UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DEPARTAMENTO ACADÊMICO DE INFORMÁTICA E
DEPARTAMENTO ACADÊMICO DE ELETRÔNICA
BACHARELADO EM ENGENHARIA DE COMPUTAÇÃO

JOCIANE FRANZONI DE LIMA
LUCAS PENHA DE MOURA

# ANÁLISE DE DESEMPENHO SOBRE CAMADAS DE PERSISTÊNCIA EM BANCO DE DADOS RELACIONAL E NÃO-RELACIONAL

TRABALHO DE CONCLUSÃO DE CURSO

CURITIBA
2017

JOCIANE FRANZONI DE LIMA
LUCAS PENHA DE MOURA

# ANÁLISE DE DESEMPENHO SOBRE CAMADAS DE PERSISTÊNCIA EM BANCO DE DADOS RELACIONAL E NÃO-RELACIONAL

Trabalho de Conclusão de Curso do Curso Superior de Engenharia de Computação do Departamento Acadêmico de Eletrônica -DAELN- e do Departamento Acadêmico de Informática -DAINF- da Universidade Tecnológica Federal do Paraná -UTFPR, como requisito parcial para obtenção do título de Bacharel em Engenharia.

Orientador: Paulo Roberto Bueno
Coorientador: Leandro Batista De Almeida

CURITIBA
2017

## RESUMO

LIMA, J. F.; MOURA, L. P. *Análise de Desempenho Sobre Camada de Persistência em Banco de Dados Relacional e Não Relacional*. Trabalho de conclusão de curso, 2017.

Os bancos de dados não-relacionais (NoSQL) vieram ao mercado como solução para problemas que necessitam de grande escalabilidade e disponibilidade em banco de dados. Este paradigma de armazenamento de dados é capaz de proporcionar grande desempenho em armazenamento de dados em grande escala. É por esta razão que vem crescendo muito o uso deste tipo de banco de dados em aplicações corporativas. Identifica-se atualmente uma demanda de aplicações que testem a viabilidade de migração de sistemas de um banco de dados relacional para um NoSQL e com isso verificar o desempenho em termos de tempo de realização de operações sobre os dois paradigmas distintos, verificando a vantagem ou não desta migração, além da análise de complexidade de implementação para esta migração. A motivação para a realização deste projeto é auxiliar a comunidade desenvolvedora a verificar a eficiência e facilidade de migração entre bancos com paradigmas distintos de armazenamento sem que tenha necessidade de reescrita do código, visando economia de tempo e custos. O objetivo deste projeto é testar a camada de persistência para bancos de dados relacionais e não-relacionais, visando identificar a viabilidade da migração entre bases de dados em sistemas legados. O projeto utilizou dois bancos de dados específicos, MySQL (relacional) e MongoDB (NoSQL) executando sobre uma arquitetura centralizada. A metodologia utilizada neste projeto foi a espiral, na qual repete-se os passos de pesquisa, implementação, testes e análise, até que os resultados finais sejam os especificados. Foi desenvolvido inicialmente uma aplicação destinada para executar sobre um banco de dados relacional, como a maior parte dos sistemas corporativos existentes, utilizando camada de persistência para realizar as operações sobre este banco. Após esta etapa foi realizado a conexão com um banco de dados NoSQL, sem modificar o código do software. A única modificação realizada foi na camada de persistência para realizar operações sobre o novo banco. Ao verificar que a camada de persistência consegue lidar com os dois paradigmas de armazenamento de dados totalmente distintos, foram realizados testes para verificar se além de suportar esta migração a camada de persistência é capaz ou não de manter um desempenho satisfatório,com isto foi feita uma análise da extração de resultados destes testes para também verificar a vantagem ou não desta migração por meio de camadas de persistência dando enfoque no desempenho.



**Palavras-chaves**: *Banco de dados, camada de persistência, relacional, Não relacional, testes de viabilidade, testes de desempenho*

## ABSTRACT

LIMA, J. F.; MOURA, L. P. *Performance Analisys of the Persistence Layer in Relational and Non-Relational Databases*. BSc. Thesis, 2017.

Non-relational databases emerged as a solution to problems that need great scalability and availability to store data. This data storage paradigm is capable to provide great performance in data storage large scale. For this reason, the use of this paradigm has grown a lot in corporate applications. Nowadays there is a demand for systems migrations feasibility testing applications from relational to non-relational databases to verify the performance of the two distinguished data storage paradigm, verifying the advantage of this migration, besides the feasibility of implementation time and complexity for this migration. The motivation to execute this project is to help the developers community to verify the efficiency ans easiness of the migration between databases with different data store paradigms without the need to rewrite the source code, aiming to save time and costs. The objective of this project is to test the persistence layer for the relational and non-relational databases to identify the viability of migration between database in legacy systems. The methodology used in this project was the spiral, in which repeats the search, implementation, tests and analysis steps until the results are as specified. An application was initially developed to run on a relational database, such as most existing corporate systems, using the persistence layer to perform operations in the initial database. After this step, was performed a connection with a non-relational database without modifying the software code. The only change was made in the persistence layer, so it can perform operations in the new database. When verifying that the persistence layer is able to handle the two storage paradigm, tests were performed to verify if the persistence layer is able to maintain a satisfactory performance with the new database. An analisys was made with the data generated in the tests to verify the viability to migrate databases with diferents paradigms only changing the persistence layer with a focus on performance.



**Key-words**: Database, Persistence Layer, Relational Databases, Non-relational Databases, Viability tests, Performance tests.

## LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| ACID | *Atomicity, Consistency, Isolation, Durability.* |
| CRUD | *Create, Read, Update and Delete.* |
| DBM | *Data Base Manager* |
| JPA | *Java Persistence API.* |
| JPQL | *Java Persistence Query Language.* |
| NoSQL | *Not Only SQL.* |
| ORM | *Object Relational Mapping.* |
| OGM | *Object Grid Mapping.* |
| SGBD | Sistema Gerenciador de Banco de Dados. |
| SQL | *Structured Query Language* |

## LISTA DE ILUSTRAÇÕES

## LISTA DE TABELAS

**SUMÁRIO**

# 1 INTRODUÇÃO

## 1.1 MOTIVAÇÃO E JUSTIFICATIVA

A principal motivação para a realização deste projeto é auxiliar a comunidade desenvolvedora a verificar a eficiência e facilidade de migração entre bancos de dados com paradigmas distintos de armazenamento sem que haja a necessidade de reescrita de código, visando economia de tempo e custos.

A camada de persistência é o elemento que faz a conexão entre a aplicação e o banco de dados, ou seja, é através dela que os dados são explorados no banco, seja inserção, consulta, atualização ou remoção.

Os gerenciadores de bancos de dados relacionais, como Oracle, Microsoft SQL e MySQL são os mais populares, porém o uso de bancos não-relacionais como o MongoDB e Apache Cassandra (LAKSHMAN; MALIK, 2009), DynamoDB (DECANDIA *et al.*, 2007) e o BigTable (CHANG *et al.*, 2006) já aparecem como bancos bastante utilizados, de acordo com o DB-Engine (DB-Engine, 2017). A partir destes dados deve-se a escolha dos bancos utilizados. O MySQL é o banco de dados *open source* mais utilizado e o MongoDB, mesmo sendo um banco NoSQL, é o quarto mais usado dentre os demais bancos existentes.

Como mostrado pelo *market research media* (MARKET..., 2017) o uso de banco de dados não-relacionais irá crescer 21% até 2020, porém existe uma falta de ferramentas e publicações sobre o tema de migração entre bancos relacionais e não-relacionais. Este projeto justifica-se, pois pretende mostrar a viabilidade da migração sem que haja nenhuma mudança nas camadas acima da camada de persistência da aplicação, realizando para isto testes que verifiquem se a camada de persistência realmente é capaz de alternar entre paradigmas de armazenamento com conceitos bem distintos e também testar o desempenho desta sobre os dois bancos levando em consideração acessos múltiplos ao sistema.

## 1.2 OBJETIVOS

Nesta seção serão descritos os objetivos a serem alcançados com a implementação deste trabalho.

### 1.2.1 Objetivo Geral

O objetivo deste projeto é realizar uma análise de desempenho de camadas de persistências sobre bancos de dados relacionais e não-relacionais.

