**CENTRO ESTADUAL DE EDUCAÇÃO TECNOLÓGICA PAULA SOUZA**

**Faculdade de Tecnologia da Baixada Santista “Rubens Lara”**

**Curso Superior de Tecnologia em Sistemas para Internet**

**ESTUDO SOBRE PERFORMANCE DE BANCO DE DADOS ORIENTADO A OBJETOS COMPARADO AO MODELO RELACIONAL**

**SANTOS**

**JUNHO/2015**

**ALEXANDRE MARCOS MENINO ALVES**

**ESTUDO SOBRE PERFORMANCE DE BANCO DE DADOS ORIENTADO A OBJETOS COMPARADO AO MODELO RELACIONAL**

Trabalho de Conclusão de Curso apresentado à Faculdade de Tecnologia Baixada Santista “Rubens Lara”, como exigência parcial para obtenção do título de Tecnólogo em Sistemas para Internet.

**Orientador: Professor Alexandre Garcia de Oliveira**

**SANTOS**

**JUNHO/2015**

Menino Alves, Alexandre Marcos.

**Estudo sobre performance de banco de dados orientado a objetos comparado ao modelo relacional /** Alexandre Marcos Menino Alves; orientador: Professor Alexandre Garcia de Oliveira – Local: Santos, 2015.

67 f.

Trabalho de Conclusão de Curso – Centro de Educação Tecnológica Paula Souza, Faculdade de Tecnologia da Baixada Santista “Rubens Lara”, Curso de Sistemas para Internet.

1. Banco de dados orientado a objetos, 2. Banco de dados relacional, 3. Benchmark (áreas de concentração)

Menino Alves, Alexandre Marcos

Aos meus pais, Ana e Marcos, e às minhas irmãs, Alessandra, Allyne e Amanda, pelo apoio, incentivo e paciência.

**AGRADECIMENTOS**

Agradeço a toda minha família por ser o suporte da minha vida. Sem eles, eu não teria o apoio, a disciplina, a força, a motivação e o amor suficientes para chegar ao final deste curso tão importante para mim.

Agradeço ao meu orientador Alexandre Garcia por ter sugerido o tema deste trabalho e por todo o auxílio prestado na elaboração do mesmo.

Além dele, agradeço a todos os demais professores de programação e banco de dados da Fatec “Rubens Lara” que, indiretamente em suas aulas, me fizeram descobrir que eu estava iniciando na carreira certa, e realmente é isso que eu gosto e quero me especializar e trabalhar por muito tempo.

E a todos os colegas que me acompanharam durante o curso, inclusive àqueles que não continuaram, pelo companheirismo, parceria e por fazerem esses últimos quase três anos mais alegres.

A vida merece algo além do aumento da sua velocidade.

Mahatma Gandhi

**RESUMO**

MENINO ALVES, AM. **Estudo sobre performance de banco de dados orientado a objetos comparado ao modelo relacional.** 2015, 67 f. Trabalho de Conclusão de Curso de Graduação de Tecnólogo em Sistemas para Internet. Centro Estadual de Educação Tecnológica Paula Souza, Faculdade de Tecnologia da Baixada Santista “Rubens Lara”, 2015.

Em um mundo globalizado, com rapidez de resultados, as pessoas buscam cada vez mais agilidade nas informações. Com isso, a tecnologia vem suprir esta demanda. Portanto, a pesquisa busca levantar um estudo sobre performance de banco de dados orientado a objetos comparado ao modelo relacional, o mais utilizado hoje em dia. Para atender estas características, foram necessárias análises em artigos, consultas em bibliografias especializadas, análises qualitativas e testes de performance para apontar qual modelo de banco de dados apresenta melhor desempenho. Então, serão apresentados conceitos sobre o modelo de banco de dados orientado a objetos, seus métodos e principais recursos. A proposta é testar e avaliar a performance de ambos os modelos com a finalidade de demonstrar qual banco de dados apresenta melhor eficiência diante dos casos apresentados.

**Palavra-Chave:** Banco de dados. Orientação a objetos. Modelo relacional. ObjectDB. MySQL.

**ABSTRACT**

MENINO ALVES, AM. **Estudo sobre performance de banco de dados orientado a objetos comparado ao modelo relacional.** 2015, 67 f. Trabalho de Conclusão de Curso de Graduação de Tecnólogo em Sistemas para Internet. Centro Estadual de Educação Tecnológica Paula Souza, Faculdade de Tecnologia da Baixada Santista “Rubens Lara”, 2015.

In a globalized world, with fast results, the people increasingly seek information speed. Thus, the technology has to meet this demand. Therefore, the monograph seeks to raise a study about database object-oriented performance compared to the relational model, which is the most widely used today. To meet these characteristics, analysis in articles, specialized bibliographies, qualitative searches and performance tests were necessary to point which database model presents better and faster result. So will be presented concepts about database object-oriented model, methods and major features. The proposal is to test and evaluate the performance of both models in order to show which had the best efficiency on the presented cases.

**Keywords:** Database. Object-oriented. Relational model. ObjectDB. MySQL.

**LISTA DE ABREVIATURAS E SIGLAS**

|  |  |
| --- | --- |
| ANSI  API  BD  BDOO  CPU  CRUD  DDL  DML  GB  GHz  GPU  ISAM  ISO  JDBC  JDO  JDOQL  JPA  JPQL  JVM  MR  ODBC  ODBMS  ODL  ODMG  OID  OO  OQL  ORM  PK  POO  RAM  SGBD  SGBDOO  SGBDR  SQL  TB | *American National Standards Institute*  *Application Programming Interface*  Banco de Dados  Banco de Dados Orientado a Objetos  *Central Processing Unit*  *Create Read Update Delete*  *Data Definition Language*  *Data Manipulation Language*  *Gigabytes*  *Gigahertz*  *General Public License*  *Indexed Sequential Access Method*  *International Standards Organization*  *Java Database Connectivity*  *Java Data Objects*  *Java Data Objects Query Language*  *Java Persistence API*  *Java Persistence Query Language*  *Java Virtual Machine*  Modelo Relacional  *Open Database Connectivity*  *Operational Database Management Systems*  *Object Definition Language*  *Object Data Management Group*  *Object Identifier*  Orientado a Objetos  *Object Query Language*  *Object Relational Mapping*  *Primary Key*  Programação Orientada a Objetos  *Random Access Memory*  Sistema Gerenciador de Banco de Dados  Sistema Gerenciador de Banco de Dados Orientado a Objetos  Sistema Gerenciador de Banco de Dados Relacional  *Structured Query Language*  *Terabytes* |
| URL | *Uniform Resource Locator* |
|  |  |
|  |  |
|  |  |

**LISTA DE FIGURAS**

|  |  |
| --- | --- |
| Figura 01 | Declaração de chave primária............................................................22 |
| Figura 02 | Exemplo de relacionamento um-para-um..........................................23 |
| Figura 03 | Exemplo de relacionamento um-para-muitos.....................................23 |
| Figura 04 | Exemplo de relacionamento muitos-para-muitos com tabela de junção.................................................................................................24 |
| Figura 05 | Diferença entre SGBDR x SGBDOO..................................................25 |
| Figura 06 | Exemplo de encapsulamento.............................................................28 |
| Figura 07 | Superclasse Pessoa...........................................................................29 |
| Figura 08 | Classes filhas da superclasse Pessoa...............................................29 |
| Figura 09 | Exemplo de uso de polimorfismo.......................................................30 |
| Figura 10 | Chamada de comportamento polimórfico..........................................30 |
| Figura 11 | Aplicação de *benchmarking.*..............................................................38 |
| Figura 12 | Simula o teste do cronômetro............................................................39 |
| Figura 13  Figura 14  Figura 15 | Uso da classe cronômetro.................................................................40  Diagrama de classes para testes no modelo relacional....................42  Diagrama de classes para testes no modelo orientado a objetos....49 |
| Figura 16 | Média de teste de inserção...............................................................58 |
| Figura 17 | Média de teste de seleção................................................................60 |
| Figura 18 | Média de teste de atualização..........................................................61 |
| Figura 19 | Média de teste de deleção................................................................63 |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

**LISTA DE TABELAS**

|  |  |
| --- | --- |
| Tabela 01 | Configurações do hardware utilizado.................................................41 |
| Tabela 02 | Inserção (*MySQL*)..............................................................................58 |
| Tabela 03 | Inserção (*ObjectDB*)...........................................................................58 |
| Tabela 04 | Seleção (*MySQL*)...............................................................................59 |
| Tabela 05 | Seleção (*ObjectDB*)............................................................................59 |
| Tabela 06 | Atualização (*MySQL*)..........................................................................61 |
| Tabela 07 | Atualização (*ObjectDB*)......................................................................61 |
| Tabela 08 | Deleção (*MySQL*)...............................................................................62 |
| Tabela 09 | Deleção (*ObjectDB*)............................................................................62 |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

**SUMÁRIO**

**1 INTRODUÇÃO** .......................................................................................................16

1.1 JUSTIFICATIVA DO TEMA..................................................................................16

1.2 PROBLEMA DA PESQUISA................................................................................17

1.3 OBJETIVOS.........................................................................................................18

**1.3.1 Objetivo geral**..................................................................................................18

**1.3.2 Objetivos específicos**.....................................................................................18

1.4 PROCEDIMENTOS METODOLÓGICOS............................................................18

1.5 ORGANIZAÇÃO DA MONOGRAFIA...................................................................19

**2 REFERENCIAIS TEÓRICOS**..................................................................................20

2.1 BANCO DE DADOS RELACIONAL.....................................................................20

**2.1.1 Chaves**..............................................................................................................21

**2.1.2 Relacionamento**...............................................................................................21

2.2 BANCO DE DADOS ORIENTADO A OBJETOS.................................................24

**2.2.1 Object Identifier - OID**.....................................................................................26

**2.2.2 Linguagens de manipulação de dados**.........................................................26

**2.2.3 Conceitos e recursos de orientação a objetos**............................................27

**3 TECNOLOGIAS**......................................................................................................32

2.2 MYSQL.................................................................................................................32

**3.1.1 Licença**.............................................................................................................33

**3.1.2 Características**.................................................................................................33

3.2 OBJECTDB..........................................................................................................33

**3.2.1 APIs**...................................................................................................................35

**4 METODOLOGIA**.....................................................................................................37

4.1 BENCHMARKING................................................................................................37

4.2 CONFIGURAÇÕES DO HARDWARE.................................................................41

4.3 CLASSES CRIADAS PARA TESTES NO MODELO RELACIONAL...................41

4.4 CLASSES CRIADAS PARA TESTES NO MODELO ORIENTADO A OBJETOS..................................................................................................................49

**5 RESULTADOS DOS TESTES**...............................................................................57

5.1 INSERÇÃO..........................................................................................................57

5.2 SELEÇÃO............................................................................................................59

5.3 ATUALIZAÇÃO....................................................................................................60

5.4 DELEÇÃO............................................................................................................62

**6 CONSIDERAÇÕES FINAIS**....................................................................................64

**REFERÊNCIAS**..........................................................................................................66

**1 INTRODUÇÃO**

Este estudo tem como proposta analisar a performance de um paradigma de banco de dados pouco difundido atualmente, que é o modelo orientado a objetos (CORBELLINI et al, 2010), e compará-lo com aquele que é o mais utilizado pela maioria das empresas e dos sistemas comercializados, o relacional (MACÁRIO; BALDO, 2005).

