapeamento objeto/relacional.

Diante do contexto mencionado, a modelagem de dados se tornou mais complexa, fazendo com que, em alguns casos, a base do modelo relacional se tornasse inadequada. Assim, as limitações desse modelo provocam algumas deficiências quando o mesmo é utilizado em aplicações mais complexas. Como solução para esses obstáculos, modelos mais novos estão sendo empregados nessas aplicações. Atualmente a orientação a objetos é o paradigma mais utilizado, devido à facilidade de manutenção e o maior aproveitamento de códigos, além de apresentar vantagens na modelagem de dados mais complexos. Em contrapartida, o mapeamento entre os dois modelos é uma atividade que consome muito tempo, diminuindo a produtividade (BOSCARIOLI et al., 2008).

O objetivo desse artigo é apresentar um estudo comparativo de desempenho entre algumas técnicas de persistências de dados, no contexto de aplicações web, estudo este influenciado principalmente pelas dificuldades de se trabalhar com paradigmas diferentes para a aplicação e para a base de dados, mas também pela intrigante promessa das vantagens oferecidas pelos Bancos de Dados Orientados a Objetos. Para esta investigação foi utilizado o MySQL como Banco de Dados Relacional (BDR), Hibernate como framework, DB4o como Banco de Dados Orientado a Objetos (BDOO), Glassfish como servidor web e Java como plataforma. Vale salientar que estes produtos foram escolhidos por serem difundidos e possuírem características *open source*.

O dilema está na decisão de: utilizar frameworks que proporcionam facilidades no mapeamento e portabilidade entre SGBD's; utilizar banco de dados orientado a objeto, pois os mesmos prometem facilidade e agilidade, principalmente pelo fato de estarem no mesmo paradigma da aplicação; ou, ainda, garantir manualmente todas as prerrogativas do bom funcionamento de um banco de dados relacional, fundamentado em integridade e confiabilidade.

Este artigo foi estruturado da seguinte forma: a Seção 2 apresenta as técnicas de persistência de dados que foram abordadas nesta pesquisa; a Seção 3 diz respeito às modelagens, ambiente de teste, implementações, soluções para os três casos do problema

HOLOS
ISSN 1807 - 1600

escolhido para o estudo e os resultados obtidos, e, por fim, a Seção 4 apresenta as considerações finais e uma proposta para trabalhos futuros.

2 TÉCNICAS DE PERSISTÊNCIA DE DADOS EM APLICAÇÕES

2.1 Banco de Dados

Um Banco de Dados pode ser definido como um conjunto de dados devidamente relacionados (MACHADO, 2008). Os Sistemas de Banco de Dados (SGBD) surgiram com o intuito de contornar os problemas apresentados no formato de armazenamento em arquivos de sistema operacional e são formados por uma série de componentes no nível de hardware e de software, dentre os quais se destaca o SGBD, que permite o gerenciamento de informações armazenadas em um Banco de Dados (DATE, 2003). Os SGBDs devem possuir uma série de características para evitar alguns problemas, tais como redundância e inconsistência dos dados, acesso indevido e problemas de atomicidade.

Dentre vários tipos de banco de dados, abaixo veremos os conceitos dos modelos aplicados nesse estudo.

2.1.1 Banco de Dados Relacional

Os sistemas de Banco de Dados Relacional (BDR) baseiam-se no modelo relacional conceituado por Codd (1970). Sua estrutura é formada por tabelas para representar dados e suas relações. Possui uma coleção de operadores, que constituem a base da linguagem SQL, e uma coleção de restrições de integridade, que definem um conjunto consistente de estados de base de dados e de alterações de estados (SILBERSCHATZ; KORTH; SUDARSHAN, 2006).

2.1.2 Banco de Dados Orientado a Objetos

Os sistemas de Banco de Dados Orientado a Objetos (BDOO) permitem acesso direto aos dados de uma linguagem de programação orientada a objetos usando o sistema de tipo nativo da linguagem (SILBERSCHATZ; KORTH; SUDARSHAN, 2006).

2.2 Framework

Segundo Fayad e Schimidt (1997), um Framework é um conjunto de classes que colaboram para realizar uma responsabilidade para um domínio de um subsistema da aplicação. Um framework consiste em utilizar várias funcionalidades de um software em um conjunto de aplicações com problemas pertencentes a um mesmo domínio. Diferentemente das Bibliotecas, os Frameworks possuem como característica controlar o fluxo da aplicação.