### 1.2.2 Objetivos Específicos

- Desenvolver aplicação que realize operações de inserção, busca, atualização e remoção (CRUD) sobre um banco de dados relacional, que serve como modelo para testar a portabilidade entre bancos de paradigmas diferentes;
- criar aplicação para geração de dados fictícios para preenchimento do banco de teste;
- desenvolver uma metodologia para testes da camada de persistência da aplicação executando sobre base relacional e NoSQL;
- extrair dados dos testes sobre a camada de persistência;
- analisar resultados dos testes para concluir sobre o desempenho de camadas de persistência, visando uma interpretação clara e precisa dos dados extraídos.

## 1.3 APRESENTAÇÃO DO DOCUMENTO

Neste documento são apresentados os conceitos para o desenvolvimento deste projeto, como os conceitos de bancos de dados relacionais e não relacionais e também conceitos sobre camadas de persistência, que serão apresentados no capítulo 2. O capítulo 3 trás o fluxo de desenvolvimento deste trabalho, desde o projeto até sua execução e a metodologia utilizada, que inclui o escopo do projeto. No capitulo 4 serão apresentados os resultados obtidos nos testes realizados através de tabelas e gráficos que traduzem de forma clara os dados obtidos. Por fim, são apresentadas no capítulo 5, as conclusões do projeto.

# 2 FUNDAMENTAÇÃO TEÓRICA

Nesta seção serão descritos os conceitos que foram utilizados para a realização deste trabalho de conclusão de curso.

## 2.1 BANCO DE DADOS

Base ou banco de dados são estruturas que possibilitam o armazenamento de dados de forma segura. Estrutura esta que permite também o gerenciamento e recuperação dos dados (DATE, 2004). Esta estrutura necessita de uma forma de gerenciamento e, para isso, tem-se o software gerenciador de banco de dados (SGDB).

### 2.1.1 Sistema de Banco de Dados

O sistema de banco de dados é composto pelo software de gerenciamento, explicado na subseção 2.1.2, e o banco de dados em si. A Figura 1 apresenta uma representação da estrutura de um banco de dados e sua interação com os usuários que a acessam.

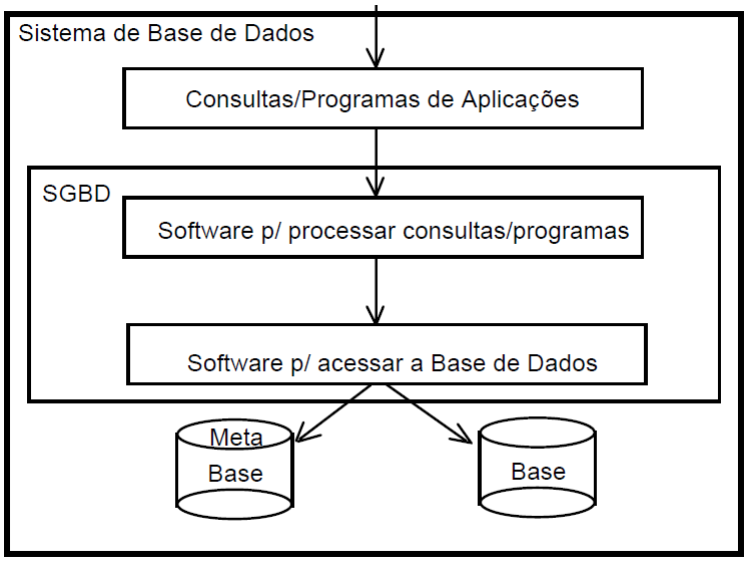



**Figura 1 – Sistema de Banco de Dados**

**Fonte: (TAKAI; ITALIANO; FERREIRA, 2005).**

### 2.1.2 Sistema Gerenciador de Banco de Dados

O sistema gerenciador de banco de dados pode ser definido como um software capaz de gerenciar funcionalidades de criação, exclusão, modificação e recuperação (CRUD) em um banco de dados (HAUSER, 2000), e também gerenciar quesitos como segurança e integridade dos dados armazenados. Integridade significa manter uma relação coesa e que reflete a realidade do que foi representado no banco de dados e que por si são consistentes (DATE, 2000).

Atualmente os bancos podem ser separados em duas classes de paradigmas: *structured query language* (SQL) e *not only structured query language* (NoSQL). As definições destes paradigmas são descritas nas subseções 2.2.1 e 2.2.2, respectivamente.

## 2.2 PARADIGMAS DE BANCO DE DADOS

Paradigmas de bancos de dados são modelos que determinam a estrutura lógica de um banco, ou seja, determina a maneira que os dados são armazenados, manipulados e organizados. Existem diversos paradigmas de armazenamento e organização de dados, o mais comum é o paradigma relacional, porém existem outros como o modelo NoSQL que abrange inúmeras formas de armazenamento e organização distintas do modelo relacional.

### 2.2.1 Relacionais (SQL)

Bancos de dados relacionais são estruturas em que os dados, como o nome diz, possuem relacionamentos entre si. Estes dados são armazenados em tabelas, e essas tabelas possuem colunas com as informações que se deseja. Por exemplo, ao criar uma tabela *user* e uma tabela *address*, deseja-se que cada novo registro contenha as seguintes informações:

- User
  - id do usuário
  - nome
  - endereço
- Address
  - id do endereço
  - cidade

Essas tabelas, com suas respectivas colunas, são representadas através de entidades (TAKAI; ITALIANO; FERREIRA, 2005). O *Java Persistence API* (JPA, descrito na seção 2.5), através de anotações utiliza essas entidades para gerenciar os bancos de dados.

O conceito de relacionamento é melhor visualizado na prática. Utilizando as entidades *user* e *address* e seus atributos, suas representações no banco de dados relacional são como mostrado na Figura 2.

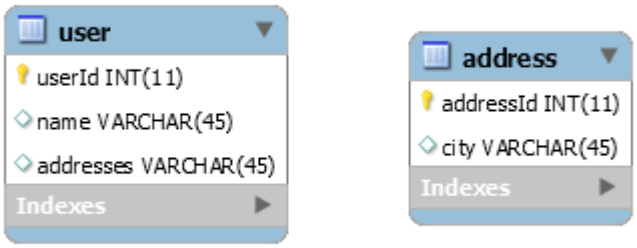



**Figura 2 – Tabelas de Um Sistema de Banco de Dados Relacional**

**Fonte: Autoria Própria.**

Ao observar as duas entidades é possível notar o relacionamento entre elas, pois cada usuário que for inserido na tabela *user* deve ter um endereço. Mas a relação entre estas duas entidades pode ser diferente, por exemplo, um usuário pode ter mais de um endereço, ou um endereço pode ter diversos usuários.

Desta forma os bancos de dados relacionais possuem três tipos de relações entre entidades. A relação *1x1* (um para um), a relação *1xN* (um para N) e a relação *NxM* (N para M).

Em uma relação um para um, seguindo o exemplo do usuário e endereço, um usuário possui apenas um endereço e este endereço possui apenas um único usuário. A Figura 3 mostra a representação gráfica de uma relação *1x1* entre duas entidades.

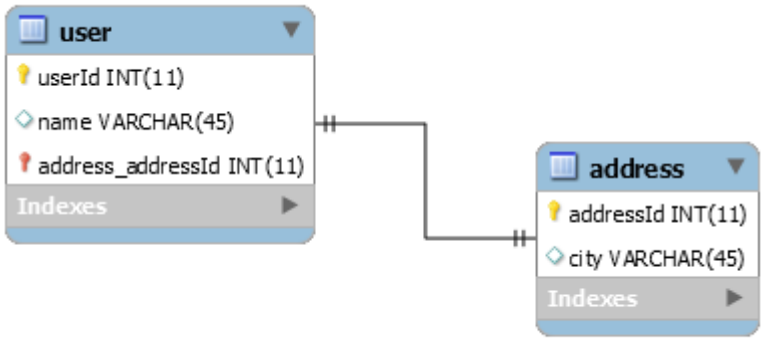



**Figura 3 – Relação *1x1***

**Fonte: Autoria Própria.**

Na relação um para *N* uma entidade possui relação com diversas instâncias de uma segunda entidade. Por exemplo, um endereço pode ter inúmeros usuários, porém cada usuário pode estar em apenas um endereço. A Figura 4 apresenta esta relação.