Para isso, será demonstrada uma série de testes idênticos utilizando as duas ferramentas envolvendo inserção, leitura, atualização e deleção de dados com o objetivo de comprovar qual delas apresenta melhor desempenho.

Nas análises realizadas neste trabalho foram utilizados o banco de dados orientado a objetos *ObjectDB*, para o modelo relacional, o *MySQL*, e para medir a performance de ambos, a ferramenta *Java Microbenchmark*.

Esta pesquisa foi baseada nos resultados dos testes citados, assim como em leitura de artigos, monografias e livros relacionados ao tema que pudessem embasar o assunto estudado e elucidar melhor sobre o cenário atual.

* 1. JUSTIFICATIVA DO TEMA

“Teste de performance é uma parte crucial de um processo de qualidade e teste de *software*” (CAMPOS, 2015). Profissionais que administram ou trabalham diretamente com banco de dados têm sempre como preocupação a velocidade das requisições das *queries* executadas e do tempo de resposta do SGBD (Sistema Gerenciador de Banco de Dados). Portanto estão continuamente buscando maneiras e artifícios para tornarem essas operações mais rápidas, dinâmicas, e ao mesmo tempo, confiáveis.

Existem diversas ferramentas de mapeamento objeto-relacional, que visam integrar as linguagens orientadas a objetos ao mundo relacional. Mas mesmo assim, se acaba perdendo diversos recursos e facilidades da orientação a objetos ao simular um banco de dados relacional, tendo que escrever uma grande quantidade de código para isso. Armazenar dados de objetos em um banco relacional é um trabalho bastante custoso, pois este modelo não possui funcionalidades suficientes para representar características básicas da orientação a objetos. (GALANTE et al, 2007)

Boa parte do tempo do desenvolvimento de um software é dedicada nessa fase de mapeamento. Com um banco orientado a objetos, além de não ser necessário investir esforços em ferramentas e códigos de mapeamento objeto-relacional, é possível aproveitar o máximo do paradigma orientado a objetos, permitindo criar objetos mais ricos e funcionais. (GALANTE et al, 2007)

Portanto, o tema central do trabalho se justifica por analisar se o este modelo de banco de dados pode ser também uma solução mais performática para programadores e analistas de sistemas que utilizam linguagens OO (Orientadas a Objetos), e ao mesmo tempo, lucrativa para empresas desenvolvedoras de software.

Boscarioli et al (2006) afirma que este paradigma oferece diversas vantagens, como o armazenamento direto de tipos de dados complexos entre outras, e este estudo pretende revelar se essa opção também é a que apresenta melhor desempenho.

* 1. PROBLEMA DA PESQUISA

Como citado acima, muitos dos recursos e do tempo de uma equipe de desenvolvimento de sistemas são investidos na fase de mapeamento objeto-relacional, a fim de implementar a base de dados à estrutura orientada a objetos e aproximar esses dois paradigmas. Mesmo assim, o resultado desse esforço não é perfeito e parte das vantagens de se utilizar uma linguagem OO se perde ao final desse processo. (GALANTE et al, 2007)

Além do mais, performance, uma das principais preocupações dos profissionais que estruturam e administram sistemas, é um fator que pode e deve ser melhorado e um problema evidente a ser lidado quando se trabalha com banco de dados. Então, o modelo orientado a objetos pode ser uma solução que pode facilitar não apenas na elaboração do sistema, mas também durante o uso do programa, pela velocidade nas respostas das requisições, e na manutenção do código-fonte.

* 1. OBJETIVOS

O enfoque desta pesquisa será delimitado e apresentado nos itens abaixo detalhando os objetivos geral e específico.

* + 1. **Objetivo geral**

Levantar um estudo sobre performance de banco de dados orientado a objetos comparado ao modelo relacional de maneira a destacar as principais características e diferenças entre ambos paradigmas, visando também contribuir para estudos e futuras pesquisas sobre o tema.

* + 1. **Objetivos específicos**

1. Apresentar detalhes a respeito dos dois modelos de banco de dados;
2. Explicar sobre os conceitos e funcionamento das ferramentas utilizadas neste trabalho;
3. Demonstrar como foram realizados os testes de performance e seus resultados.
   1. PROCEDIMENTOS METODOLÓGICOS

Para a realização do trabalho, além de análises em artigos e consultas em bibliografias especializadas, testes de desempenho foram desenvolvidos. Eles consistem em uma série de rotinas envolvendo inclusão, leitura, atualização e deleção de dados utilizando o modelo relacional e o orientado a objetos.

No início e ao final de cada rotina foi captado o tempo exato do momento da execução através da ferramenta *Java Microbenchmark* a fim de calcular o tempo total da execução, e dessa forma, medir o desempenho e comparar a performance dos dois modelos de banco de dados.

* 1. ORGANIZAÇÃO DA MONOGRAFIA

No capítulo 2, foram abordados os referenciais teóricos, onde são apresentados detalhes a respeito dos dois modelos de bancos de dados alvos do estudo deste trabalho. No terceiro capítulo, foram conceituadas as tecnologias utilizadas, no caso, o *MySQL* para o paradigma relacional e o *ObjectDB* para o modelo orientado a objetos. A seguir, na parte de metodologias, foram descritas informações sobre o meio utilizado para medir o desempenho dos bancos, a técnica de *benchmarking*. Após, no capítulo 5, é apresentado detalhes sobre os testes realizados e seus respectivos resultados. Por fim, o capítulo de conclusão com reflexões sobre as avaliações feitas e as considerações finais do trabalho.

**2 REFERENCIAIS TEÓRICOS**

A seguir, serão abordados conceitos sobre os dois paradigmas de banco de dados (BD) propostos nesta pesquisa. Dessa forma, neste capítulo, ambos os modelos serão devidamente conceituados e embasados em fontes bibliográficas a fim de apresentar, de maneira clara e confiável, informações sobre os padrões de banco orientado a objetos e relacional.

2.1 BANCO DE DADOS RELACIONAL

O paradigma relacional foi proposto por Edgar Codd em 1970 no artigo "*Relational Model of Data for Large Shared Data Banks*" (modelo de dados relacional para grandes bancos de dados compartilhados). Na publicação, Codd apresentava o novo modelo de dados baseando-se em conceitos matemáticos – teoria dos conjuntos e lógica do predicado.

Para Elmasri e Navathe (2011) apud Costa (2011), neste modelo de dados, o banco é representado como um conjunto de relações. Tendo em vista que, de certo modo, a relação é similar a uma tabela de valores e aplicando a terminologia do modelo relacional (MR), se diz que as linhas são chamadas de tuplas; as colunas, atributos; e a tabela em si, uma relação.

No modelo relacional a principal construção para representação dos dados é a relação, uma tabela com linhas não ordenadas e colunas. Uma relação consiste de um esquema e de uma instância. O esquema especifica o nome da relação e domínio de cada coluna, também denominada atributo ou campo da relação. (MACÁRIO; BALDO, 2005)

O nome da coluna, nesse modelo de BD, é uma propriedade fundamental, no qual deve ser único dentro de uma mesma tabela. Já o domínio pode ser denominado como o tipo de dado da coluna, onde será definido se esse campo receberá valores alfanuméricos, apenas numéricos (inteiros ou decimais), booleanos, etc.

Segundo Macário e Baldo (2005), a instância de uma relação é o conjunto de linhas (tuplas), distintas entre si, que compõem a relação em um dado momento. Sendo que ela é variável, já que a quantidade de tuplas e o conteúdo de seus atributos podem variar.

Para trabalhar com banco de dados relacional, é utilizado o *Structured Query Language* (SQL), ou linguagem de consulta estruturada. Esse padrão criado no início dos anos 1970 pela *IBM Research* para produzir o protótipo de um sistema de BD chamado de *System R*. (DATE, 2004, apud COSTA, 2011)

Hoje, o SQL é a linguagem padrão para os sistemas gerenciadores de banco de dados (SGBD) relacionais e foi baseada nos conceitos de álgebra e cálculo relacional. Ela é padronizada conjuntamente pelo *American National Standards Institute* (ANSI) e *International Standards Organization* (ISO). (ELMASRI; NAVATHE, 2011, apud COSTA, 2011)

O SQL possui dois componentes fundamentais para o manuseio dos dados. São eles o *Data Definition Language* (DDL), ou linguagem de definição de dados, e o *Data Manipulation Language* (DML), ou linguagem de manipulação de dados.

A DDL compreende os comandos para criação, modificação e deleção da estrutura das tabelas (relações). As suas cláusulas principais são *CREATE TABLE*, *ALTER TABLE* e *DROP TABLE*.

Já a DML é o subconjunto do SQL usado para inserção, modificação, recuperação e exclusão dos dados contidos nas tabelas. E seus principais comandos são *INSERT INTO*, *UPDATE FROM*, *SELECT* e *DELETE FROM*.

Há décadas, o modelo relacional (MR) é o mais difundido no mercado e nas faculdades de tecnologia. Com os primeiros sistemas comerciais disponibilizados a partir dos anos 80, existe uma série SGBDs para se trabalhar com MR, entre eles, os principais são *MySQL*, *Oracle* e *SQL Server*. (ELMASRI; NAVATHE, 2011, apud COSTA, 2011)

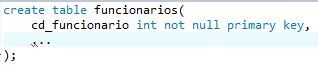
**2.1.1 Chaves**

As chaves são campos da tabela usados, separadamente ou em conjunto, para identificar um registro no banco de dados e torna-lo único. (MARTINS, 2012)

Uma chave em banco de dados relacional é denominada como primária, ou *primary key* (PK), quando é usada unicamente para identificar um registro. Se para isso, for utilizado dois ou mais campos de uma mesma tabela, é caracterizado o uso de uma chave composta ou concatenada. (MARTINS, 2012)

Exemplo de declaração de chave primária:

**Figura 1:** Declaração de chave primária**.**



**Fonte:** Elaborada pelo autor.

Outro tipo fundamental no MR é a chamada chave estrangeira ou secundária. Trata-se da identificação de um relacionamento existente entre distintas tabelas. É um campo que serve de referência à PK de outra entidade, caracterizando assim, um relacionamento entre elas.

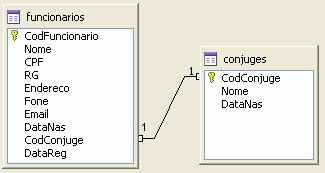
**2.1.2 Relacionamento**

Relacionamento ocorre quando um ou mais dados de uma entidade estão relacionados com um ou mais campos de outra. Por exemplo, um sistema de *blog* possui uma tabela para cadastro de usuários e outra para *posts*. Cada usuário pode publicar infinitos *posts* e cada *post* pode ter apenas um usuário, portanto existe a necessidade de ter um campo na entidade *posts* que receba o valor da *primary key* do usuário (chave estrangeira). Logo, este é um caso de relacionamento entre tabelas. (SERRANO, 2015)

Em um sistema de banco de dados de modelo relacional, podemos ter os seguintes tipos:

1. um-para-um (1:1): Quando um registro de uma entidade possui um único correspondente na tabela relacionada.

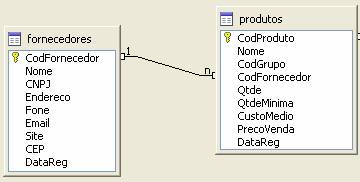
**Figura 2:** Exemplo de relacionamento um-para-um.