O fortalecimento da orientação a objetos foi de grande importância para o desenvolvimento dos frameworks, sendo os mesmos bastante utilizados atualmente pelos profissionais de TI.

HOLOS
ISSN 1807 - 1600

3 ESTUDO DE CASO

3.1 Ambiente de Teste

Para o teste de desempenho foi utilizado o seguinte ambiente:

3.1.1 Tecnologias Utilizadas

O MySQL foi escolhido como SGBD baseado no Modelo Relacional. O mesmo utiliza a linguagem SQL e, mesmo diante de um constante avanço em tecnologias que utilizam outros paradigmas, é um dos Bancos de Dados mais populares devido à sua facilidade de uso e confiabilidade (MYSQL, 2012).

Já para a implementação da solução utilizando BDOO foi usado o DB4o, que é um banco de dados inteiramente orientado a objetos, podendo ser utilizado em aplicações do tipo embarcada, cliente-servidor e desktop. Este BDOO permite armazenar classes Java diretamente no banco, sem precisar utilizar consultas SQL. O DB4o possui algumas vantagens em relação aos bancos relacionais: a ferramenta é nativa em Java (ou .Net), utiliza pouco recurso computacional, tem fácil aprendizado e não possui nenhuma linha de código SQL para CRUD - Create, Read, Update e Delete (DB4Objects, 2012).

O Hibernate foi utilizado como framework para facilitar o mapeamento dos atributos entre uma base tradicional de dados relacionais e o modelo objeto de uma aplicação. Sua principal característica é a transformação das classes em Java para tabelas de dados. O Hibernate gera as chamadas SQL e libera o desenvolvedor do trabalho manual da conversão dos dados resultante, mantendo o programa portável para quaisquer bancos de dados SQL (HIBERNATE, 2012).

Além das tecnologias citadas acima, foi utilizado um servidor web que pode ser definido como um programa de computador que recebe uma requisição HTTP e retorna uma resposta HTTP. Para o estudo de caso optou-se pelo GlassFish, um servidor de aplicação *open source* para a plataforma Java EE. O GlassFish possui suporte para todas as especificações da API Java EE, inclusive a JDBC, especificação mais importante para esta pesquisa (GLASSFISH, 2012).

Logo abaixo seguem algumas informações importantes sobre as tecnologias aqui mencionadas:

- MySQL: Site oficial http://www.mysql.com
  - Licença: GNU General Public License;
  - Versão: 2 (GPLv2);
- DB4o: Site oficial http://www.db4o.com
  - Licença: GNU GENERAL PUBLIC LICENSE;
  - Versão: 3;
- Hibernate: Site oficial http://www.hibernate.org
  - Licença - GNU LESSER GENERAL PUBLIC LICENSE;
  - Versão: 2.1;
- Máquina Virtual Java: Site oficial http://www.oracle.com/technetwork/java/javase/downloads/
- JDBC - Mysql Connector: Site oficial http://dev.mysql.com/downloads/connector/j/

HOLOS ISSN 1807 - 1600

- Licença: GNU GENERAL PUBLIC LICENSE;
- Versão: 2;

- Glassfish: Site oficial http://www.oracle.com/technetwork/java/javaee/downloads/ogs-3-1-1-downloads-439803.html
  - Licença: Oracle Technology Network Developer License Terms;

3.1.2 Hardware Utilizado

- Notebook Acer®;
- Processador Intel® Pentium P6200 2,13Ghz;
- 1.74GB memória RAM;
- Windows 7® Professional SP1;

3.2 Descrição da Aplicação

3.2.1 Modelagem

Objetivando uma simulação que represente parte do ônus de conversão entre os modelos investigados, foi escolhido um problema que envolve o conceito de herança do paradigma de orientação a objetos. O mesmo é de simples solução, trata-se de um "Cadastro de Alunos", considerando-se apenas duas classes Java, Aluno e Pessoa, sendo subclasse e superclasse respectivamente. A figura 1 representa o modelo das classes mencionadas.

Figura 1: Diagrama de Classes do problema supracitado.

3.2.2 Persistência

A figura 2 exibe o diagrama entidade relacionamento do problema acima citado e, logo após, são apresentados os códigos-fontes das classes para cada uma das técnicas escolhidas.

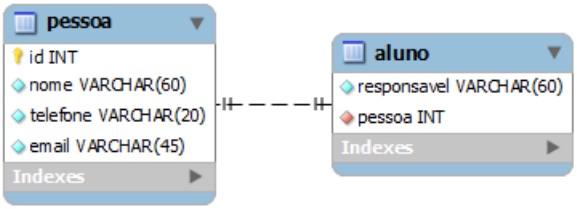

Figura 2: Diagrama Entidade Relacional do problema supracitado.