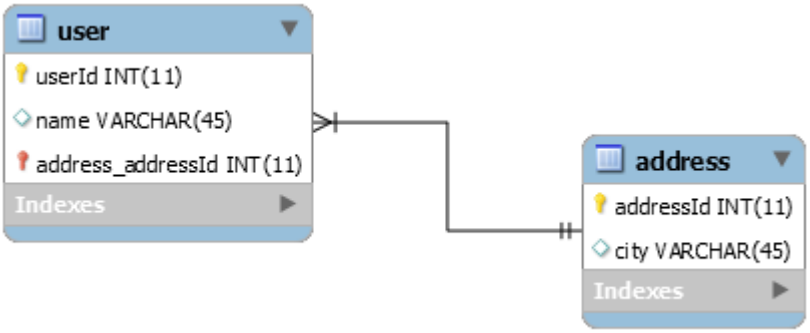



**Figura 4 – Relação *1xN***

**Fonte: Autoria Própria.**

Por fim, a relação *N* para *M* representa uma relação de muitos para muitos, isso quer dizer que uma instância de uma entidade tem inúmeras relações com instâncias de uma segunda entidade, e vice versa. No exemplo dado um usuário pode possuir um ou mais endereços, assim como um endereço pode ter diversos usuários. Essa relação é mostrada na Figura 5

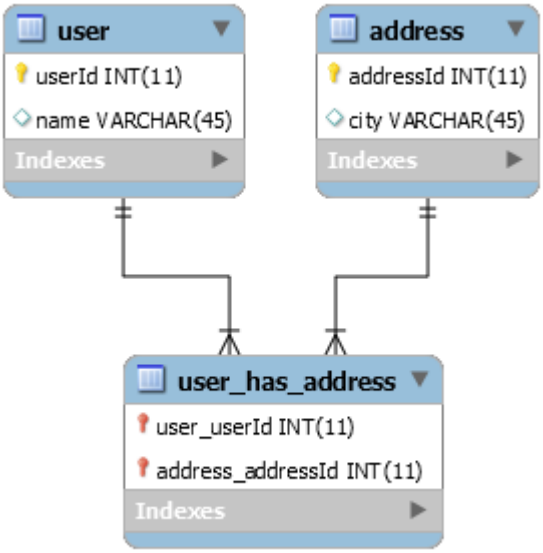



**Figura 5 – Relação *NxM***

**Fonte: Autoria Própria.**

Pode-se notar que uma tabela auxiliar é criada para armazenar todos os registro de relações entre as duas entidades, ou seja, esta tabela contém todos os registros de usuários que contém um endereço e todos os registros de endereço que contém usuários.

Nas Figuras 3, 4 e 5 a relação fica evidenciada pela linha que liga as duas tabelas. Esta notação, chamada *crows foot*, é bastante utilizada para representar a cardinalidade de uma

relação no banco de dados. Em cada ponta existe uma representação que indica a quantidade de instâncias daquela entidade. Essa notação é apresentada na Figura 6.

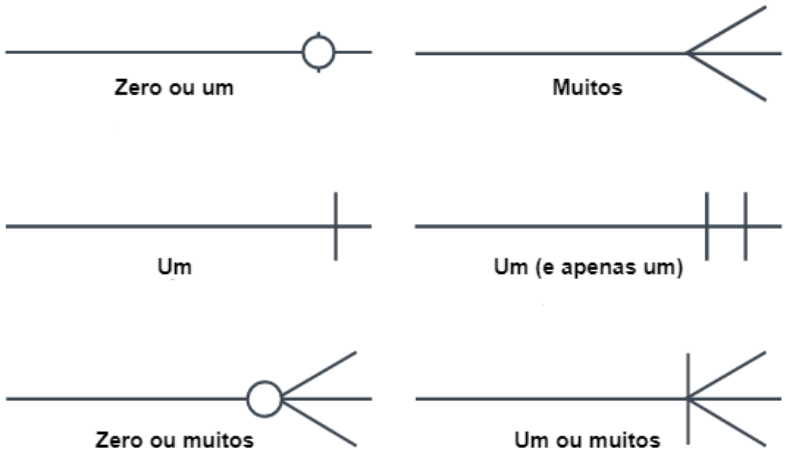
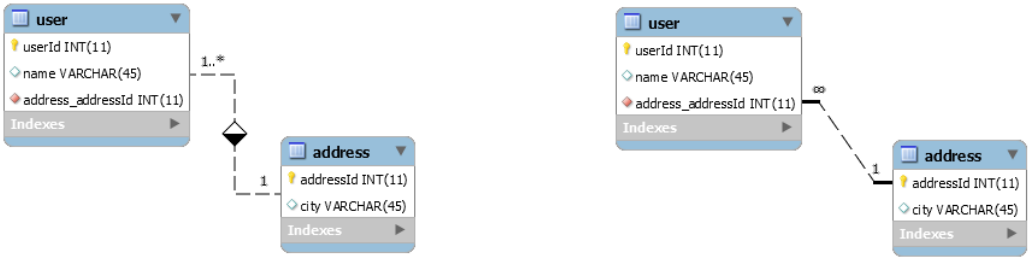



**Figura 6 – Representação das Relações *Crows Foot***

**Fonte: Autoria Própria.**

Existem diversas outras notações para representar as relações entre tabelas, como o modelo de Peter Chen (CHEN, 1976), apresentado na Figura 7a, e o modelo *connect to columns*, apresentado na Figura 7b.




**(a) Modelo de Peter Chen** **(b) Modelo *connect to columns***

**Figura 7 – Notações de Relacionamento de Tabelas**

**Fonte: Autoria Própria.**

As notações apresentadas na Figura 7 tem o mesmo significado que a notação apresentada na Figura 4, ou seja, representam um relacionamento um para muitos. Neste trabalho as relações são todas apresentadas no modelo *crows foot*.

### 2.2.2 Não-Relacionais (NoSQL)

Banco de dados não-relacionais são citados como banco de dados NoSQL. Este tipo de banco surgiu como uma forma de tentar solucionar problemas de escalabilidade e armazenamento em casos em que se utilizam um grande volume de dados.

Existem diferentes estruturas de bancos de dados NoSQL. As principais estruturas são apresentadas abaixo:

- **Chave - valor**

  Neste modelo o endereçamento de dados é realizado através de uma única chave. A única maneira de guardar os dados é através de chaves para cada valor. Isso se deve porque os valores neste modelo são uma sequência de *bytes* e não representam significado. Este tipo de armazenamento de dados se assemelha a um dicionário em que seus dados são endereçados através de uma chave única.

  Os valores armazenados são completamente isolados e independentes, devido a isto as relações entre os dados se dão apenas através da aplicação. A utilidade deste modelo foca-se em aplicações simples que são baseadas em indexação. A principal vantagem do modelo chave-valor são as pesquisas que, em geral, são rápidas (HECHT; JABLONSKI, 2011).

  Foi observado a partir de 2009 que bancos de dados não relacionais poderiam se tornam uma alternativa aos bancos SQL existentes (HECHT; JABLONSKI, 2011), isso se deve à referência de seus expoentes, o Amazon DynamoDB (DECANDIA *et al.*, 2007) e o Google BigTable (CHANG *et al.*, 2006).

- **Orientado a colunas**

  Os dados, neste modelo, não são armazenado em linhas, como em uma tabela, mas sim armazenados em colunas separadas a cada registro.

  As informações não são interpretadas pelo banco de dados, sendo tratadas pela aplicação. Para facilitar a organização do banco as colunas podem ser agrupadas em famílias de colunas, com a possibilidade de colunas serem incluídas a qualquer momento nestas famílias (HECHT; JABLONSKI, 2011). Um exemplo deste tipo de banco é o Apache Cassandra (APACHE..., 2017a).

- **Banco de dados em grafos**

  O modelo orientado a grafos é especializado no gerenciamento de dados fortemente conectados.

  Aplicações que contém muito relacionamento de dados são bem adequadas para este tipo de banco devido ao baixo custo para realização de pesquisas contendo muitas conexões. Este modelo, como o nome diz, permite que a base tenha uma representação em forma de

grafo. A organização dos dados é gerida pelo sistema que constrói um grafo em que cada nó contém um par de chave-valor.