**Fonte:** Fais (2015).

1. um-para-muitos (1:N): Para Serrano (2010), neste relacionamento, “um dado da tabela um pode estar relacionado a diversos dados da tabela dois, porém cada dado da tabela dois estão relacionados a apenas um dado da tabela um”.

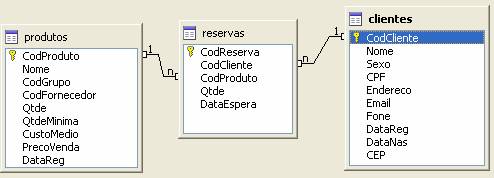
**Figura 3:** Exemplo de relacionamento um-para-muitos.



**Fonte:** Fais (2015).

1. muitos-para-muitos (N:N): Neste caso, diversos dados da primeira tabela podem estar relacionados com a segunda tabela e vice versa. Martins (2012) cita que este tipo de relacionamento exige a existência de uma tabela de junção entre as duas participantes.

**Figura 4:** Exemplo de relacionamento muitos-para-muitos com tabela de junção.



**Fonte:** Fais (2015).

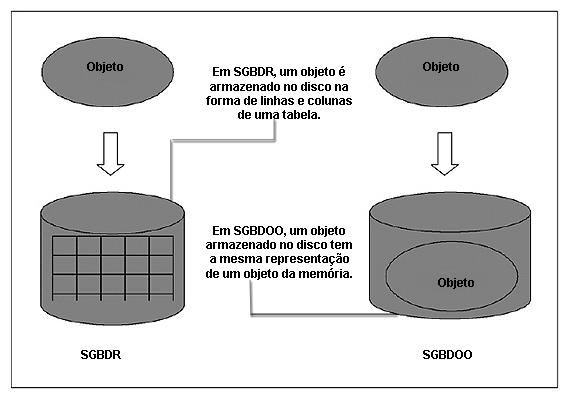
2.2 BANCO DE DADOS ORIENTADO A OBJETOS

Com o advento das linguagens orientadas a objetos, o uso do modelo relacional tornou-se mais complexo e passou a existir a necessidade de se criar uma série de ajustes para que banco de dados e programa pudessem interagir. Por exemplo, os objetos começaram a ser adaptados para que seus dados fossem gravados no BD. A partir desse novo cenário, surgiu o banco de dados orientado a objetos (BDOO), com a finalidade de armazenar dados mais complexos e de tornar a modelagem deles mais simples. (MARTINS, 2012)

BDOO é que um modelo de BD em que os dados são armazenados e recuperados em forma de objetos utilizando os conceitos da programação OO.

A Figura 2 exemplifica como o os objetos são armazenados neste paradigma comparando com o esquema relacional.

**Figura 5:** Diferença entre SGBDR x SGBDOO.



**Fonte:** Martins (2012) adaptado de Kahate (2006).

Existe uma série de entraves para se trabalhar com banco de dados relacional em uma linguagem OO, uma delas é que em diversas situações, os atributos de um mesmo objeto precisam ser divididos e gravados em tabelas diferentes. E para recuperar esses dados, passa a existir a necessidade de usar um *JOIN* dessas tabelas para que os dados voltem a ser atribuídos a um mesmo objeto. (PINHEIRO et al, 2009)

Segundo Pinheiro el al (2009), com a orientação a objetos, passou a ser possível modelar objetos de uma maneira mais próxima do mundo real e um BDOO permite ainda que a aplicação manipule os objetos, mesmo que não sejam persistentes, devido à possibilidade de armazenar todo o objeto e não apenas os seus atributos.

Em bancos de dados OO, o estado (valor corrente) de um objeto complexo pode ser construído a partir de outros objetos (ou outros valores) pelo uso de alguns construtores de tipos. Um modo formal de representar tais objetos é visualizar cada objeto como uma tripla (i, c, v), na qual i é o identificador único do objeto (o OID), C é um construtor de tipo (ou seja, uma indicação de como o estado do objeto é construído) e v é o estado do objeto (ou valor corrente). (ELMASRI; NAVATHE, 2005, apud MARTINS, 2012)

**2.2.1 *Object Identifier –* OID**

Ao contrário do que acontece no paradigma relacional com as tabelas, o modelo OO não utiliza chaves primárias ou secundárias. Neste caso, é usado o identificador de objeto, ou *object identifier* (OID). Ele é definido e manipulado pelo próprio sistema gerenciador de banco de dados orientado a objetos (SGBDOO) e pode ser considerado “como uma referência ao objeto em memória, assemelhando-se a um ponteiro, porém o OID nunca é alterado nem reaproveitado, diferentemente do que acontece quando o objeto está em memória”. (PINHEIRO et al, 2009)

De acordo com Corbellini et al (2010), as propriedades mais importantes da identidade do objeto são: ser inalterável e o fato de ser utilizado apenas uma vez. Ainda que o objeto seja removido do banco, este mesmo *object identifier* não pode ser usado por outro objeto. O mesmo autor completa lembrando que o OID pode ser comparado com a chave primária do modelo relacional.

Apesar da existência do OID, nada impede que o desenvolvedor crie classes com atributos específicos como chave que recebam valores auto incrementados para facilitar a identificação dos objetos gravados no banco.

**2.2.2 Linguagens de definição e manipulação de dados**

As linguagens de definição, *object definition language* (ODL), e manipulação, *object query language* (OQL), surgiram a fim de criar padrões para BDOO independentemente do SGBDOO utilizado, assim como acontece no paradigma relacional.

Essas definições foram oficializadas pelo *Object Data Management Group* (ODMG), em uma publicação intitulada como DBMG-93, sendo este o primeiro documento a tratar do assunto. (MARTINS, 2012)

Segundo Galante et al (2007), a ODL tem o objetivo de criar especificações de objetos, ou seja, classes e interfaces, e não pode ser considerada uma linguagem de programação completa. Ela permite que o usuário especifique um banco de dados independentemente da linguagem usada.

A OQL é utilizada para consulta de objetos e segue um padrão de sintaxe praticamente idêntica ao SQL no modelo relacional, dando suporte a cláusulas como *SELECT*, *FROM*, *WHERE*, *GROUP BY* e *ORDER BY*, porém retorna atributos de objetos. Para Galante et al (2007), se trata de uma linguagem de consulta declarativa que prevê suporte ao tratamento de objetos complexos, invocação de métodos, herança e polimorfismo.

Apesar de ser um padrão ainda seguido por diversos SGBDOOs, de acordo com o site da *Operational Database Management Systems* (OBDMS), a ODMG concluiu as suas atividades sobre normas de gestão de dados de objetos em 2001. E a API (*Application Programming Interface*) padrão para bancos OO passou a ser o *Java Data Objects* (JDO).

O JDO possui uma série de métodos para efetuar operações de CRUD (*Create; Read; Update; Delete*), e assim como SQL no modelo relacional o e OQL no padrão da ODMG, também apresenta uma solução em que é possível estruturar *queries*. Trata-se do *Java Data Objects Query Language* (JDOQL).

No caso do *ObjectDB*, o banco orientado a objetos escolhido para este trabalho, além do JDO, também é possível utilizar o *Java Persistence API* (JPA). E a exemplo da outra API, possui métodos prontos para a manipulação de dados e uma linguagem para a utilização de *queries*, o *Java Persistence Query Language* (JPQL).

**2.2.3 Conceitos e recursos de orientação a objetos**

A orientação a objetos surgiu no final dos anos 1960 com o desenvolvimento de uma linguagem chamada *Simula-67*, que introduziu conceitos de classe e herança. Já o termo “programação orientada a objetos” (POO) passou a ser mencionado apenas na década de 1970, quando Alan Kay e outros desenvolvedores criaram a linguagem *SmallTalk.* (MARTINS, 2012)

O objeto, segundo Martins (2012), representa a abstração de uma entidade do mundo real ou o estado e comportamento do objeto em relação as suas propriedades. “Corresponde a uma instância de uma determinada classe” e “o estado de um objeto pode ser alterado através de métodos”.

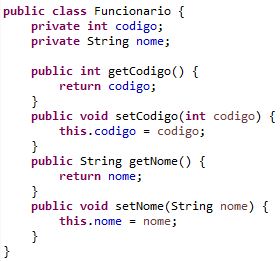
Um objeto é algo que possui significado para uma aplicação. Mesmo sendo derivados de uma mesma classe e possuindo os mesmos atributos, os objetos são distintos entre si, pois se diferenciam pelos valores de seus atributos e relacionamento com os outros objetos. (BLAHA; RUMBAUGH, 2006, apud MARTINS, 2012)

Em POO são usados recursos como encapsulamento, herança, polimorfismo e sobrecarga. Cada um deles serão apresentados na sequência.

Encapsulamento, em OO, está relacionado “com os conceitos de tipos abstratos de dados e ocultar informação nas linguagens de programação”. (ELMASRI; NAVATHE, 2005, apud PINHEIRO et al, 2009)

Para Pinheiro et al (2009), ao encapsular os dados, as variáveis passam a ser acessadas apenas por métodos definidos em sua estrutura. Sendo assim, os dados do sistema ficam protegidos por esses métodos de acesso, tornando o programa mais seguro, pois dessa forma é possível saber como os dados estão sendo acessados.

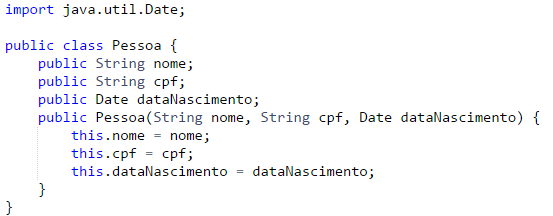
**Figura 6:** Exemplo de encapsulamento.



**Fonte:** Elaborada pelo autor.

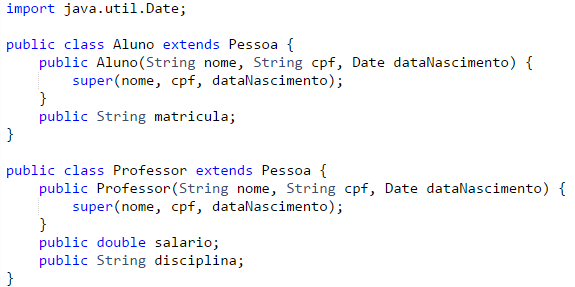
Herança é uma característica da programação OO que permite a uma classe reaproveitar, ou seja, herdar métodos e atributos de outra classe. Esse mecanismo geralmente é usado quando a chamada superclasse ou classe mãe é utilizada como base para a criação de outras, conhecidas como subclasses ou classes filhas.

**Figura 7:** Superclasse Pessoa.



**Fonte:** Adaptado de Geovane (2015).

**Figura 8:** Classes filhas da superclasse Pessoa.



**Fonte:** Adaptado de Geovane (2015).

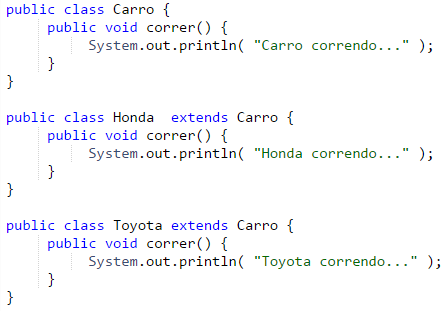
No exemplo acima, o conceito de herança acontece porque as classes filhas mostradas na Figura 8, através do uso da palavra-chave *extends*, passam a possuir os mesmos atributos e o método da superclasse apresentada na Figura 7.