3.2.2.1. DB4o

Na tabela 1, o código descreve duas classes persistentes no DB4o; as mesmas diferem um pouco de um JavaBean, ou seja, uma classe contendo atributos e seus respectivos métodos get's e set's. Neste caso é possível otimizar consultas criando índices com a anotação *@Indexed*. Uma das vantagens do DB4o é a capacidade que o mesmo possui em utilizar todos os benefícios da

HOLOS ISSN 1807 - 1600

orientação a objetos de forma natural, como a herança mostrada com a palavra-chave *extends* do Java.

Tabela 1: Definição das classes modelo para o DB4o.

```
import com.db4o.config.annotations.Indexed;

public abstract class Pessoa {

    @Indexed
    private String nome;
    private String telefone;
    private String email;

    public String getNome() {
        return nome; }

    public void setNome(String nome) {
        this.nome = nome;}

    public String getTelefone() {
        return telefone;
    }

    public void setTelefone(String telefone) {
        this.telefone = telefone;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}
```

```
public class Aluno extends Pessoa{

    private String responsavel;

    public String getResponsavel() {
        return responsavel;
    }

    public void setResponsavel(String responsavel) {
        this.responsavel = responsavel;
    }

}
```

3.2.2.2. Hibernate

No Hibernate, o uso de anotações do JPA (*Java Persistence API*) é uma das formas de realizar o mapeamento objeto-relacional. Na tabela 2, a anotação *@Entity* é utilizada para mapear entidades no banco de dados; já a anotação *@Id* define uma chave primária, sendo a mesma, item obrigatório em uma classe persistente do Hibernate. Também é possível gerar automaticamente o valor para este campo com a anotação *@GeneratedValue*. Assim como no DB4o, pode-se gerar índices através da anotação *@Index*. A herança, neste caso, pode ser feita com o emprego da anotação *@Inheritance(strategy=InheritanceType.JOINED)*, onde é possível definir uma estratégia de mapeamento das entidades no banco de dados. Para o estudo de caso definido, foi necessária a criação de duas tabelas similares às classes em questão, através do valor *InheritanceType.JOINED* passado para o parâmetro *strategy*. É importante ressaltar que há outras estratégias para realizar a herança com o Hibernate.

Tabela 2: Definição das classes modelo para o Hibernate.

```
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
```

```
import javax.persistence.Entity;
import javax.persistence.NamedQuery;
```

```
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;

import org.hibernate.annotations.Index;


@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public abstract class Pessoa
    @Id
    @Index(name="index_pessoa_id")
    @GeneratedValue
    private Long id;

    @Index(name="index_pessoa_nome")
    @Column(unique=true)
    private String nome;

    private String telefone;

    private String email;

    // set's e get's ...
}
```

```
@Entity
public class Aluno extends Pessoa{

    private String responsavel;


// set's e get's ...

}
```

3.2.2.3. MySQL/JDBC

A tabela 3 representa as classes a serem persistidas no SGBDR MySQL. As definições do banco de dados, como, por exemplo, índices e chaves primárias, são feitas diretamente no próprio SGBD. A tabela 4 exibe essas definições do SQL para o MySQL.

Tabela 3: Definição das classes modelo para o MySQL/JDBC.

```
public abstract class Pessoa {
    private Long id;
    private String nome;
    private String telefone;
    private String email;
// set's e get's ...
}
```

```
public class Aluno extends Pessoa{

    private String responsavel;

// set's e get's ...

}
```

Tabela 4: Definição do SQL para o MySQL.

```
CREATE TABLE IF NOT EXISTS 'aluno' (
'responsavel' varchar(255) DEFAULT NULL,
'id' bigint(20) NOT NULL,
  PRIMARY KEY ('id'),
  KEY 'aluno_id' ('id'),
  KEY 'id' ('id')
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

CREATE TABLE IF NOT EXISTS 'pessoa' (
  'id' bigint(20) NOT NULL AUTO_INCREMENT,
  'email' varchar(255) DEFAULT NULL,
  'nome' varchar(255) DEFAULT NULL,
  'telefone' varchar(25) DEFAULT NULL,
  PRIMARY KEY ('id'),
UNIQUE KEY 'nome' ('nome'),
```

```
  KEY 'index_pessoa_nome' ('nome'),
KEY 'index_pessoa_id' ('id')
) ENGINE=InnoDB;

ALTER TABLE 'aluno'
  ADD CONSTRAINT 'aluno_ibfk_1' FOREIGN KEY ('id') REFERENCES
'pessoa' ('id`) ON DELETE NO ACTION ON UPDATE NO ACTION;
```

3.2.3 Conexão

Por se tratar de uma aplicação web, atendendo a recomendações da comunidade de desenvolvedores Java, foi feito o emprego de um pool de conexões para as implementações com o Hibernate e o JDBC, gerenciadas pelo servidor web Glassfish.