Uma boa aplicação do modelo em questão é encontrar caminhos em sistemas de navegação, isso devido ao fato da facilidade de deslocamento entre os nós (HECHT; JABLONSKI, 2011). São bancos de dados orientado a grafos o AlegroGraph (ALEGROGRAPH, 2017) e o ArangoDB (ARANGODB, 2017).

- **Orientado a documentos**

  Este modelo é baseado em organização de dados por estrutura de documentos. Os documentos são estruturas de dados com estrutura de árvores hierárquicas, sendo capazes de realizar uma representação de coleções, mapas e outros tipos de objetos.

  Os documentos são coleções de atributos e valores, em que um atributo pode ser multivalorados. Em bancos de dados orientados a documentos os documentos armazenados não necessitam ter estrutura em comum. Cada documento possui um identificador único denominado de ID para que ele possa ser identificado em uma coleção.

  Os documentos não fornecem relações entre si dando a este tipo de banco a característica de ser livre de esquemas, pois integram dados relacionados com outros dados ao próprio documento sem que haja a necessidade de armazenar tais dados relacionados em uma outra área de armazenamento a parte.

  Exemplos deste tipo de bancos de dados são o MongoDB, *open source*, gerenciado pela 10gen e também o CouchDB gerido pela Apache (APACHE..., 2017b).

### 2.2.3 Relacional x Não-Relacional

Para efeito de comparação entre paradigmas de banco de dados, devem-se destacar alguns pontos importantes:

- **Escalabilidade**

  Bancos de dados relacionais e não-relacionais tem vantagens e desvantagens com relação ao seu uso. Podem-se citar alguns pontos importantes na comparação entre os dois paradigmas de banco de dados.

  Banco de dados relacionais são de difícil escalabilidade, os sistemas podem ser classificados com base em dois tipos de escalabilidade, vertical e horizontal, vertical ocorre quando adiciona-se mais hardware ao sistema, por exemplo, aumento de memória e processamento.

  A escalabilidade horizontal ocorre quando adiciona-se mais nós ao sistema, aumentando o número de computadores atuando na aplicação, como, por exemplo, uma clusterização do sistema, sendo muito útil em aplicações distribuídas.

Quando iniciou-se o desenvolvimento de banco de dados relacionais, por serem pioneiros não foram projetados para serem escaláveis, não pensando em sistemas distribuídos, como por exemplo aplicações Web que são muito abrangentes atualmente.

Bancos NoSQL surgiram com o intuito de atender essa demanda, seguindo assim uma escalabilidade horizontal e tendo grande abrangência distribuída (MOHAMED; ALTRAFI; ISMAIL, 2014).

- **Consistência de dados**

  Bancos relacionais são bem difundidos devido ao alto grau de confiança em suas operações, tendo em vista a consistência dos dados. Bancos relacionais tem grande confiabilidade, pois seguem o paradigma ACID, o qual força a consistência em cada operação realizada sobre os dados.

  Bancos NoSQL possuem consistência de dados eventual que é definida quando o sistema de armazenamento garante que, se nenhuma nova operação for realizada, todos os acessos a esse dado retornarão o último valor atualizado. Na ocorrência de modificação já não pode-se garantir que todos os demais processos que estão acessando estes dados terão o valor devidamente atualizado (BRITO, 2010).

- **Disponibilidade** Como citado anteriormente bancos NoSQL seguem a escalabilidade horizontal, assim disponibilizando os dados de forma paralela, o que aumenta a disponibilidade para acessos, garantindo também que ao ocorrer, por exemplo, a queda de um servidor desses dados, não implique na indisponibilidade de acesso.

  Bancos relacionais possuem escalabilidade vertical e são mais vulneráveis a grande quantidade de requisições de acessos, visto que não foram feitos para acesso distribuído (LEITE, 2010).

A Tabela 1 apresenta de forma sucinta a comparação entre paradigmas de banco de dados relacionais e não-relacionais.

**Tabela 1 – Análise Comparativa Relacional X Não relacional**

| | **Relacional** | **NoSQL** |
|---|---|---|
| **Escalabilidade** | Possível, mas complexo. Devido à natureza estruturada do modelo, a adição de forma dinâmica e transparente de novos nós no grid não é realizada de modo natural. | Uma das principais vantagens desse modelo. Por não possuir nenhum tipo de esquema pré-definido, o modelo possui maior flexibilidade o que favorece a inclusão transparente de outros elementos. |
| **Consistência** | Ponto mais forte do modelo relacional. As regras de consistência presentes propiciam um maior grau de rigor quanto à consistência das informações. | Realizada de modo eventual no modelo: só garante que, se nenhuma atualização for realizada sobre o item de dados, todos os acessos a esse item devolverão o último valor atualizado. |
| **Disponibilidade** | Dada a dificuldade de se conseguir trabalhar de forma eficiente com a distribuição dos dados, esse modelo pode não suportar a demanda muito grande de informações do banco. | Outro fator fundamental do sucesso desse modelo. O alto grau de distribuição dos dados propicia que um maior número de solicitações aos dados seja atendida por parte do sistema e que o sistema fique menos tempo não-disponível. |

**Fonte: (BRITO, 2010).**

## 2.3 SOFTWARES GERENCIADORES DE BANCO DE DADOS

Os softwares genrenciadores de banco de dados são as soluções de gerenciamento encontradas no mercado, como o MySQL, Oracle, Postgres no paradigma relacional e MongoDB, Cassandra, DynamoDB no paradigma NoSQL. As seções 2.3.1 e 2.3.2 apresentam mais detalhes sobre o MySQL e sobre o MongoDB, respectivamente.

### 2.3.1 MySQL

MySQL é um banco de dados com paradigma relacional possuindo versão paga e versão gratuita, que foi criado pela empresa MySQL AB. Inicialmente foi construído visando ser utilizado para pequenas e médias aplicações, porém atualmente é capaz também de abranger

projetos de grande porte, sendo o banco de paradigma relacional *open souce* mais utilizado da atualidade.

Muito utilizado por ser multiplataformas podendo ser utilizado nos mais diversos sistemas operacionais. Além de ter o poder de comunicação e integração com diversas linguagens de programação, como Java, C#, C/C++, entre outras.

Por se tratar de um gerenciador de banco de dados relacional, os dados são inseridos em tabelas, em que as colunas representam as informações pertinentes a um registro, e cada registro é adicionado em uma linha. A Figura 8 apresenta alguns registros armazenados em forma de tabela.

| userId | name | addressId |
|---|---|---|
| 1 | João | 4 |
| 2 | Maria | 2 |
| 3 | Tiago | 1 |
| 4 | Ana | 3 |

**(a) Representação dos Dados da Tabela *User*.**

| addressId | city |
|---|---|
| 1 | Curitiba |
| 2 | São Paulo |
| 3 | Ribeirão Preto |
| 4 | Rio de Janeiro |

**(b) Representação dos Dados da Tabela *Address*.**

**Figura 8 – Representação dos Dados em Tabela no MySQL.**

**Fonte: Autoria Própria.**

O MySQL foi o gerenciador de banco de dados relacional escolhido por ser *open source* e um dos gerenciadores mais usados atualmente no mercado. Segundo *DB-Engines Ranking* (DB-Engine, 2017) é o segundo gerenciador mais utilizado, ficando atrás apenas do Oracle.

### 2.3.2 MongoDB

Este trabalho foi fundamentado sobre um banco de dados NoSQL orientado a documentos. O SGBD escolhido foi o MongoDB, devido a seu grande uso no mercado atual, de acordo com o *DB-Engines Ranking* (DB-Engine, 2017), o MongoDB encontra-se em quinto lugar como banco de dados mais utilizado e é o primeiro NoSQL da lista.

Este tipo de estrutura de banco de dados utiliza em sua representação e estruturação de coleções de atributos e valores (DIANA; GEROSA, 2010).