A palavra “polimorfismo” significa “muitas formas” e é o termo definido em orientação a objetos que permite ao desenvolvedor usar o mesmo elemento de diferentes formas. Pode ser do tipo de sobrecarga ou de sobreposição. (MEDEIROS, 2015)

O polimorfismo de sobrecarga ocorre quando uma mesma operação é implementada várias vezes na mesma classe. A operação que será chamada depende da assinatura dos métodos sobrecarregados. (MEDEIROS, 2015)

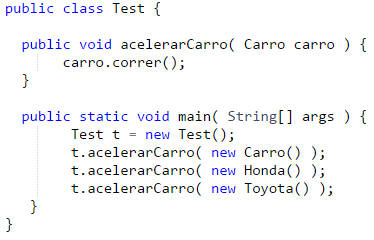
O tipo de sobreposição acontece na herança, no momento em que a subclasse sobrepõe o método original definido na superclasse. A ação ocorre em tempo de execução e não de compilação. O método chamado depende do tipo do objeto que recebe a mensagem. (MEDEIROS, 2015)

**Figura 9:** Exemplo de uso de polimorfismo.



**Fonte:** Elaborada pelo autor.

**Figura 10:** Chamada de comportamento polimórfico.



**Fonte:** Elaborada pelo autor.

Na chamada apresentada na Figura 10, o mesmo método retornará um resultado diferente de acordo com o argumento que é passado.

**3 TECNOLOGIAS**

Neste capítulo, serão apresentadas as tecnologias utilizadas para a realização deste trabalho. Entre elas, os sistemas gerenciadores de banco de dados (SGBD) relacional, o *MySQL*, e orientado a objetos, o *ObjectDB*.

3.1 MYSQL

O *MySQL* é um SGBD relacional criado primeiramente para trabalhar com aplicações de médio e pequeno portes, mas hoje atende a grandes projetos. Possuindo características de um banco de dados de alta capacidade e performance, é reconhecido por algumas entidades como o melhor SGBD de código aberto (*open source*) para concorrer com outros sistemas similares pagos, como o *SQL Server*, da *Microsoft*, e o *Oracle*. (MILANI, 2007, p. 22)

Inicialmente, o MySQL foi projetado para trabalhar com aplicações de pequeno a médio porte, algo em torno de 100 milhões de registros por tabela, tendo como tamanho médio aproximadamente 100MB por tabela. Contudo, esses números eram os recomendados para as primeiras versões da ferramenta. Atualmente os limites e capacidades do MySQL ultrapassaram essas fronteiras inúmeras vezes. (MILANI, 2007, p. 24)

A origem do *MySQL* se deu na década de 1990, quando os desenvolvedores David Axmark, Allan Larsson e Michael Widenius precisaram de uma interface SQL compatível com as rotinas ISAM (*Indexed Sequential Access Method*) que utilizavam em suas aplicações e tabelas. Usando as linguagens *C* e *C++*, criaram uma nova API que resultou em um novo sistema que viria a ser o *MySQL*. (MILANI, 2007, p. 23) Hoje, o programa é desenvolvido e distribuído pela *Oracle Corporation*.

Como padrão para banco de dados relacionais, este SGBD utiliza o SQL para o acesso e manipulação dos seus dados.

Conforme informações divulgadas em seu site oficial, grandes empresas e sites do mundo são clientes que utilizam *MySQL*. Alguns deles: *YouTube, PayPal, Google, Facebook, Twitter, eBay, Cisco, Linkedin, Adobe, Amazon, Walmart, Dropbox, Ticketmaster*, *Symantec*, *DirecTV, Sony, HP, Nokia* e *NASA*.

Além de oferecer suporte a praticamente todos os sistemas operacionais do mercado, o *MySQL*, com o auxílio de *drivers* como ODBC e JDBC, pode ser utilizado em diversas linguagens de programação. Entre elas: *Java, C/C++, C#, Python, PHP e Ruby.* (ALECRIM, 2015)

**3.1.1 Licença**

O *MySQL* possui dois tipos de licença, sendo uma delas a de *software* livre, que permite a utilização e a distribuição aberta do programa entre todas as pessoas, seja qual for o propósito. Além disso, o seu código-fonte também é distribuído para que qualquer programador possa alterá-lo a fim de atender às suas necessidades. Para mais detalhes sobre esse tipo de licença, deverão ser consultadas as cláusulas da GNU-GPL (*General Public License*). (MILANI, 2007, p. 23)

Outra licença que o *MySQL* possui é a comercial, que dá acesso a suporte diferenciado e permite obter pacotes com mais ferramentas, entre outras vantagens. Geralmente, essa licença é mais utilizada por sistemas maiores e que possuam uma complexidade maior no armazenamento e no processamento dos dados.

**3.1.2 Características**

Entre as principais características do *MySQL*, se destaca o fato de ser *multithreads*, ou seja, usa a programação de *threads* diretamente no *kernel* da plataforma, o que aumenta consideravelmente a velocidade de processamento e facilita a integração da ferramenta em *hardwares* com mais de uma CPU. (MILANI, 2007, p. 26)

Mais um destaque da ferramenta é a forma de armazenamento. O *MySQL* disponibiliza vários tipos de tabelas para a gravação dos dados, tendo cada uma suas próprias características. “A vantagem dessa variedade de tabelas é a possibilidade de escolher o tipo em cada situação diferente. Enquanto um tipo prioriza velocidade, outro prioriza volume de dados, entre outras características”. (MILANI, 2007, p. 26)

3.2 OBJECTDB

O *ObjectDB* é um sistema gerenciador de banco de dados orientado a objetos (SGBDOO) que possui todos os serviços padrões de um SGBD, como armazenamento e recuperação de dados, operações, processamento de consultas, etc. Usando o modelo de orientação a objetos para o gerenciamento e a manipulação dos dados, no *ObjectDB*, assim como em outros SGBDOOs, não é necessário definir tabelas e nem converter objetos para gravá-los usando SQL, como acontece em banco de dados relacionais. (OBJECTDB, 2015)

Diferente de outras soluções *NoSQL* (que não usa SQL), este SGBDOO fornece suporte a *queries* mais complexas usando duas linguagens padrões de consulta para *Java*: JPQL (*Java Persistence Query Language*) e JDOQL (*Java Data Objects Query Language*). De acordo com o site do *ObjectDB*, a capacidade dessas linguagens é equivalente a do SQL, mas com a vantagem de ser mais fácil de usar em uma estrutura orientada a objetos.

Segundo o seu site oficial, para ser usado, o *ObjectDB* requer a versão 5 do *Java* ou posterior, sendo recomendável o uso a partir da 6. Por se tratar de uma aplicação que usa apenas *Java*, este SGBDOO é capaz de rodar em qualquer plataforma.

O *ObjectDB* pode ser utilizado por qualquer linguagem que rode na *Java Virtual Machine* (JVM), como *Scala*, *Groovy*, além do próprio *Java*. Também é possível usá-lo em outras, como *Python, Ruby* e *PHP*, mas através de suas respectivas implementações JVM, tais quais *Jython, JRuby* e *Quercus*. (OBJECTDB, 2015)

Entre as principais características do *ObjectDB*, se destaca a alta escalabilidade e a capacidade de gerenciar bases de dados de diferentes tamanhos. Um arquivo deste banco de dados OO pode ter até 128 TB (*Terabytes*). Além disso, pode ser incorporado em aplicações *multithreads*, com vários usuários e ilimitadas conexões ao BD. (OBJECTDB, 2015)

**3.2.1 APIs**

Para se trabalhar com *ObjectDB*, existem duas opções de APIs: JPA (*Java Persistence API)* e JDO (*Java Data Objects*).

O JPA é uma API padrão para acesso a banco de dados a partir de aplicações escritas em *Java*. A principal vantagem do JPA sobre o JDBC (*Java Database Connectivity*), conjunto de classes e interfaces em *Java* usado para enviar instruções SQL para bancos relacionais, é que os dados são representados por classes e objetos em vez em vez de tabelas. (OBJECTDB, 2015)

Uma implementação JPA, geralmente, é usada para permitir a interação com bancos relacionais, tendo como um dos principais exemplos o *Hibernate*. Essas aplicações são chamadas de ferramentas de mapeamento objeto relacional, ou *object relational mapping* (ORM), e servem para criar representações de um BD relacional (tabelas e registros) em aplicações orientada a objetos (classes e objetos). (OBJECTDB, 2015)

Conforme o site oficial do *ObjectDB*, existem duas vantagens de trabalhar com JPA. Uma é a portabilidade do programa desenvolvido, e outra é a não utilização de uma camada intermediária ORM no processo, deixando assim, a ferramenta mais performática, e consequentemente, a aplicação mais rápida.

Por ser mais usual, outro proveito que se pode ter ao escolher o JPA é a facilidade de encontrar fontes de auxílio como livros, tutoriais, exemplos de códigos, além de ferramentas, e um maior número de desenvolvedores com conhecimento nesta API.

O JDO é outro padrão utilizado para acesso de dados persistentes. Diferentemente do JPA, que foi projetado para trabalhar apenas com bancos relacionais, esta API foi desenvolvida para ser usada também com o modelo orientado a objetos. Certamente, este é o padrãomais usado pelos demais SGBDOOs e por isso é considerada a opção mais vantajosa quando se deseja ter uma maior portabilidade com outros bancos OO. (OBJECTDB, 2015)

**4 METODOLOGIA**

Na sequência deste estudo, serão apresentados o meio utilizado para a medição de desempenho dos bancos apresentados e detalhes a cerca das configurações do *hardware* usado para a elaboração dos testes.

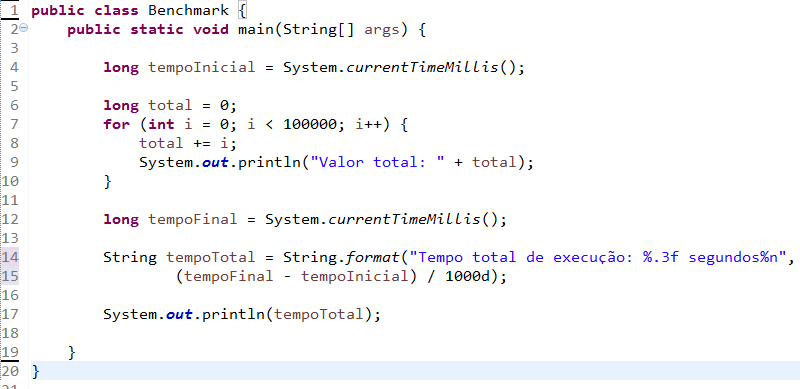
4.1 BENCHMARKING

Para avaliar a performance dos sistemas gerenciadores de bancos de dados citados neste estudo, foi aplicado o processo de *benchmarking*. Trata-se de um procedimento comparativo entre operações distintas que serve principalmente para medir o tempo de processamento do programa. Podem ser testados desde pequenos trechos de algoritmos até sistemas completos.