Para tornar a comunicação com o banco de dados mais eficiente, um pool de conexões é criado com o objetivo de tornar as conexões reutilizáveis. Em um ambiente JAVA EE, é possível utilizar pools de conexões gerenciados pelo servidor web, assim o desenvolvedor não precisa se preocupar com o ciclo de vida das conexões realizadas.

A figura 3 ilustra como as conexões são gerenciadas por um servidor de aplicação web, onde é possível a existência de vários clientes requisitando conexões simultaneamente. Como há várias conexões abertas dentro do pool de conexões, quando um cliente precisar realizar uma operação com o banco de dados não será necessário abrir uma nova conexão, o que economiza tempo, pois as operações de abrir e fechar conexão com o banco de dados são bastante onerosas em termos de tempo de processamento. No caso de todas as conexões abertas estarem ocupadas, seria preciso que se abrisse uma nova conexão, o que levaria o mesmo tempo de uma aplicação que não utiliza um pool de conexão. A explicação detalhada de como configurar um pool de conexões no Glassfish está fora do âmbito desse artigo.

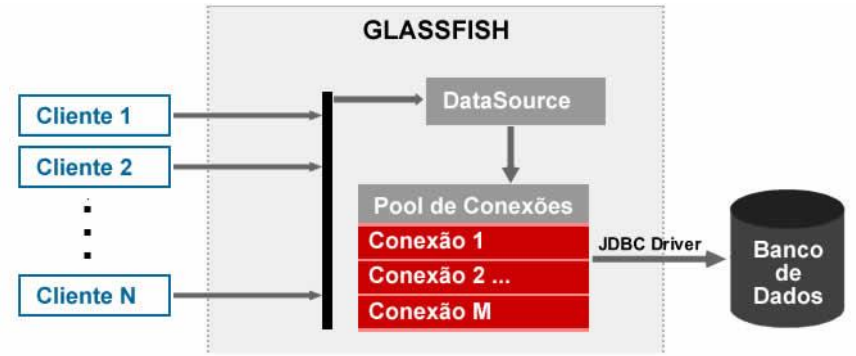

Figura 3: Arquitetura de um Pool de Conexões gerenciada pelo Glassfish.

3.2.3.1. DB4o

A tabela 5 descreve a classe que configura as definições de conexão com o DB4o. A conexão do DB4o deve ser feita utilizando um objeto da classe *ObjectContainer*, que pode receber como parâmetro o endereço do servidor no qual ficará armazenada a base de dados.

Tabela 5: Definição da classe de conexão para o DB4o.

```
import com.db4o.Db4oEmbedded;
import com.db4o.ObjectContainer;

public class ConnectionFactory {
```

HOLOS ISSN 1807 - 1600

```
    privates tatic ObjectContainer connection ;

public static ObjectContainergetConnection(){

    if (connection == null){
    connection =  Db4oEmbedded.openFile(System.getProperty("user.dir")+"/soae.db4o");
      }
    return connection;
    }

    public static void close() {
          if(connection != null){
                connection.close();
          }

    }
}
```

3.2.3.2. Hibernate

Para configurar o Hibernate em uma aplicação, um arquivo chamado *persistence.xml* deve ser criado dentro de um diretório chamado META-INF, que deve estar no *classpath* da aplicação. Na tabela 6, logo abaixo, segue a configuração desse arquivo para este estudo.

Tabela 6: Definição do arquivo persistence.xml.

```
<?xml version="1.0"encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
          version="1.0">

          <persistence-unit name="context_hibernate" transaction-type="JTA">
                <provider>org.hibernate.ejb.HibernatePersistence</provider>
                <jta-data-source>jdbc_hibernate_artigo</jta-data-source>
          </persistence-unit>
</persistence>
```

Neste arquivo *persistence.xml* deve-se utilizar a tag*<persistence-unit>* para definir uma unidade de persistência, onde deve se atribuir um valor à propriedade *name* para que seja possível referenciá-la na aplicação. A propriedade *transaction-type* define o tipo de transação RESOURCE_LOCAL para aplicações Java SE ou JTA quando se acessa um pool de conexões em um servidor web.