A Figura 9 apresenta como seria representado em NoSQL orientado a documentos (Figura 9b) um sistema de banco de dados (9a).

| userId | name | addressId |
|---|---|---|
| 3 | Tiago | 1 |

| addressId | city |
|---|---|
| 1 | Curitiba |

**(a) Tabela em Banco de Dados Relacional**

```
{
    "userId"    : 1,
    "name"      : "João",
    "address"   : "Curitiba"
}

{
    "addressId" : 1,
    "city"      : "Curitiba"
}
```

**(b) Mapeamento em Banco de Dados NoSQL Orientado a Documentos**

**Figura 9 – Representação em Documento de Uma Tabela Relacional**

**Fonte: (STöRL *et al.*, 2015).**

Os documentos utilizados pelo MongoDB são semelhantes ao JSON e os dados são armazenados utilizando-se um par de chave/valor. O MongoDB é um banco de dados escrito em C++ e é um projeto de software livre. A linguagem de pesquisa do MongoDB baseada em documentos o distingue dos demais, tornando assim a migração de bases MongoDB para bases relacionais com grande facilidade, pois suas consultas são de fácil conversão, sendo também simples a conversão de instruções em SQL para chamadas de funções de pesquisa no MongoDB (IBM..., 2017).

## 2.4 CAMADAS DE PERSISTÊNCIA

Persistência, em computação, significa uma forma de se manter os dados mesmo após o escopo de uma aplicação ter finalizado, ou seja, todos os dados manipulados pelo contexto de uma aplicação devem ser mantidos de forma permanente.

A camada de persistência é a parte do sistema que faz o controle e gerência do sistema de banco de dados. Ela é responsável por armazenar e manipular dados do banco (BAUER; KING, 2007).

Existem diversos *frameworks* de camada de persistência, esta seção irá detalhar o *framework* Hibernate de persistência de dados.

### 2.4.1 Hibernate

O Hibernate trata-se de um *framework* que executa sobre a linguagem Java, que serve para facilitar o desenvolvimento de aplicações que utilizam bancos de dados. Inicialmente o Hibernate foi desenvolvido para mapeamento objeto-relacional, o seu principal objetivo é trans-

formar as descrições orientadas a objetos Java para a interpretação sobre um sistema de banco de dados (BAUER; KING, 2007).

Em termos práticos, o Hibernate trata-se de um conjunto de arquivos que realizam a configuração sobre a aplicação, que executam um determinado conjunto de classes e interfaces, as quais permitem a criação de camadas de serviços, que abstraem funcionalidades sobre um sistema de banco de dados (MIILLER; BONNETI, 2015). Hibernate é um software livre de código aberto distribuído com a licença LGPL.

### 2.4.2 Hibernate ORM

O Hibernate ORM foi criado com a finalidade de auxiliar no mapeamento objeto-relacional em aplicações desenvolvidas sobre sistemas de banco de dados relacionais de forma transparente, ou seja, o Hibernate abstrai toda a parte de comunicação entre a aplicação e o banco de dados (HIBERNATE..., 2017c).

O Hibernate ORM é capaz de persistir dados utilizando o conceito de orientação a objetos, incluindo herança. O ORM permite também, mapear objetos em tabelas de sistemas com bancos relacionais. Quando mapeia-se dados no modelo relacional cria-se o efeito de orientação a objetos sobre o banco, fazendo com que o desenvolvedor não precise se preocupar com a disposição dos dados nas tabelas e então possa focar apenas na manipulação de objetos e nas regras de negócios da aplicação (BAUER; KING, 2007).

### 2.4.3 Hibernate OGM

O Hibernate OGM é a versão do *framework* capaz de realizar o mapeamento de entidades em sistemas de banco de dados não-relacionais. Com ele é possível o acesso a diversos bancos não relacionais utilizando os mesmos conceitos em todos, sem a preocupação de mudar a sintaxe de cada banco (HIBERNATE..., 2017a) utilizando a *Java Persistence API*.

O que o Hibernate OGM de fato faz é estender toda a estrutura existente do Hibernate ORM para o mapeamento de entidades em documentos do MongoDB, em chave-valor como o Infinispan ou em grafos como o Neo4j (DEBNATH, 2016). O OGM herda do Hibernate ORM diversas características como a abstração entre modelo de objetos da aplicação e o modelo de persistência de dados do banco e também mantém a característica de chave primária para interfaceamento com as entidades. Os dados são persistidos como tipos básicos e o OGM ainda mantém o conceito de chave estrangeira para relacionar duas entidades.

A versão OGM do Hibernate persiste dados em bancos não-relacionais através de interfaces que realizam a comunicação entre a aplicação e o banco de dados que se deseja utilizar.

A arquitetura básica de acesso a bancos não relacionais pode ser vista na Figura 10. Nela é mostrado que a aplicação envia uma requisição para o JPA ou qualquer outra API do Hibernate e esta por sua vez utiliza o *Hibernate ORM Core* para delegar tarefas ao Hibernate OGM. Nota-se que apenas neste nível é utilizado o *core*, dois níveis abaixo da aplicação, do Hibernate OGM. Neste ponto o OGM utiliza o dialeto específico do banco de dados e utiliza a *query engine* para analisar e fazer a tradução das *queries* para a linguagem do banco utilizado.

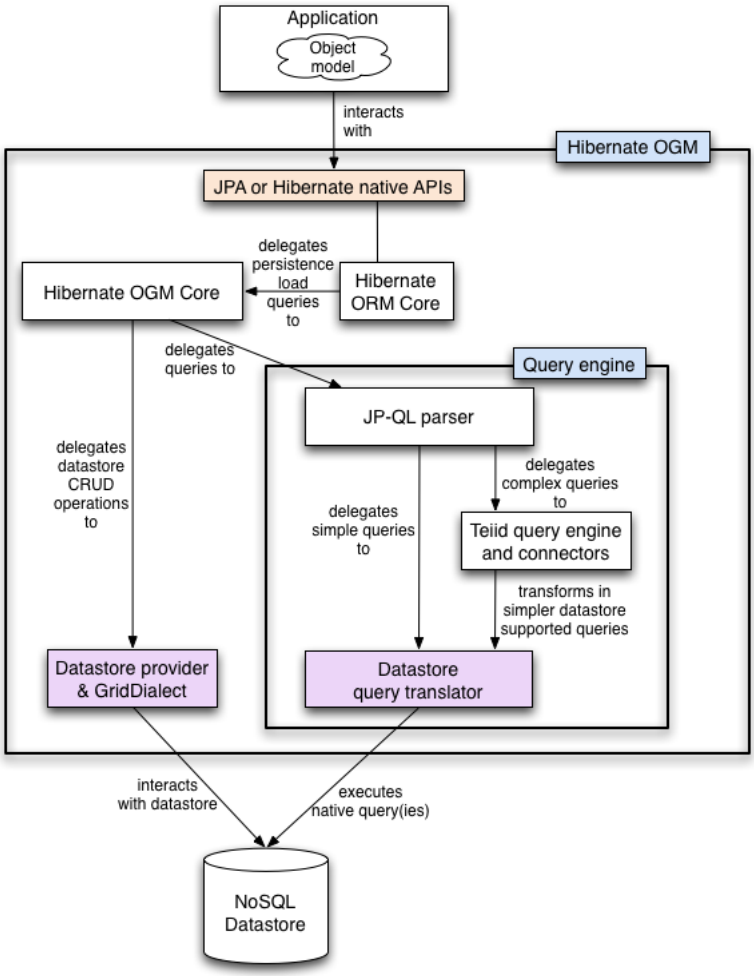


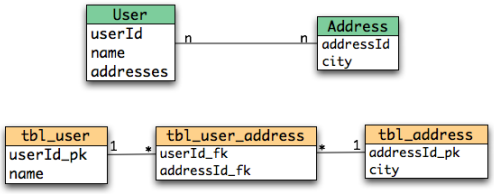

**Figura 10 – Arquitetura Básica de Acesso a Bancos de Dados NoSQL**

**Fonte: (HIBERNATE..., 2017b).**

De acordo com o banco de dados escolhido é necessário um *GridDialect* para realizar a conversão do mapeamento para o dialeto do banco. Por exemplo, para bancos orientados a documentos, como o MongoDB, o mapeamento é representado por um documento e cada valor do mapeamento corresponde a uma propriedade do documento (HIBERNATE..., 2017b).