O aspecto fundamental do *benchmarking* é a comparação. Um único resultado produzido por ele não tem muito valor, só é válido quando ocorre a comparação. *Benchmarks* tipicamente medem a quantidade de tempo que se leva para realizar uma tarefa específica, mas também podem ser usadas para medir outras variáveis tais como a quantidade de memória necessária para realizar determinada tarefa. (RIBEIRO FILHO, 2010)

O ambiente de desenvolvimento integrado utilizado para a realização dos testes de desempenho é o *Eclipse Luna* com a linguagem *Java* *8*. Na elaboração dos *benchmarks*, foi usado o método *currentTimeMillis()* que pertence à classe *System*. De acordo com Wilson (2001) apud Ribeiro Filho (2010), este método retorna uma variável do tipo *long* contendo a quantidade de milissegundos contados a partir da meia-noite de 1º de janeiro de 1970. Então, para que o tempo de processamento seja calculado, é necessário armazenar o tempo corrente do início e do final da seção de código que se deseja testar e subtrair ambos os valores ao fim da execução. É como se usasse um cronômetro que inicia e para automaticamente dentro do próprio algoritmo. Exemplo na figura a seguir.

**Figura 11:** Aplicação de *benchmarking*.



**Fonte:** Elaborada pelo autor.

Segundo Ribeiro Filho (2010), uma boa prática para a realização desse tipo de *benchmarking* consiste no encapsulamento, o que significa criar uma classe apenas para o comportamento do cronômetro. Dessa forma, bastaria instanciar um objeto dessa classe e aplicar os seus métodos para a medição da performance no código desejado. Com isso, o desenvolvimento fica mais simples e passa a atender um dos princípios fundamentais da orientação a objeto, que é a reutilização de código. Sendo assim, o desenvolvedor passa a ter algumas vantagens como não se preocupar com as rotinas responsáveis pelo *benchmarking* e não ter a necessidade de replicar o algoritmo de teste em diferentes partes do sistema, o que evita possíveis erros durante a codificação. Exemplo da classe e da aplicação do objeto nas figuras 12 e 13.

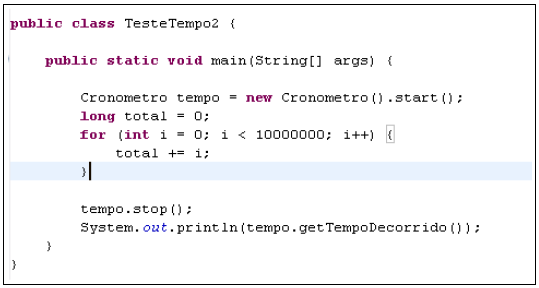
**Figura 12:** Simula o teste do cronômetro.



**Fonte:** Ribeiro Filho (2010).

Pode-se observar na Figura 12, a construção da classe que simula o funcionamento do cronômetro, com os métodos *start()* e *stop()*, usados para captar o tempo inicial e final da execução do programa através do método *currentTimeMillis()*. Além disso, também é gravado em uma variável do tipo *boolean* se a marcação do tempo foi iniciada. No método *getTempoDecorrido()* é calculada subtração dos tempos registrados a fim de obter o tempo total de execução. E há também o método *reset()* para limpar todos os registros e permitir o cálculo de um novo *benchmark*.

**Figura 13:** Uso da classe cronômetro.



**Fonte:** Ribeiro Filho (2010)

Existem duas categorias de *benchmarks*. Uma delas chama-se *macro-benchmark* e serve para a realização de testes no sistema como um todo a partir do ponto de vista do usuário. A outra é chamada de *micro-benchmark* e trata-se pontos de referência no código para medir o tempo de uma função específica e geralmente é aplicado em determinados trechos do programa. Este último é o utilizado na elaboração deste trabalho. (WILSON, 2001, apud RIBEIRO FILHO, 2010)

Um dos proveitos que se pode tirar do uso de *benchmarking* é poder avaliar a evolução do sistema durante o período de desenvolvimento. Com essa técnica é possível quantificar, por exemplo, o impacto de performance que uma nova funcionalidade ou alteração de rotina afetou no processamento. Dessa forma inclusive, esse processo torna possível o monitoramento constante do desempenho do programa e auxilia na elaboração de algoritmos mais performáticos.

4.2 CONFIGURAÇÕES DO HARDWARE

**Tabela 1:** Configurações do hardware utilizado.

|  |
| --- |
| **Processador:** *Intel® Core™* i5-2450M CPU @ 2.50 GHz |
| **Memória RAM:** 4,00 GB |
| **Sistema Operacional:** *Windows 7 Home Basic* 64 bits. |

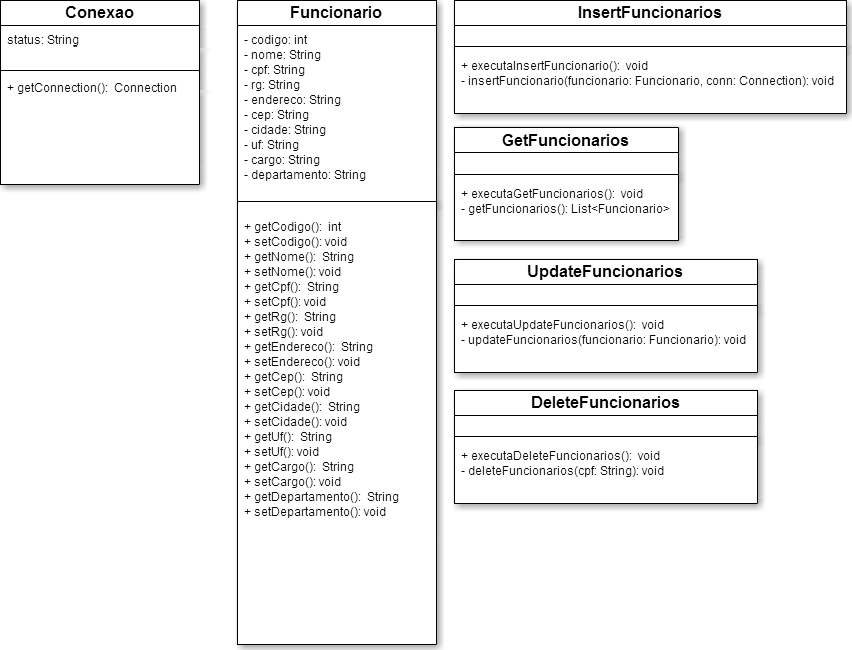
**Fonte:** Elaborada pelo autor.

O computador utilizado para a realização dos testes foi um *notebook* *Dell* do modelo *Inspiron N4050*, com processador *Intel® Core™* i5-2450M de 2.50 GHz. A memória RAM instalada é de 4,00 GB e utiliza um sistema operacional *Windows 7 Home Basic* de 64 *bits*.

4.3 CLASSES CRIADAS PARA TESTES NO MODELO RELACIONAL

Para os testes de performance no modelo relacional, foram utilizadas as seguintes classes representadas no diagrama demonstrado na Figura 14.

**Figura 14:** Diagrama de classes para testes no modelo relacional.



**Fonte:** Elaborada pelo autor.

Foi criada também uma tabela na base de dados através do *software MySQL Workbench 6.2 CE* para receber um cadastro de funcionários.

No projeto utilizado para os testes foi usada uma classe para conexão com o banco de dados através da API JDBC. Nela, existe um método chamado *getConnection()* que é será utilizado para todas as rotinas de manipulação de dados no projeto. Abaixo, o código da classe.

**import** java.sql.Connection;

**import** java.sql.DriverManager;

**import** java.sql.SQLException;

**public** **class** Conexao {

**static** String *status* = "";

**public** **static** Connection getConnection(){

Connection conn = **null**;

**try**{

Class.*forName*("com.mysql.jdbc.Driver").newInstance();

String url = "jdbc:mysql://localhost/test?user=root&password=root";

conn = DriverManager.*getConnection*(url);

*status* = "Connection opened";

}**catch** (SQLException e){

*status* = e.getMessage();

}**catch** (ClassNotFoundException e){

*status* = e.getMessage();

}**catch** (Exception e){

*status* = e.getMessage();

}

**return** conn;

}

}

A URL passada como parâmetro no método *getConnection*, da classe *DriverManager*, deve possuir os valores de usuário e senha configurados na parte administrativa do banco de dados.

Após isso, foi criada uma classe *bean* para armazenar os dados cadastrais do funcionário. Essa classe será usada como objeto como será visto na sequência.

Foi criada uma classe para cada operação de teste, sendo elas: inserção, seleção, atualização e exclusão de dados.

Na classe responsável pela inserção, foram utilizados dois métodos. Um deles, privado para execução da *query* para armazenar os dados vindos do objeto funcionário. O outro método da classe é público, onde é preenchido os atributos do funcionário e chama o método privado dentro de um *loop* para inserir 5000 registros.

Marcadores de tempo inicial e final são utilizados para técnica de *benchmarking*. Segue o código-fonte da classe.

**import** java.sql.Connection;

**import** java.sql.PreparedStatement;

**import** java.sql.SQLException;

**public** **class** InsertFuncionarios {

**public** **void** executaInsertFuncionario() {

**try**{

Funcionario funcionario = **new** Funcionario();

funcionario.setNome("Alexandre Marcos");

funcionario.setCpf("123.654.789-00");

funcionario.setRg("12345876");

funcionario.setEndereco("Rua Jorge Tibiriçá");

funcionario.setCep("11055-250");

funcionario.setCidade("Santos");

funcionario.setUf("SP");

funcionario.setCargo("Programador");

System.***out***.println("Iniciando teste de insert com MySQL...");

**long** tempoInicial = System.*currentTimeMillis*();

Connection conn = Conexao.*getConnection*();

**for** (**int** i = 0; i < 5000; i++) {

funcionario.setCodigo(i);

insertFuncionario(funcionario,conn);

System.***out***.println("Funcionário inserido com o código: "+i);

}

conn.close();

**long** tempoFinal = System.*currentTimeMillis*();

String tempoTotal = String.*format*("Tempo total de execução: %.3f segundos%n", (tempoFinal - tempoInicial) / 1000d);

System.***out***.println(tempoTotal);

}**catch**(SQLException e){

System.***out***.println("Erro no método executaInsertFuncionario: "+e.getMessage());

}

}

**private** **void** insertFuncionario (Funcionario funcionario,Connection conn){

**try**{

StringBuilder qry = **new** StringBuilder();

PreparedStatement ps = **null**;

**int** i = 1;

qry.append(" insert into funcionarios (cd\_funcionario, nm\_funcionario, cd\_cpf, cd\_rg, ds\_endereco, cd\_cep, nm\_cidade, sg\_uf, nm\_cargo) ");

qry.append(" values (?,?,?,?,?,?,?,?,?) ");

ps = conn.prepareStatement(qry.toString());

ps.setInt(i++, funcionario.getCodigo());

ps.setString(i++, funcionario.getNome());

ps.setString(i++, funcionario.getCpf());

ps.setString(i++, funcionario.getRg());

ps.setString(i++, funcionario.getEndereco());

ps.setString(i++, funcionario.getCep());

ps.setString(i++, funcionario.getCidade());

ps.setString(i++, funcionario.getUf());

ps.setString(i++, funcionario.getCargo());

ps.execute();

ps.close();

}**catch**(SQLException e){

System.***out***.println("Erro no método insertFuncionario: "+e.getMessage());

}

}

}

Na classe de seleção de dados, a estrutura é semelhante à de inserção de dados. Um método privado para enviar a *query* que receba todos os registros da tabela funcionário para armazená-los em uma lista, e um outro público para a chama-lo.

A diferença é que o método privado não é chamado dentro do *loop*. Mas sim, a lista retornada é percorrida e o resultado dos 5000 registros é exibido. Os *benchmarks* também são marcados nessa classe conforme mostra código-fonte a seguir.