O trecho de código *<provider>org.hibernate.ejb.HibernatePersistence</provider>* define que a implementação da especificação JPA utilizada será o Hibernate.

Para configurar um pool de conexões no próprio servidor de aplicação web, ao invés de obter uma conexão diretamente a partir do drive JDBC, deve-se criar um *DataSource* no servidor e configurá-lo. O trecho de código *<jta-data-source>jdbc_hibernate_artigo</jta-data-source>* faz uma chamada ao *DataSource* previamente configurado para prover as conexões com o SGBD.

3.2.3.3. JDBC para o MySQL

HOLOS ISSN 1807 - 1600

A tabela 7 descreve a classe que configura as definições de conexão com o JDBC (*Java DatabaseConnectivity*). Para isso, deve ser instanciado um objeto da classe *Connection*. Semelhante ao Hibernate, foi utilizado o pool de conexões, onde o mesmo disponibilizará as conexões necessárias para o bom funcionamento da aplicação. Para gerenciar o pool de conexões é necessário que seja criado um *DataSource*, conforme especificado no seguinte trecho de código:

ctx = new InitialContext();
DataSource ds = (DataSource) ctx.lookup("jdbc_artigo");

Tabela 7: Definição da classe de conexão para o JDBC/MySQL.

```
import java.sql.Connection;
import java.sql.SQLException;
import javax.sql.DataSource;

public class ConnectionFactory {
        private static Connection connection ;

        public static Connection getConnection(){
                if (connection == null){
                        Context ctx;
                        try {     ctx = new InitialContext();
                                DataSource ds;
                                ds = (DataSource) ctx.lookup("jdbc_artigo");
                                try { connection = ds.getConnection();
                                } catch (SQLException e) {
                                        e.printStackTrace();  }
                        } catch (NamingException e) {
                                // TODO Auto-generated catch block
                                e.printStackTrace(); }
                }
                return connection;
        }
        public static void close(){
                if(connection != null){
                        try {
                                connection.close();
                        } catch (SQLException e) {
                                e.printStackTrace();
} } } }
```

3.2.4 CRUD (Create, Read, Update e Delete)

3.2.4.1. DB4o

A tabela 8 descreve a classe CRUD usada nesta investigação para o DB4o, onde são implementados os métodos de acesso à base de dados, na qual isso ocorre através de uma instância da conexão configurada no subitem 3.2.3.1.

Tabela 8: Definição da classe CRUD para o DB4o.

```
import com.db4o.ObjectContainer;
import com.db4o.query.Predicate;

public class AlunoDAO {
        private ObjectContainer db = ConnectionFactory.getConnection();
```

```
	public void save(Aluno aluno) {
		try {
			this.db.store(aluno);
			this.db.commit();
		} catch (Exception e) {
			e.printStackTrace();
			this.db.rollback();
		}
	}
	public void delete(Aluno aluno) {
		try {
			this.db.delete(aluno);
			this.db.commit();
		} catch (Exception e) {
			e.printStackTrace();
			this.db.rollback();
		}
	}

	public List<Aluno> listar() {
		List<Aluno> alunos = new ArrayList<Aluno>();
		try {
			alunos = this.db.query(Aluno.class);
		} catch (Exception e) {
			e.printStackTrace();
		}

		return alunos;
	}
}
```

3.2.4.2. HIBERNATE

A classe CRUD para o Hibernate é definida na tabela 9. Assim como no DB4o, os métodos de acesso à base de dados são implementados e o acesso ao banco de dados se dá por meio de uma instância da conexão apresentada no subitem 3.2.3.2. É importante destacar a anotação *@PersistenceContext(unitName="context_hibernate")*, pois a mesma referencia a configuração da conexão estabelecida no arquivo *persistence.xml.*