Associações de entidades em bancos não-relacionais não são realizadas de forma trivial como em bancos relacionais. Porém, o Hibernate OGM consegue realizar estas associações. Ele armazena informações que são pertinentes para a recuperação da associação persistida, utilizando para isto chaves de pesquisa que são compostas pelo nome da tabela correspondente, pelo nome da coluna que corresponde a chave estrangeira da relação e o valor desta chave estran-

geira. A Figura 11a apresenta a forma como as relações são realizadas em bancos relacionais, já a Figura 11b apresenta a forma como o Hibernate OGM realiza as associações em bancos não relacionais.




**(a) Representação em Modelo Relacional da Relação *User* e *Adress***

| key | value |
|---|---|
| tbl_user,userId_pk,1 | {userId_pk=1,name="Emmanuel"} |
| tbl_user,userId_pk,2 | {userId_pk=2,name="Caroline"} |
| tbl_address,addressId_pk,3 | {addressId_pk=3,city="Paris"} |
| tbl_address,addressId_pk,5 | {addressId_pk=5,city="Atlanta"} |
| tbl_user_address,userId_fk,1 | { {userId_fk=1, addressId_fk=3}, {userId_fk=1, addressId_fk=5} } |
| tbl_user_address,userId_fk,2 | { {userId_fk=2, addressId_fk=3} } |
| tbl_user_address,addressId_fk,5 | { {userId_fk=1, addressId_fk=5} } |
| tbl_user_address,addressId_fk,3 | { {userId_fk=1, addressId_fk=3}, {userId_fk=2, addressId_fk=3} } |

**(b) Representação em Modelo Não-Relacional da Relação *User* e *Adress***

**Figura 11 – Associações em Bancos de Dados Relacionais e Não-Relacionais**

**Fonte: (HIBERNATE..., 2017b).**

Este método para realizar as associações é bastante eficaz, pois utiliza de um conceito puramente relacional e o utiliza em um paradigma NoSQL. Porém, apesar de ser uma estratégia muito boa, devido ao fato de ser possível realizar todas as operações de CRUD, isso causa redundância de dados, porque o mesmo dado é salvo várias vezes nas chaves de pesquisa.

## 2.5 JAVA PERSISTENCE API (JPA)

O JPA é um *framework* que é utilizado para controlar a camada de persistência. Este *framework* abstrai os conceitos da camada de persistência, ou seja, a partir do JPA é possível mapear todo o sistema de banco de dados sem conhecimento de como este é estruturado. Esse mapeamento é realizado através de anotações.

### 2.5.1 Anotações

As anotações são um recurso do Java que permitem a inserção de metadados nas classes já existentes. Isso significa que *frameworks* como o JPA podem interpretar essas informações e fazer o mapeamento do sistema de banco de dados da maneira que foi pretendida sem que a aplicação conheça como esse banco foi configurado, bem como sua estrutura. O JPA possui diversas anotações para definir os mais diversos parâmetros. O caractere @ sempre precede as anotações JPA.

Dentre as diversas anotações que o JPA possui, algumas são indispensáveis para a utilização deste *framework*. Dentre estas estão as anotações *@Entity* e *@Id*.

A *Entity* é a anotação que define uma classe Java como a representação de uma tabela no banco de dados (no caso relacional) ou um documento (no caso NoSQL). Assim uma classe Java deve ser precedida da anotação:

```
@Entity
```

Para o JPA esta classe já representa uma tabela no banco de dados, ou seja, ela está mapeada como uma entidade. No exemplo apresentado na Figura 12 é mostrado o uso da anotação de entidade e também da anotação *Id*. Esta anotação é indispensável pois toda entidade necessita de uma identificação única. Em uma analogia com bancos relacionais, o campo marcado com a anotação *Id* seria a chave primária da tabela, pois é um campo único que não admite valores repetidos.

```
@Entity
public class User implements Serializable {

    @Id
    private int userId;
    private String name;

    //sets and gets
}
```

**Figura 12 – Anotação @*Entity* e @*Id* na Classe *User***

**Fonte: Autoria Própria.**

Essas anotações são essenciais para o funcionamento, pois elas indicam para o JPA que a classe é uma entidade e qual é o valor único (chave primária) desta entidade. Ao utilizar uma entidade com apenas estas anotações o que o JPA faz é buscar uma tabela ou documento chamado *User* e que tenha atributos que tenham os nomes *userId* e *name*. O nome da classe Java ou de seus atributos podem ser diferentes do nome da tabela ou documento e seus atributos. Para isso devem-se usar as anotações @*Table* e @*Column*. Essas anotações indicam para o JPA quais os nomes que constam no banco de dados. Isso é feito através do parâmetro *name* dessas anotações. Assim a classe *User* pode ser escrita como na Figura 13 (COELHO, 2013).

O uso destas anotações não é obrigatório. Elas são recomendadas para o caso das classes e atributos terem nomes diferentes daquelas existentes no banco de dados.

```java
@Entity
@Table(name = "User")
public class Usuario implements Serializable {

    @Id
    @Column(name = "userId")
    private int idusuario;
    @Column(name = "name")
    private String nomeUsuario;

    //sets and get
}
```

**Figura 13 – Anotação *@Table* e *@Column* na Classe *User***

**Fonte: Autoria Própria.**

Foi apresentado na subseção 2.2.1 as formas de relações entre entidades. Cada uma dessas relações possuem uma representação em forma de anotações JPA. Através delas se definem as relações em cada entidade nos seus respectivos atributos.

A relação mais básica existente é a *one to one* e é bastante simples. Ela é indicada através da anotação:

```
@OneToOne
```

A Figura 14 apresenta como são estas entidades em uma relação 1x1. Na Figura 14a é apresentado o lado *user* e em 14b é mostrado o código para o lado *address* desta relação.

```java
@Entity
public class User implements Serializable {

    @Id
    private int userId;
    private String name;

    @OneToOne
    private Address adress;

    //sets and gets
}
```

**(a) Lado *User* da Relação 1x1.**

```java
@Entity
public class Address implements Serializable {

    @Id
    private int userId;
    private String name;

    @OneToOne(mappedBy = "address")
    private User user;

    //sets and gets
}
```

**(b) Lado *Address* da Relação 1x1.**

**Figura 14 – Anotação da Relação 1x1 Entre *User* e *Address*.**

**Fonte: Autoria Própria.**

Para que exista uma relação única entre *user* e *address* foi adicionado o parâmetro *mappedBy* na entidade *address*. Isso indica que a dominância do relacionamento está na entidade *user*.

Sem definir a dominância de relação, o JPA considera que existem duas relações entre estas entidades, ou seja, para o JPA, neste caso, existiria uma relação entre o usuário e o endereço e outra relação entre o endereço e o usuário, isto é, duas relações unidirecionais. O que o *mappedBy* faz é definir que estas duas relações são na realidade apenas uma e apenas a entidade *user* irá conter uma chave estrangeira com a identificação de endereço, e o JPA assume através do *mappedBy*, o retorno desta relação, transformando-a em bidirecional (COELHO, 2013). O valor contido em *mappedBy* indica qual o atributo dentro da entidade *user* é utilizado para o relacionamento entre as duas entidades.

Outra forma de relação é a muitos para um ou um para muitos. Para este relacionamento são utilizadas as seguintes anotações JPA:

`@ManyToOne e @OneToMany`

Neste caso deve-se qual será a entidade que terá muitos e qual a entidade que terá um. Assim foi considerado que um usuário pode ter não apenas um, mas muitos endereços registrados. A estrutura das entidades para este relacionamento é vista na Figura 15:

```java
@Entity
public class User implements Serializable

    @Id
    private int userId;
    private String name;

    @OneToMany(mappedBy = "user")
    private List<Address> adresses;

    //sets and gets
}
```

**(a) Lado *User* da Relação 1xN**

```java
@Entity
public class Address implements Serializable {

    @Id
    private int addressId;
    private String city;

    @ManyToOne
    @JoinColumn(name = "userId")
    private User user;

    //sets and gets
}
```

**(b) Lado *Address* da Relação Nx1**

**Figura 15 – Anotação da Relação 1xn Entre *User* e *Address***

**Fonte: Autoria Própria.**

A Figura 15a mostra que a entidade *user* tem uma lista de endereço no atributo marcado com a anotação *@OneToMany*. Isso deixa claro que um usuário pode possuir muitos endereços. A entidade *address* (Figura 15b), por sua vez, possui apenas uma instância de *user* indicando que um endereço possui apenas um usuário.