**import** java.sql.Connection;

**import** java.sql.PreparedStatement;

**import** java.sql.ResultSet;

**import** java.sql.SQLException;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** GetFuncionarios {

**public** **void** executaGetFuncionarios(){

System.***out***.println("Iniciando teste de get com MySQL...");

**long** tempoInicial = System.*currentTimeMillis*();

List<Funcionario> listaFuncionario = getFuncionarios();

**for** (Funcionario funcionario : listaFuncionario) {

System.***out***.println(funcionario.getCodigo() + " | " + funcionario.getNome() + " | " + funcionario.getCargo() + " | " + funcionario.getCidade());

}

**long** tempoFinal = System.*currentTimeMillis*();

String tempoTotal = String.*format*("Tempo total de execução: %.3f segundos%n", (tempoFinal - tempoInicial) / 1000d);

System.***out***.println(tempoTotal);

}

**private** List<Funcionario> getFuncionarios(){

List<Funcionario> lista = **new** ArrayList<Funcionario>();

**try**{

Connection conn = Conexao.*getConnection*();

StringBuilder qry = **new** StringBuilder();

PreparedStatement ps = **null**;

ResultSet rs = **null**;

Funcionario funcionario = **null**;

qry.append(" select \* from funcionarios order by cd\_funcionario ");

ps = conn.prepareStatement(qry.toString());

rs = ps.executeQuery(qry.toString());

**while**(rs.next()){

funcionario = **new** Funcionario();

funcionario.setCodigo(rs.getInt("cd\_funcionario"));

funcionario.setNome(rs.getString("nm\_funcionario"));

funcionario.setCpf(rs.getString("cd\_cpf"));

funcionario.setRg(rs.getString("cd\_rg"));

funcionario.setEndereco(rs.getString("ds\_endereco"));

funcionario.setCep(rs.getString("cd\_cep"));

funcionario.setCidade(rs.getString("nm\_cidade"));

funcionario.setUf(rs.getString("sg\_uf"));

funcionario.setCargo(rs.getString("nm\_cargo"));

lista.add(funcionario);

}

rs.close();

ps.close();

conn.close();

}**catch**(SQLException e){

System.***out***.println("Erro no método getFuncionarios: "+e.getMessage());

}

**return** lista;

}

}

Na classe de atualização, o método privado executa um *query* para alterar os dados dos registros que possuírem o código do CPF passado. Como todos os registros têm os mesmos dados, todos os 5000 registros são alterados. Nesse caso, a classe pública apenas marca os *benchmarks* e executa o método privado, além de instanciar e preencher alguns dos atributos do objeto que representa o cadastro de funcionários. Segue o código da classe.

**import** java.sql.Connection;

**import** java.sql.PreparedStatement;

**import** java.sql.SQLException;

**public** **class** UpdateFuncionarios {

**public** **void** executaUpdateFuncionarios(){

Funcionario funcionario = **new** Funcionario();

funcionario.setCidade("San Francisco");

funcionario.setUf("CA");

funcionario.setCargo("Programador Pleno");

funcionario.setCpf("123.654.789-00");

System.***out***.println("Iniciando teste de update com MySQL...");

**long** tempoInicial = System.*currentTimeMillis*();

updateFuncionarios(funcionario);

**long** tempoFinal = System.*currentTimeMillis*();

String tempoTotal = String.*format*("Tempo total de execução: %.3f segundos%n", (tempoFinal - tempoInicial) / 1000d);

System.***out***.println(tempoTotal);

}

**private** **void** updateFuncionarios(Funcionario funcionario){

**try**{

Connection conn = Conexao.*getConnection*();

StringBuilder qry = **new** StringBuilder();

PreparedStatement ps = **null**;

**int** i = 1;

qry.append(" update funcionarios set nm\_cidade = ?, sg\_uf = ?, nm\_cargo = ? where cd\_cpf = ? ");

ps = conn.prepareStatement(qry.toString());

ps.setString(i++, funcionario.getCidade());

ps.setString(i++, funcionario.getUf());

ps.setString(i++, funcionario.getCargo());

ps.setString(i++, funcionario.getCpf());

ps.execute();

ps.close();

conn.close();

}**catch**(SQLException e){

System.***out***.println("Erro no método updateFuncionarios: "+e.getMessage());

}

}

}

Na classe de deleção, a exemplo da de atualização, o método público chama o privado apenas uma vez passando um dado como parâmetro para que a *query* de exclusão seja executado. Como os dados são os mesmos para todos os registros, a deleção é aplicada em toda a tabela.

**import** java.sql.Connection;

**import** java.sql.PreparedStatement;

**import** java.sql.SQLException;

**public** **class** DeleteFuncionarios {

**public** **void** executaDeleteFuncionarios(){

String cpf = "123.654.789-00";

System.***out***.println("Iniciando teste de delete com MySQL...");

**long** tempoInicial = System.*currentTimeMillis*();

deleteFuncionarios(cpf);

**long** tempoFinal = System.*currentTimeMillis*();

String tempoTotal = String.*format*("Tempo total de execução: %.3f segundos%n", (tempoFinal - tempoInicial) / 1000d);

System.***out***.println(tempoTotal);

}

**private** **void** deleteFuncionarios(String cpf){

**try**{

Connection conn = Conexao.*getConnection*();

StringBuilder qry = **new** StringBuilder();

PreparedStatement ps = **null**;

**int** i = 1;

qry.append(" delete from funcionarios where cd\_cpf = ? ");

ps = conn.prepareStatement(qry.toString());

ps.setString(i++, cpf);

ps.execute();

ps.close();

conn.close();

}**catch**(SQLException e){

System.***out***.println("Erro no método deleteFuncionarios: "+e.getMessage());

}

}

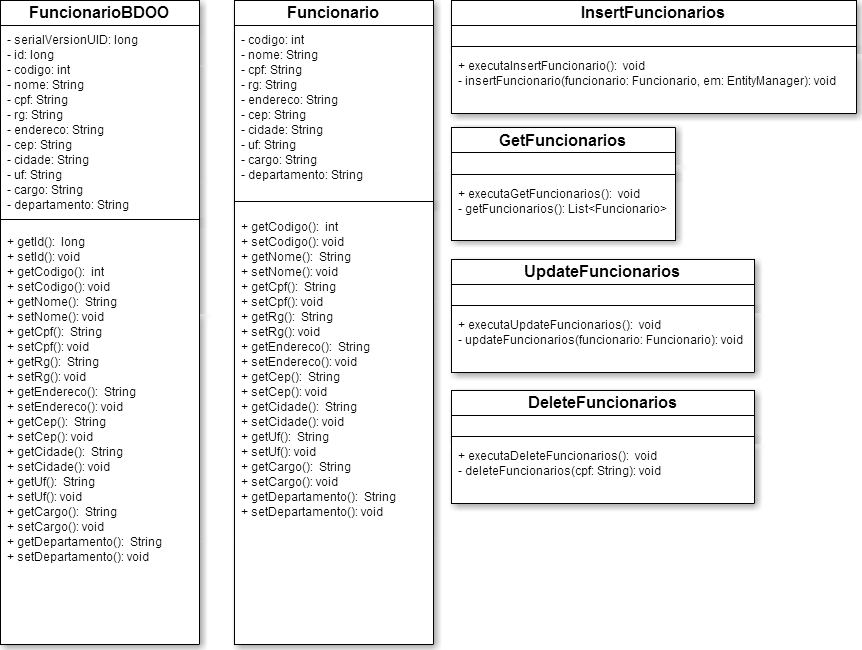
}

Para a execução dos métodos públicos de cada classe, um objeto para cada uma delas foi instanciado na classe principal do projeto e os respectivos métodos foram invocados.

4.4 CLASSES CRIADAS PARA TESTES NO MODELO ORIENTADO A OBJETOS

Para os testes de performance no modelo orientado a objetos, foram usadas as seguintes classes representadas no diagrama da Figura 15.

**Figura 15:** Diagrama de classes para testes no modelo orientado a objetos.



**Fonte:** Elaborada pelo autor.

No projeto criado para os testes do modelo orientado a objetos, em vez de uma tabela para representar registros de funcionários, foi criada uma classe e os objetos dela são gravados diretamente no banco de dados. A estrutura é semelhante a um *bean* como apresentado na sequência.

**import** java.io.Serializable;

**import** javax.persistence.Entity;

**import** javax.persistence.GeneratedValue;

**import** javax.persistence.Id;

@Entity

**public** **class** FuncionarioBDOO **implements** Serializable{

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

@Id @GeneratedValue

**private** **long** id;

**private** **int** codigo;

**private** String nome;

**private** String cpf;

**private** String rg;

//demais campos omitidos

**public** FuncionarioBDOO(){

}

FuncionarioBDOO(Funcionario funcionario){

**this**.codigo = funcionario.getCodigo();

**this**.nome = funcionario.getNome();

**this**.cpf = funcionario.getCpf();

**this**.rg = funcionario.getRg();

//demais campos omitidos

}

**public** **long** getId() {

**return** id;

}

**public** **void** setId(**long** id) {

**this**.id = id;

}

**public** **int** getCodigo() {

**return** codigo;

}

**public** **void** setCodigo(**int** codigo) {

**this**.codigo = codigo;

}

**public** String getNome() {

**return** nome;

}

**public** **void** setNome(String nome) {

**this**.nome = nome;

}

**public** String getCpf() {

**return** cpf;

}

**public** **void** setCpf(String cpf) {

**this**.cpf = cpf;

}

**public** String getRg() {

**return** rg;

}

**public** **void** setRg(String rg) {

**this**.rg = rg;

}

//demais métodos omitidos

}

Por questão de organização e boa prática, foi optado em também criar uma classe *bean* para os atributos do funcionário para que não fosse necessário passar vários parâmetros no método construtor da classe que representa os registros do banco de dados.

A API utilizada para a utilização do *ObjectDB* é a JPA, usando JPQL para construções *queries* como é feito no modelo relacional com o SQL.

A estrutura do projeto e das classes utilizadas nesses testes foi semelhante ao que foi usado nos testes do paradigma relacional.

Na classe de inserção, tem um método privado encarregado de iniciar e finalizar a operação com o banco chamando o método de persistência para gravar um objeto na base de dados. O método público preenche os atributos do *bean*, cria conexão com o banco para que o mesmo não seja feito dentro do *loop* onde é chamado o método privado 5000 vezes. Após, a conexão é fechada e o cálculo dos *benchmarks* é efetuado. A seguir, o código-fonte da classe.