Tabela 9: Definição da classe CRUD para o Hibernate.

```
import javax.ejb.Stateless;
import javax.ejb.TransactionAttribute;
import javax.ejb.TransactionAttributeType;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.TypedQuery;

public class AlunoDAO {

 @PersistenceContext(unitName="context_hibernate")
 private EntityManager manager;

 public void save(Alunoaluno) {
	try {
		manager.persist(aluno);
	} catch (Exception e) {
```

```
                System.out.println(aluno.getEmail());
        }
 }
 public void update(Aluno aluno) {
        try {
                manager.merge(aluno);
        } catch (Exception e) {
                e.printStackTrace();
        }
 }
 public void delete(Long id) {
        try {
                AlunoobjectTemp = manager.find(Aluno.class, id);
                if(objectTemp != null){ manager.remove(objectTemp);  }

        } catch (Exception e) {
                e.printStackTrace();
        } }
 public List<Aluno> listar() {
        List<Aluno> alunos = new ArrayList<Aluno>();
        try {
                TypedQuery<Aluno>     query     =     manager.createNamedQuery("aluno.listar",
Aluno.class);
                return (ArrayList<Aluno>) query.getResultList();
        } catch (Exception e) {
                e.printStackTrace();
        }
        return alunos;
 }
```

3.2.4.3. MySQL

Assim como nas implementações anteriores, a classe definida na tabela 10 descreve os métodos de acesso ao SGBD e a comunicação é feita através de uma instância da classe descrita no subitem 3.2.3.3.

Tabela 10: Definição da classe CRUD para o JDBC/MySQL.

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

public class AlunoDAO {
  private Connection connection = ConnectionFactory.getConnection();

  public void save(Aluno aluno) {
        String sql = "insert into pessoa" +"(nome,email, telefone)"+ " values (?,?,?)";
        try {
                PreparedStatementstmt                =                connection.prepareStatement(sql,
Statement.RETURN_GENERATED_KEYS);
                stmt.setString(1, aluno.getNome());
                stmt.setString(2, aluno.getEmail());
                stmt.setString(3, aluno.getTelefone());
                stmt.executeUpdate();
                ResultSetrs = stmt.getGeneratedKeys();
                Long id = null;
                if(rs.next()){
```

```
                        id = rs.getLong(1);
                }
        stmt = connection.prepareStatement("insert into aluno (id, responsavel) values (?, ?)");
                stmt.setLong(1, id);
                stmt.setString(2, aluno.getResponsavel());
                stmt.executeUpdate();
                stmt.close();
        } catch (SQLException e) {
                e.printStackTrace();
        }
}

public void update (Alunoaluno) {
        try {
                PreparedStatementstmt = connection
                .prepareStatement("update pessoa set nome=?, email=?, telefone=? where id=?");
                stmt.setString(1, aluno.getNome());
                stmt.setString(2, aluno.getEmail());
                stmt.setString(3, aluno.getTelefone());
                stmt.setLong(4, aluno.getId());
                stmt.executeUpdate();
                stmt = connection
                                .prepareStatement("update aluno set responsavel=? where id=?");
                stmt.setString(1, aluno.getResponsavel());
                stmt.setLong(2, aluno.getId());
                stmt.executeUpdate();
                stmt.close();
        } catch (SQLException e) {
                e.printStackTrace();
        }
}

public void delete(Long id) {
        try {
                PreparedStatementstmt = connection
                                .prepareStatement("delete from aluno where id=?");
                stmt.setLong(1, id);
                stmt.executeUpdate();
                stmt = connection.prepareStatement("delete from pessoa where id=?");
                stmt.setLong(1, id);
                stmt.executeUpdate();
                stmt.close();
        } catch (SQLException e) {
                e.printStackTrace();
        }
}
public List<Aluno>listar() {
        try {
                List<Aluno> alunos = new ArrayList<Aluno>();
                PreparedStatementstmt = this.connection
                                .prepareStatement("select p.id, p.nome, p.telefone, p.email,
a.responsavel from aluno as a, pessoa as p WHERE p.id=a.id");
                ResultSetrs = stmt.executeQuery();
                while (rs.next()) {
                        Aluno aluno = new Aluno();
                        aluno.setId(rs.getLong("id"));
                        aluno.setNome(rs.getString("nome"));
                        aluno.setEmail(rs.getString("email"));
                        aluno.setTelefone(rs.getString("telefone"));
                        aluno.setResponsavel(rs.getString("responsavel"));
```

HOLOS ISSN 1807 - 1600