Neste exemplo da Figura 15 além das anotações de relacionamento, também foi utilizada a anotação *@JoinColumn*. Esta anotação indica para o JPA qual parâmetro deve ser usado para fazer a ligação entre as entidades.

Da mesma maneira que a relação um para um, a relação um para muitos deve conter um lado que comanda a relação, neste caso, o lado dominante é o *address*. Desta forma a entidade não dominante deve indicar qual é o parâmetro na entidade dominante que representa a relação através do parâmetro *mappedBy* da anotação.

É importante também saber qual anotação colocar em cada entidade. A anotação *@ManyToOne* deve ficar na entidade em que pode se ter muitos, portanto no exemplo esta anotação deve estar na entidade *address*, pois podem existir diversos endereços para um único usuário e *user* pode ter apenas um registro na entidade *address*, ou seja, cada endereço possui apenas um único usuário. Desta forma a entidade que recebe a anotação *@OneToMany* é a *address*.

A relação mais completa, que permite múltiplos relacionamentos em ambos os lados, é feita através da anotação:

```
@ManyToMany
```

Através deste relacionamento um usuário pode ter muitos endereços cadastrados e um endereço pode conter muitos usuários. A Figura 16 apresenta como deve ser implementado o código em Java utilizando JPA para este tipo de relacionamento.

```java
@Entity
public class User implements Serializable {

    @Id
    private int userId;
    private String name;

    @ManyToMany
    @JoinTable(name = "User_Address",
            joinColumns = = @JoinColumn(name = "userId"),
            inverseJoinColumns = @JoinColumn(name = "addressId"))
    private List<Address> adresses;

    //sets and gets
}
```

**(a) Lado *User* da Relação NxM**

```java
@Entity
public class Address implements Serializable {

    @Id
    private int addressId;
    private String city;

    @ManyToOne(mappedBy = "adresses");
    private List<User> user;

    //sets and gets
}
```

**(b) Lado *Address* da Relação NxM**

**Figura 16 – Anotação da Relação mxn Entre *User* e *Address***

**Fonte: Autoria Própria.**

Como em todos os tipos de relacionamentos já citados, para a relação muitos para muitos também deve ter uma entidade dominante, que comanda a relação. A entidade não dominante tem apenas que adicionar o parâmetro *mappedBy* à anotação *@ManyToMany*, como é mostrado na Figura 16b.

Para o lado dominante, é necessário que alguns parâmetros sejam configurados corretamente para um perfeito funcionamento do relacionamento. A Figura 16a apresenta a anotação que indica que esta estidade tem uma relação de muitos para muitos com *address* (*@ManyToMany*). Mas é preciso também a anotação *@JoinTable*. O parâmetro *name* dela indica a tabela ou documento de relação que é utilizada para armazenar os registros da relação e os parâmetros *joinColumn* indica o atributo da entidade dominante e *inverseJoinColumn* indica o atributo da entidade não dominante incluídos na tabela ou documento de relação. Esses atributos são marcados com a anotação *@JoinColumn*.

O JPA disponibiliza diversas anotações para configuração de suas entidades. As anotações apresentadas nesta seção são as mínimas necessárias para a configuração de um projeto.

O JPA quando devidamente configurado utiliza objetos Java para fazer referência a estados do banco de dados (COELHO, 2013). Assim as operações realizadas no banco de dados sempre são representadas por um objeto mapeado com a anotação *@Entity*. Por exemplo, uma inserção é realizada no banco de dados através de um objeto da entidade que se deseja inserir. Essa entidade tem seus parâmetros definidos e em seguida é enviado para a classe responsável pela persistência dos dados que irá persistir esse objeto como um registro no banco de dados.

## 2.6 JAVA PERSISTENCE QUERY LANGUAGE

A *Java Persistence Query Language* (JPQL) é uma linguagem baseada em *Queries* SQL utilizadas para realização de operações de CRUD sobre um banco de dados e mapeada com o auxílio do JPA. O JPQL permite escrever *queries* capazes de realizar operações em diversos bancos de dados sem a necessidade de modifica-las. Por exemplo, uma busca simples pode ter mudanças sutis nas *queries* de cada banco de dados relacionais disponíveis no mercado (COELHO, 2013), como mostra a Tabela 2.

**Tabela 2 – Query de Pesquisa em Diferentes Bases de Dados Relacionais**

| | |
|---|---|
| **MySQL** | SELECT * FROM User LIMIT 15 |
| **MS SQL** | SELECT TOP 15 FROM User |
| **Postgres** | SELECT * FROM User WHERE rownum <= 15 |

**Fonte: (COELHO, 2013)**

As diferenças na consulta de usuários apresentadas nesta tabela existem considerando apenas as bases relacionais. Ao incluir as bases não-relacionais as diferenças são ainda maiores, pois existem diversos tipos de armazenamento NoSQL. Neste caso não é preciso considerar a limitação na busca para as diferenças aparecerem, a busca simples por *Users* já é bastante diferente, como é apresentado na Tabela 3.

Se for considerado o uso de apenas um banco de dados a grande variedade de *query*

**Tabela 3 – Query de Pesquisa em Diferentes Bases de Dados Não Relacionais**

| | |
|---|---|
| **MongoDB** | db.User.find(); |
| **Neo4j** | MATCH (u.User) RETURN u.name |

**Fonte: (PANIZ, 2016)**

*languages* não seria um problema. Mas se um sistema for projetado pensando na sua portabilidade, o JPQL aparece como uma ótima ferramenta para unificar todas essas sintaxes. O JPQL é a linguagem utilizada pelo JPA para realização de *queries* e sua sintaxe é bastante semelhante a sintaxe do MySQL (COELHO, 2013). Por exemplo, em uma pesquisa na tabela *User* a *query* em JPQL seria realizada conforme apresentado na Figura 17.

```
Query pesquisa = entityManager.createQuery("SELECT u FROM User u");
pesquisa.getResultList();
```

**Figura 17 – *Query* JPQL de Pesquisa**

**Fonte: Autoria Própria.**

Como mencionado, esta sintaxe é muito próxima a sintaxe já conhecida para bases relacionais. Assim, em uma aplicação que utiliza o JPA com JPQL não é necessário saber qual será o banco de dados que se esta trabalhando. Esta *query* também é capaz de realizar operações em bases não-relacionais graças a camada de persistência como o Hibernate OGM, que foi detalhado na subseção 2.4.1.

Mesmo em pesquisas mais complexas o JPQL pode ser bastante simples (Figura 18) abstraindo o banco de dados que está abaixo.

```
Query pesquisa = entityManager.createQuery(
                    "SELECT u.userName, a.userAdress "
                    "FROM User u "
                    "LEFT JOIN u.userAdress a "
                    "WHERE a.userAdress LIKE CONCAT('%', :userAdress, '%')";
pesquisa.setParameter("userAdress", userAdress).getResultList();
```

**Figura 18 – *Query* JPQL de Pesquisa com Operador *Join***

**Fonte: Autoria Própria.**

Por se tratar de uma linguagem orientada a objetos, não é necessário mostrar como é a ligação entre as entidades, mas sim mostrar qual a relação entre os objetos (COELHO, 2013). Por esse motivo são utilizados os *alias* de cada entidade, que no exemplo acima é a letra $u$ referenciando a entidade *User* e a letra $a$ representado a entidade *Address*, que é definida neste caso pela relação existente entre as duas entidades. O JPQL consegue distinguir que existe uma relação entre as entidades, pois *userAddress* é uma chave estrangeira na entidade *User* definida por uma anotação JPA.

Esta linguagem de *queries* proporciona abstrair ainda mais a persistência de entidades, bem como suas relações. Para casos em que o sistema de banco de dados é relacional cada entidade do JPA tem relação com o nome da tabela mapeada na base, e nos casos em que o sistema de banco de dados é NoSQL, como em bancos orientados a documento (MongoDB), as entidades são relacionadas com o nome de cada documento. Com isso as *queries* JPQL interpretam as seleções e demais operações pelos nomes das entidades (ORACLE, 2017).