**import** javax.persistence.EntityManager;

**import** javax.persistence.EntityManagerFactory;

**import** javax.persistence.Persistence;

**public** **class** InsertFuncionarios {

**public** **void** executaInsertFuncionario() {

Funcionario funcionario = **new** Funcionario();

funcionario.setNome("Alexandre Marcos");

funcionario.setCpf("123.654.789-00");

funcionario.setRg("12345876");

funcionario.setEndereco("Rua Jorge Tibiriçá");

funcionario.setCep("11055-250");

funcionario.setCidade("Santos");

funcionario.setUf("SP");

funcionario.setCargo("Programador");

System.***out***.println("Iniciando teste de insert com ObjectDB...");

**long** tempoInicial = System.*currentTimeMillis*();

//abrindo conexão e criando o banco se não existir

EntityManagerFactory emf = Persistence.*createEntityManagerFactory*("$objectdb/db/funcionario.odb");

EntityManager em = emf.createEntityManager();

**for** (**int** i = 0; i < 5000; i++) {

funcionario.setCodigo(i);

insertFuncionario(funcionario,em);

System.***out***.println("Funcionário inserido com o código: "+i);

}

// Fechanco conexão

em.close();

emf.close();

**long** tempoFinal = System.*currentTimeMillis*();

String tempoTotal = String.*format*("Tempo total de execução: %.3f segundos%n", (tempoFinal - tempoInicial) / 1000d);

System.***out***.println(tempoTotal);

}

**private** **void** insertFuncionario(Funcionario funcionario, EntityManager em) {

em.getTransaction().begin();

FuncionarioBDOO funcionarioBDOO = **new** FuncionarioBDOO(funcionario);

em.persist(funcionarioBDOO);

em.getTransaction().commit();

}

}

Para abrir ou criar uma nova conexão com o banco, deve passar o diretório em que o BD está ou será criado no método *Persistence.createEntityManagerFactory*. Se o arquivo existir, o acesso será feito. Senão, uma nova base será criada.

Toda transação efetuada com *ObjectDB*, se inicia com *getTransaction().begin()* e termina com *getTransaction().commit()*.

Na classe de seleção dos registros armazenados, o método privado executa uma *query* e o resultado da mesma é armazenado em uma lista de funcionários. No público, o método privado é chamado, a lista é percorrida e todos os 5000 registros são exibidos. Código-fonte apresentado na sequência.

**import** java.util.ArrayList;

**import** java.util.List;

**import** javax.persistence.EntityManager;

**import** javax.persistence.EntityManagerFactory;

**import** javax.persistence.Persistence;

**import** javax.persistence.TypedQuery;

**public** **class** GetFuncionarios {

**public** **void** executaGetFuncionarios(){

System.***out***.println("Iniciando teste de get com ObjectDB...");

**long** tempoInicial = System.*currentTimeMillis*();

List<Funcionario> listaFuncionario = getFuncionarios();

**for** (Funcionario funcionario : listaFuncionario) {

System.***out***.println(funcionario.getCodigo() + " | " + funcionario.getNome() + " | " + funcionario.getCargo() + " | " + funcionario.getCidade());

}

**long** tempoFinal = System.*currentTimeMillis*();

String tempoTotal = String.*format*("Tempo total de execução: %.3f segundos%n", (tempoFinal - tempoInicial) / 1000d);

System.***out***.println(tempoTotal);

}

**private** List<Funcionario> getFuncionarios(){

EntityManagerFactory emf = Persistence.*createEntityManagerFactory*("$objectdb/db/funcionario.odb");

EntityManager em = emf.createEntityManager();

Funcionario funcionario = **null**;

List<Funcionario> lista = **new** ArrayList<Funcionario>();

StringBuilder qry = **new** StringBuilder();

TypedQuery<FuncionarioBDOO> query = **null**;

List<FuncionarioBDOO> results = **null**;

qry.append(" SELECT f FROM FuncionarioBDOO f ");

query = em.createQuery(qry.toString(),FuncionarioBDOO.**class**);

results = query.getResultList();

**for** (FuncionarioBDOO funcionarioBDOO : results) {

funcionario = **new** Funcionario();

funcionario.setCodigo(funcionarioBDOO.getCodigo());

funcionario.setNome(funcionarioBDOO.getNome());

funcionario.setCpf(funcionarioBDOO.getCpf());

funcionario.setRg(funcionarioBDOO.getRg());

funcionario.setEndereco(funcionarioBDOO.getEndereco());

funcionario.setCep(funcionarioBDOO.getCep());

funcionario.setCidade(funcionarioBDOO.getCidade());

funcionario.setUf(funcionarioBDOO.getUf());

funcionario.setCargo(funcionarioBDOO.getCargo());

lista.add(funcionario);

}

em.close();

emf.close();

**return** lista;

}

}

A *String* com o conteúdo da *query* é passado junto com a classe do banco como parâmetro para o método *createQuery* do objeto *EntityManager*. O resultado é armazenado em uma variável *TypedQuery* do tipo da classe do banco. A partir dela, é chamado o método *getResultList* que retorna uma lista do tipo da classe referente ao banco com os resultados da consulta.

A classe de atualização é semelhante ao que foi criado nos testes com *MySQL*. Possui um método privado para a execução da *query* que recebe como parâmetro um objeto com dados preenchidos para efetuar a atualização. Como os dados inseridos são iguais em todos os registros, o valor do CPF, passado na cláusula *WHERE*, faz com que toda a base seja alterada. Segue o código.

**import** javax.persistence.EntityManager;

**import** javax.persistence.EntityManagerFactory;

**import** javax.persistence.Persistence;

**public** **class** UpdateFuncionarios {

**public** **void** executaUpdateFuncionarios(){

Funcionario funcionario = **new** Funcionario();

funcionario.setCidade("San Francisco");

funcionario.setUf("CA");

funcionario.setCargo("Programador Pleno");

funcionario.setCpf("123.654.789-00");

System.***out***.println("Iniciando teste de update com ObjectDB...");

**long** tempoInicial = System.*currentTimeMillis*();

updateFuncionarios(funcionario);

**long** tempoFinal = System.*currentTimeMillis*();

String tempoTotal = String.*format*("Tempo total de execução: %.3f segundos%n", (tempoFinal - tempoInicial) / 1000d);

System.***out***.println(tempoTotal);

}

**private** **void** updateFuncionarios(Funcionario funcionario){

EntityManagerFactory emf = Persistence.*createEntityManagerFactory*("$objectdb/db/funcionario.odb");

EntityManager em = emf.createEntityManager();

em.getTransaction().begin();

StringBuilder qry = **new** StringBuilder();

qry.append(" UPDATE FuncionarioBDOO SET cidade = ");

qry.append("'"+funcionario.getCidade()+"'");

qry.append(" , uf = ");

qry.append("'"+funcionario.getUf()+"'");

qry.append(" , cargo = ");

qry.append("'"+funcionario.getCargo()+"'");

qry.append(" WHERE cpf = ");

qry.append("'"+funcionario.getCpf()+"'");

em.createQuery(qry.toString()).executeUpdate();

em.getTransaction().commit();

em.close();

emf.close();

}

}

No caso de atualização e exclusão de dados, a *query* é executada pelo método *executeUpdate*.

Na classe de deleção, a estrutura também é semelhante, com um método privado recebendo o dado do CPF como parâmetro e este mesmo é usado na cláusula *WHERE* para a exclusão dos registros com esse mesmo valor. Como já explicado anteriormente, toda a base será excluída nessa rotina. Segue o código da classe.

**import** javax.persistence.EntityManager;

**import** javax.persistence.EntityManagerFactory;

**import** javax.persistence.Persistence;

**public** **class** DeleteFuncionarios {

**public** **void** executaDeleteFuncionarios(){

String cpf = "123.654.789-00";

System.***out***.println("Iniciando teste de delete com ObjectDB...");

**long** tempoInicial = System.*currentTimeMillis*();

deleteFuncionarios(cpf);

**long** tempoFinal = System.*currentTimeMillis*();

String tempoTotal = String.*format*("Tempo total de execução: %.3f segundos%n", (tempoFinal - tempoInicial) / 1000d);

System.***out***.println(tempoTotal);

}

**private** **void** deleteFuncionarios(String cpf){

EntityManagerFactory emf = Persistence.*createEntityManagerFactory*("$objectdb/db/funcionario.odb");

EntityManager em = emf.createEntityManager();

em.getTransaction().begin();

StringBuilder qry = **new** StringBuilder();

qry.append(" DELETE FROM FuncionarioBDOO WHERE cpf = ");

qry.append("'"+cpf+"'");

em.createQuery(qry.toString()).executeUpdate();

em.getTransaction().commit();

em.close();

emf.close();

}

}

**5 RESULTADOS DOS TESTES**

Para os testes de performance, foram utilizadas as seguintes APIs: JDBC para *MySQL* e JPA para *ObjectDB*. Em ambos ambientes, foi simulado um cadastro de funcionários com dados como código, nome, endereço, cargo, etc. Sendo assim, foi criada uma tabela no banco relacional e uma classe *Java* para o SGBDOO com campos e atributos referentes a esses dados.

Outro detalhe importante a se destacar é que para deixar os testes idênticos nos dois sistemas, foi usado JPQL para o modelo orientado a objeto devido à possibilidade de se elaborar *queries* mais flexíveis e ter a mesma capacidade que o SQL apresenta.

Um projeto *Java* foi criado para cada um dos bancos e neles foi usado um objeto para popular os dados do funcionário. Esse mesmo objeto é passado como parâmetro na chamada dos métodos referentes às rotinas dos bancos de dados.

Nos dois testes, foi avaliado o desempenho na inserção, seleção, atualização e deleção dos dados. Cada etapa e os resultados serão detalhados na sequência.

Neste trabalho, foi definido o uso de cálculo de média para a medição de performance, a exemplo do que foi realizado no estudo feito por Martins (2012). Ribeiro Filho (2010) cita que é recomendável a determinação de três números chaves: um para o pior caso, outro para o melhor e um terceiro para a média dos tempos colhidos.

5.1 INSERÇÃO

Para avaliar o desempenho de inserção dos SGBDs, todos os dados do funcionário foram passados para o objeto via métodos *getters* e *setters*. Logo depois, foi utilizado um *benchmark* para marcar o tempo inicial do processo. A partir daí, foi aberta uma conexão com o banco e dentro de um *loop*, foi executado o método de inserção dos dados 5000 vezes. Depois disso, foi encerrada a conexão e marcado o tempo final. Após, o tempo de execução foi calculado e exibido em tela. O código dessa etapa foi rodado dez vezes, sempre zerando os registros do banco no término de cada teste, e os resultados foram os seguintes:

**Tabela 2:** Inserção (MySQL) **Tabela 3:** Inserção (ObjectDB)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Execução** | **Tempo** |  |  | **Execução** | **Tempo** |
| 1ª | 9,228s |  |  | 1ª | 14,889s |
| 2ª | 5,912s |  |  | 2ª | 13,031s |
| 3ª | 5,533s |  |  | 3ª | 12,395s |
| 4ª | 5,663s |  |  | 4ª | 12,466s |
| 5ª | 5,660s |  |  | 5ª | 14,294s |
| 6ª | 5,751s |  |  | 6ª | 12,191s |
| 7ª | 5,827s |  |  | 7ª | 12,260s |
| 8ª | 5,832s |  |  | 8ª | 12,310s |
| 9ª | 5,313s |  |  | 9ª | 12,408s |
| 10ª | 5,686s |  |  | 10ª | 12,538s |
| **Média** | **6,041s** |  |  | **Média** | **12,878s** |

**Fonte:** Elaborada pelo autor  **Fonte:** Elaborada pelo autor

**Figura 16:** Média de teste de inserção.

**Fonte:** Elaborada pelo autor.

Conforme demonstrado, a média de tempo do *MySQL* com JDBC foi cerca de duas vezes mais rápido que o *ObjectDB* com JPA, com uma diferença de 6,837 segundos a favor do banco relacional.

5.2 SELEÇÃO

Nos testes de medição de performance na seleção dos dados, foi utilizado um *benchmark* no início da execução. Após isso, uma lista de objetos foi usada para receber o resultado do método que seleciona todos os dados de cada um dos 5000 registros inserido na etapa anterior.