```
                        alunos.add(aluno);
                }
                rs.close();

                stmt.close();

                return alunos;
        } catch (SQLException e) {
                throw new RuntimeException(e);
        }
 }
}
```

3.3 Desempenho

O teste abaixo objetiva, de forma simples, demonstrar os resultados de desempenho dos modelos de persistência de dados.

3.3.1 Metodologia

1. Para cada técnica de persistência foi utilizado um aplicativo de teste com a seguinte descrição: definição de instâncias dos objetos descritos nos códigos-fontes demonstrados ao longo deste artigo e implementação de uma rotina que realiza automaticamente operações de inserção, deleção, alteração e consulta repetidamente por 1000 vezes cada uma, obtendo-se o tempo médio para cada operação em cada caso;
2. Repetição da rotina descrita acima por três vezes, e obtenção do tempo médio de cada uma das três técnicas de persistência;
3. O tempo de execução para cada operação CRUD e em cada uma das técnicas foi constatado a partir da diferença, em nanossegundos, entre o instante que imediatamente antecedia e o instante posterior à instrução que requisitava o acesso à base de dados.

Apesar de se tratar de uma aplicação web, não foram realizados testes de acesso simultâneo aos SGBDs. Também é importante mencionar a existência de uma considerável falha para essa metodologia, tendo em vista que a mesma faz uso do sistema operacional, o que evidencia a concorrência aos recursos de hardware, ou seja, o tempo de resposta é influenciado pela quantidade e nível de prioridade dos processos que estão sendo executados no momento da realização do teste.

3.3.2 Resultados

Ao observar a figura 4 ou a tabela 11, é notório que o melhor desempenho para as operações de inserção, deleção e atualização ficam com o Hibernate, porém seu desempenho em relação ao MySQL com o JDBC é cerca de três vezes inferior nas operações de consultas. Já em comparação com o DB4o, também em consultas, o mesmo revela ganho considerável.

HOLOS ISSN 1807 - 1600

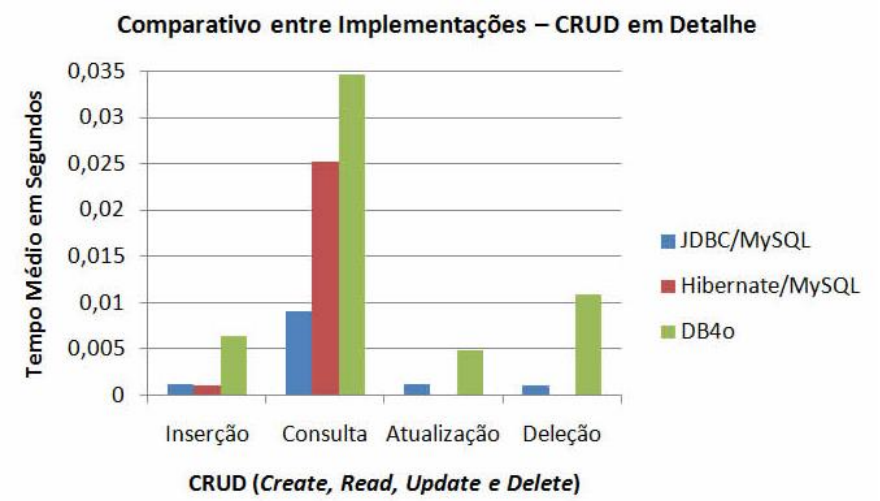

Figura 4: Representação gráfica do comparativo entre implementações – CRUD em detalhe.

Tabela 11: Comparativo entre implementações – CRUD em detalhe.

	Tempo Médio em Segundos / Operação			
SGBD/Framework	Inserção	Consulta	Atualização	Deleção
JDBC/MySQL	$1.13 * 10^{-3}$	$8.99 * 10^{-3}$	$1.13 * 10^{-3}$	$1.06 * 10^{-3}$
Hibernate/MySQL	$1.00 * 10^{-3}$	$2.52 * 10^{-2}$	$3.00 * 10^{-5}$	$3.00 * 10^{-5}$
DB4o	$6.43 * 10^{-3}$	$3.46 * 10^{-2}$	$4.86 * 10^{-3}$	$1.09 * 10^{-2}$

Analisando a figura 5 ou a tabela 12, pode-se constatar que o uso do MySQL com o JDBC se destaca em relação às outras técnicas de persistência aqui estudadas. A escolha pelo pool de conexões com o Glassfish foi de suma importância para tal fato, visto que o mesmo, neste caso, assume a otimização do gerenciamento das conexões. Vale a pena enfatizar o resultado do Hibernate, pois o mesmo, apesar de possuir uma camada extra, teve o seu desempenho em aproximadamente o dobro de tempo do MySQL com o JDBC e surpreendentemente quase três vezes mais rápido que o DB4o, que por sua vez, apesar de dispensar a conversão objeto-relacional, mostrou-se inferior aos demais.

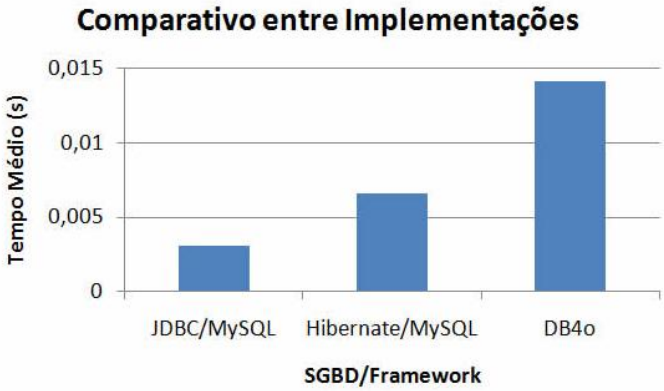

Figura 5: Representação gráfica do comparativo entre Implementações.

Tabela 12: Comparativo entre Implementações.

HOLOS ISSN 1807 - 1600

SGBD/Framework	Tempo Médio (s)
JDBC/MySQL	$3,07 * 10^{-3}$
Hibernate/MySQL	$6,57 * 10^{-3}$
DB4o	$1,42 * 10^{-2}$

É importante salientar que os métodos aplicados neste estudo foram desenvolvidos pelos autores deste artigo. Logo, o mesmo tipo de comparação em contextos diferentes pode resultar em uma variação dos resultados.

3.3.3 Avaliação

Apesar das diferenças no desempenho de cada ferramenta, é importante realizar uma análise no projeto e considerar o que é prioridade. Cada uma das técnicas citadas possui vantagens que podem atender diversos tipos de prioridades. Por exemplo:

- Se a aplicação exige maior complexidade de dados e dispensa um Administrador de Banco de Dados é recomendável utilizar o Modelo Orientado a Objetos;
- Se o desempenho é prioridade, o Modelo Relacional, segundo os resultados obtidos nas tabelas 11 e 12, é o mais recomendado;
- Se o projeto necessita de portabilidade entre SGBDs, o Framework de Mapeamento Objeto/Relacional pode ser uma boa alternativa.

4 CONCLUSÕES

Diante do exposto, pode-se observar que todas as técnicas de persistência de dados aqui estudadas possuem vantagens que se complementam, todavia, em aplicações que necessitam de um recurso específico associado à base de dados, a melhor técnica é aquela que agrega o maior número de benefícios acerca desse recurso, ou seja, o ideal é tentar descobrir qual deve ser empregada em uma determinada situação. Além do desempenho, outros aspectos como recursos, documentação e suporte devem ser considerados.

Como se tratou de uma aplicação web, sugere-se, como trabalho futuro, realizar uma nova investigação fazendo uso de *threads* Java a fim de simular vários usuários simultâneos requisitando conexões com o SGBD.

5 REFERÊNCIAS BIBLIOGRÁFICAS

BOSCARIOLI, Clodis; BEZERRA, Anderson; BENEDICTO, Marcos de; DELMIRO, Gilliard. Uma reflexão sobre Banco de Dados Orientados a Objetos. Disponível em: <http://conged.deinfo.uepg.br/artigo4.pdf>. Acesso em: 14 Out. 2012.

CODD, T. (1970). A Relational Model of Data for Large Shared Data Banks. Communications of the ACM, Vol. 13, No. 6, June 1970.

DATE, C. J. Uma Introdução a Sistemas de Banco de Dados. 8. ed. Traduzido por Daniel Vieira. Rio de Janeiro: Elsevier, 2003.

DB4O. DB4Objects. Disponível em: <http://www.db4o.com/portugues/>. Acesso em: 21 Out.

2012.

FAYAD, Mohamed; SCHIMIDT, Douglas C. Object-oriented application frameworks, New York, ACM, 1997 38p.

Glassfish. Site oficial. Disponível em: <http://glassfish.java.net/>. Acesso em: 07 Out. 2012.

Hibernate. Site oficial. Disponível em: <http://www.hibernate.org/>. Acesso em: 15 Out. 2012.

MACHADO, Felipe Nery Rodrigues. BANCO DE DADOS: Projeto e Implementação. 2. ed. São Paulo: Érica Ltda, 2008.

MySQL. Site oficial. Disponível em: <http://www.mysql.com/>. Acesso em: 18 Out. 2012.

SILBERSCHATZ, Abraham; KORTH, Henry F; SUDARSHAN, S. Sistema de banco de dados. 5. ed. Traduzido por Daniel Vieira. Rio de Janeiro: Elsevier, 2006.