Com essa lista de funcionários, foi feito um *loop* para exibir os atributos código, nome, cargo e cidade de cada objeto retornado. Por fim, o *benchmark* marcador do tempo final e o cálculo dos dois tempos subtraídos. Os resultados, nas tabelas 4 e 5:

**Tabela 4:** Seleção (*MySQL*) **Tabela 5:** Seleção (*ObjectDB*)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Execução** | **Tempo** |  |  | **Execução** | **Tempo** |
| 1ª | 0,699s |  |  | 1ª | 1,001s |
| 2ª | 0,714s |  |  | 2ª | 1,031s |
| 3ª | 0,710s |  |  | 3ª | 2,606s |
| 4ª | 0,722s |  |  | 4ª | 0,923s |
| 5ª | 0,681s |  |  | 5ª | 0,947s |
| 6ª | 0,656s |  |  | 6ª | 0,922s |
| 7ª | 0,655s |  |  | 7ª | 0,851s |
| 8ª | 0,654s |  |  | 8ª | 0,894s |
| 9ª | 0,702s |  |  | 9ª | 1,855s |
| 10ª | 0,697s |  |  | 10ª | 1,007s |
| **Média** | **0,689s** |  |  | **Média** | **1,204s** |

**Fonte:** Elaborada pelo autor  **Fonte:** Elaborada pelo autor

**Figura 17:** Média de teste de seleção.

**Fonte:** Elaborada pelo autor.

Nessa etapa, o *MySQL* com JDBC também se demonstrou mais performático que o *ObjectDB* com JPA. O SGBD relacional foi quase duas vezes mais rápido e com uma diferença média de 0,515 segundo.

5.3 ATUALIZAÇÃO

Para os testes de alteração dos registros, a exemplo da etapa de inserção, foi usado um objeto para receber os dados necessários e esse objeto, passado como parâmetro no método encarregado pela execução da rotina no banco.

Mais uma vez, os *benchmarks* para marcarem o tempo do início e do fim da execução foram utilizados. E o método responsável pela atualização, utilizou os atributos do objeto para alterar os valores dos campos referentes a cidade, UF e cargo nos registros em que o CPF fosse igual ao dado recebido por parâmetro.

Como os dados de todos os registros inseridos foram os mesmos, essa mesma rotina foi capaz de alterar as 5000 linhas gravadas até então. Os resultados dos testes foram:

**Tabela 6:** Atualização (*MySQL*) **Tabela 7:** Atualização (*ObjectDB*)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Execução** | **Tempo** |  |  | **Execução** | **Tempo** |
| 1ª | 0,728s |  |  | 1ª | 0,760s |
| 2ª | 0,749s |  |  | 2ª | 0,701s |
| 3ª | 0,638s |  |  | 3ª | 0,786s |
| 4ª | 0,607s |  |  | 4ª | 0,729s |
| 5ª | 0,612s |  |  | 5ª | 0,757s |
| 6ª | 0,651s |  |  | 6ª | 0,706s |
| 7ª | 0,678s |  |  | 7ª | 0,719s |
| 8ª | 0,655s |  |  | 8ª | 0,873s |
| 9ª | 0,594s |  |  | 9ª | 0,699s |
| 10ª | 0,602s |  |  | 10ª | 0,829s |
| **Média** | **0,592s** |  |  | **Média** | **0,756s** |

**Fonte:** Elaborada pelo autor  **Fonte:** Elaborada pelo autor

**Figura 18:** Média de teste de atualização.

**Fonte:** Elaborada pelo autor.

Os testes de atualização apresentaram uma diferença mais próxima entre os dois bancos e suas respectivas APIs. Mas ainda o *MySQL* junto ao JDBC demonstrou um desempenho superior. A vantagem dessa vez foi de 0,164 segundo.

5.4 DELEÇÃO

Na avaliação de desempenho para deleção nos SGBDs, foi utilizada uma variável com um valor de CPF. Esse dado foi passado como parâmetro em um método para efetuar a exclusão das linhas em que esse valor fosse igual ao campo referente armazenado na base de dados. Assim como na fase anterior dos testes, uma única rotina foi usada para apagar os 5000 registros. Isso ocorre devido ao fato dos dados serem sempre os mesmos.

E como em todas as etapas desses testes, marcadores *benchmarks* foram usados antes e após a execução do método citado. Os resultados dos tempos apresentados foram:

**Tabela 8:** Deleção (*MySQL*) **Tabela 9:** Deleção (*ObjectDB*)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Execução** | **Tempo** |  |  | **Execução** | **Tempo** |
| 1ª | 0,489s |  |  | 1ª | 0,714s |
| 2ª | 0,576s |  |  | 2ª | 0,658s |
| 3ª | 0,476s |  |  | 3ª | 0,805s |
| 4ª | 0,587s |  |  | 4ª | 1,082s |
| 5ª | 0,428s |  |  | 5ª | 0,647s |
| 6ª | 0,513s |  |  | 6ª | 0,669s |
| 7ª | 0,576s |  |  | 7ª | 0,647s |
| 8ª | 0,422s |  |  | 8ª | 0,648s |
| 9ª | 0,582s |  |  | 9ª | 0,617s |
| 10ª | 0,453s |  |  | 10ª | 0,624s |
| **Média** | **0,510s** |  |  | **Média** | **0,711s** |

**Fonte:** Elaborada pelo autor  **Fonte:** Elaborada pelo autor

**Figura 19:** Média de teste de atualização.

**Fonte:** Elaborada pelo autor.

Nos testes de deleção, a diferença também foi mínima. Uma vantagem de 0,201 segundo de tempo médio a favor do *MySQL* com JDBC.

**6 CONSIDERAÇÕES FINAIS**

Trabalhar com banco de dados orientado a objetos pode ser uma solução interessante para desenvolvedores que desejam produtividade e clareza no código em seus projetos. Entretanto, há de se considerar outros quesitos, como performance, na escolha do paradigma do SGBD.

O desempenho de execução desses sistemas pode variar devido a uma série de fatores. Uma dessas variantes, certamente, é a API escolhida.

O site oficial do *ObjectDB* divulga resultados de diversos testes de *benchmarking*, desenvolvidos pela organização *JPA Performance Benchmark* (JPAB), que apontam eles como a solução que apresenta melhor desempenho para se trabalhar com essa API. Inclusive, se comparado com opções do modelo relacional, como o *MySQL*.

Apesar disso, de acordo com os testes de *benchmarking* realizados neste estudo, a alternativa de desenvolver um sistema utilizando *ObjectDB* com JPA não o torna mais performático do que se usar o *MySQL* com a API JDBC.

Com exceção das etapas de inserção e seleção dos dados nos testes feitos, em que a desvantagem de tempo do modelo orientado objeto foi maior, a diferença de performance para atualização e deleção foi menos notável.

Portanto, a escolha de um modelo de banco de dados para um projeto, assim como a opção da API, depende em grande parte do que se deseja no desenvolvimento e na utilização do sistema. Cada alternativa desencadeia em vantagens e desvantagens para o programa.

Dentre os paradigmas de BD e APIs estudadas neste trabalho, se chega à conclusão que se o objetivo for facilidade na codificação, utilização de métodos prontos para persistência de dados e uma melhor integração com a linguagem orientada a objeto, o modelo OO com JPA é a opção mais adequada de acordo com os estudos realizados. Mas se o foco for desempenho e o uso de um padrão mais conhecido para o desenvolvimento, a escolha apropriada é o paradigma relacional junto ao JDBC.

Como sugestão para trabalhos futuros que visam estudar performance de banco de dados, podem ser realizados outros testes em diversas APIs e comparar cada modelo em diferentes cenários, a fim de analisar além do desempenho dessas ferramentas, os recursos que podem oferecer.

**REFERÊNCIAS**

ALECRIM, Emerson. **Banco de dados MySQL e PostgreSQL**. Info Wester. Disponível em <http://www.infowester.com/postgremysql.php>. Acesso em 12 de junho de 2015.

BOSCARIOLI, Clodis et al. **Uma reflexão sobre Banco de Dados Orientados a Objetos**. Universidade Estadual do Oeste do Paraná, 2006.

CAMPOS, Fábio Martinho. **Teste de desempenho: Conceitos, Objetivos e Aplicação – Parte 1**. Linha de Código. Disponível em <http://www.linhadecodigo.com.br/artigo/3256/teste-de-desempenho-conceitos-objetivos-e-aplicacao-parte-1.aspx>. Acesso em 12 de junho de 2015.

CORBELLINI, Anderson et al. **Banco de Dados Orientados a Objetos**. Porto Alegre: Faculdade Dom Bosco de Porto Alegre, 2010.

COSTA, Elisângela Rocha da. **Banco de dados relacionais**. São Paulo: Faculdade de Tecnologia de São Paulo, 2011.

FAIS, Roni Márcio. **Tipos de relacionamento em um banco de dados relacional**. Artigos RMFAIS. Disponível em <http://www.rmfais.com/rmfais/artigos/relatorio.php?titulo=TIPOS%20DE%20RELACIONAMENTOS%20EM%20UM%20BANCO%20DE%20DADOS%20RELACIONALhttp://www.rmfais.com/rmfais/artigos/relatorio.php?titulo=TIPOS%20DE%20RELACIONAMENTOS%20EM%20UM%20BANCO%20DE%20DADOS%20RELACIONAL>. Acesso em 12 de junho de 2015.

GALANTE, Alan Carvalho et al. **Banco de dados orientado a objetos: uma realidade**. Macaé: Faculdade Salesiana Maria Auxiliadora, 2007.

GEOVANE, Hudson. **Entendendo e Aplicando Herança em Java**. DevMedia. Disponível em: < http://www.devmedia.com.br/entendendo-e-aplicando-heranca-em-java/24544>. Acesso em 12 de junho de 2015.

MACÁRIO, Carla Geovana do N.; BALDO, Stefano Monteiro. **O Modelo Relacional**. Campinas: Instituto de Computação da Universidade Estadual de Campinas, 2005.

MARTINS, Carla de Almeida. **Um estudo sobre performance de queries: um comparativo entre banco de dados relacional e orientado a objetos**. Praia Grande: Faculdade de Tecnologia de Praia Grande, 2012.

MEDEIROS, Higor. **Uso de Polimorfismo em Java**. DevMedia. Disponível em <http://www.devmedia.com.br/uso-de-polimorfismo-em-java/26140>. Acesso em 12 de junho de 2015.

MILANI, André. **MySQL – Guia do Programador**. São Paulo: Editora Novatec, 2007.

MYSQL. Disponível em <http://www.mysql.com>. Acesso em 12 de junho de 2015.

OBJECTDB. **Frequently Asked Questions (FAQ)**. Disponível em <http://www.objectdb.com/object/db/database/faq>. Acesso em 12 de junho de 2015.

PINHEIRO, Daniel Ramon Silva et al. **Comparativo entre Banco de Dados Orientado a Objetos (BDOO) e Banco de Dados Objeto Relacional (BDOR)**. Artigo. 2009.

RIBEIRO FILHO, José Tiago. **Entendendo a Performance em Aplicações Java**. Fortaleza: Faculdade Lourenço Filho, 2010.

SERRANO, Guilherme. **Entendendo o relacionamento de tabelas no banco de dados**. O Desenvolvedor. Disponível em <http://odesenvolvedor.andafter.org/publicacoes/entendendo-o-relacionamento-de-tabelas-no-banco-de-dados\_1467.html>. Acesso em 12 de junho de 2015.