### 2.6.1 JMETER

JMeter é uma aplicação desenvolvida pela Apache para realização de testes funcionais e de desempenho. Primeiramente foi pensado para realizar testes em aplicações Web, porém tem sua utilização em demais tipos de aplicações (APACHE. . . , 2017c).

O Apache JMeter é capaz de realizar testes sobre determinados tipos de aplicações como por exemplo:

- Web - solicitações HTTP, HTTPS.
- WebServices - SOAP e REST.
- Acesso direto a base de dados por JDBC.
- TCP.
- Acesso a objetos java.

O principal motivo para escolha da utilização do JMeter foi o fato desta ferramenta realizar acesso direto a objetos Java. Esses objetos fazem o acesso ao banco de dados através da camada de persistência, portanto foi possível realizar os testes desta camada diretamente pelo JMeter. Além do acesso direto aos objetos Java da aplicação também é possível realizar a contagem de tempo apenas de um bloco de código, podendo separar sua análise de tempo de todo o resto da execução da aplicação, fator este que proporciona a extração de dados de tempo específicos que a camada de persistência necessita para a realização das operações sobre os bancos de dados utilizados.

O JMeter é um *framework multi-threading* que proporciona a possibilidade de utilização de processos concorrentes, a escolha deste *framework* se deu devido a necessidade neste projeto de realizar testes envolvendo carga em bancos de dados. A concorrência em banco de dados é um fator diferenciador no desempenho, pois em uma aplicação nem sempre os acessos ao banco de dados serão feitos de forma exclusiva por um usuário. Na maioria das vezes as aplicações são acessadas por diversos usuários simultaneamente, tendo esses que concorrer ao acesso dos recursos da aplicação.

O acesso concorrente ao banco de dados mostra uma real visão da performance, pois o banco tem que lidar com o gerenciamento do acesso aos dados com diversos processos, aumentando assim a complexidade da realização das operações.

A utilização do JMeter no projeto em questão se deu devido a solicitação de acesso de multi processos em concorrência (*multi-threading*) aos recursos dos bancos de dados MySQL e MongoDB, através da análise de requisições a objetos Java utilizados na aplicação para a persistência de dados.

Através da ferramenta de auxílio a testes foi possível a extração de dados importantes para a análise realizada do desempenho da camada de persistência em bancos de dados com paradigmas tão diferenciados como os em questão. O JMeter é capaz de exportar arquivos com os dados dos resultados de medições dos cenários de testes executados, sendo de grande importância para a construção de gráficos e análises sobre o desempenho da camada de persistência utilizada.

### 2.6.2 Acesso a Objetos Java

A aplicação desenvolvida no projeto para manipulação de dados envolvendo a camada de persistência é uma requisição Java sendo referenciada como um cliente e o cenário de testes criados faz o papel de servidor da requisição (APACHE. . . , 2017e).

Para realizar a comunicação entre o JMeter e os objetos Java o cliente da requisição deve implementar a interface *JavaSamplerClient* da biblioteca do JMeter para Java.

O recomendado é realizar a extensão da classe *AbstractJavaSamplerClient* ao invés da implementação da interface *JavaSamplerClient*, isto é utilizado para a proteção dos testes desenvolvidos de possíveis mudanças posteriores de interface.

Uma instância da *JavaSamplerClient* é criada pelo JMeter para cada usuário do teste a ser simulado, bem como algumas instâncias adicionais de uso interno da ferramenta (APACHE. . . , 2017d).

A configuração para a comunicação entre o JMeter e objetos Java pode ser separada nas seguintes partes:

- Configuração Cliente Java

  A estrutura a ser utilizada para escrita de código de uma classe Java que possibilite acesso através de requisições do JMeter é demonstrada na Figura 19.

  Dentro do escopo do método **runTest** é que deve ser escrito o código a ser executado nos testes ao receber uma requisição do JMeter.

- Configuração JMeter

```java
public class ClasseDeTeste extends AbstractJavaSamplerClient implements Serializable {

    @Override
    public SampleResult runTest(JavaSamplerContext jsc) {

        //Código a ser executado no teste
        SampleResult result = new SampleResult();

        result.sampleStart();
        //Tempo contabilizado será do código entre sampleStart e sampleEnd.
        result.sampleEnd();

        //Retorna status do teste para JMeter
        result.setSuccessful(true);
        return result;
    }
```

**Figura 19 – Código Java - Cliente**

**Fonte: Autoria Própria.**

O JMeter faz o papel de servidor da requisição a acessos Java. Para poder acessar os objetos é necessário ser gerado o arquivo executável da aplicação, este arquivo deve estar inserido na pasta **lib/ext** dentro do diretório dos arquivos do JMeter. A Figura 20 mostra a estrutura de arquivos da pasta do JMeter, bem como a localização correta em que deve estar o arquivo executável Java (*.jar*) da aplicação que será testada. Caso a aplicação Java utilize alguma biblioteca, seus arquivos devem estar também neste diretório do JMeter especificamente dentro da pasta **lib/ext/lib**.

A interface do JMeter pode ser acessada dentro da pasta ***bin*** que pode ser vista na estrutura de arquivos mostrada na Figura 20 através do arquivo executável ApacheJMeter.

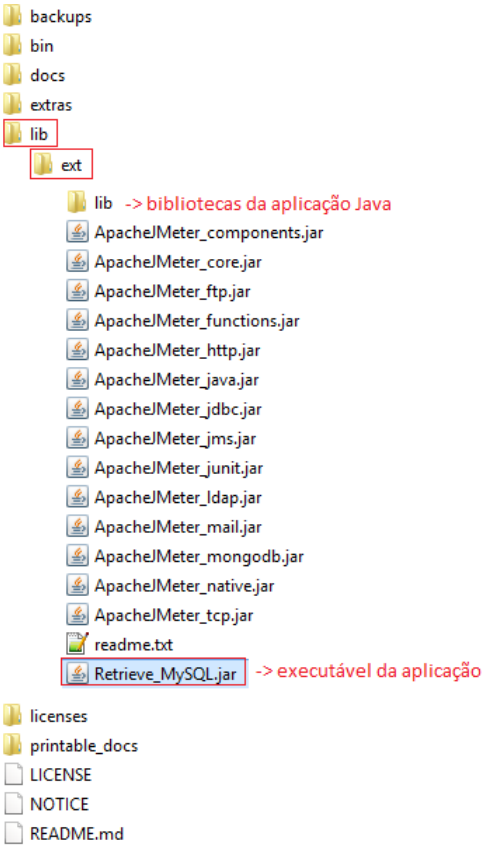
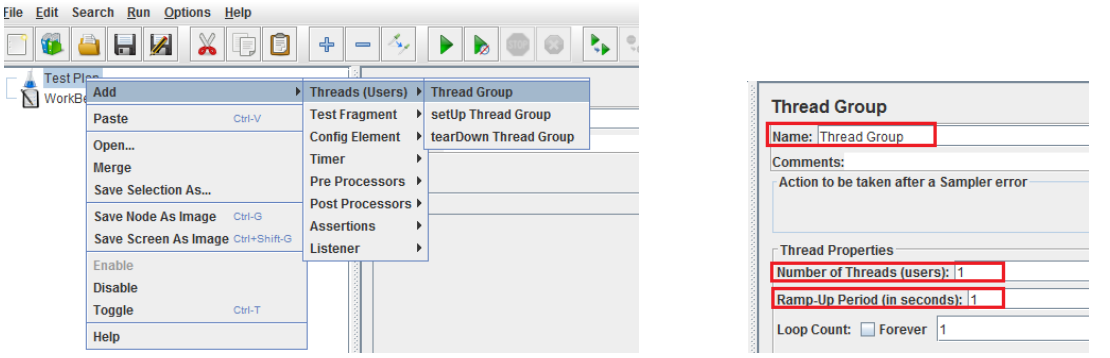



**Figura 20 – Estrutura de arquivos JMeter para Requisição Java**

**Fonte: Autoria Própria.**

Através da interface é necessário criar um plano de testes (*TestPlan*), em seguida deve-se adicionar grupos de usuários (*ThreadGroup*) a este plano de testes. O gripo de usuários é onde configura a quantidade usuários a serem simulado no teste. A Figura 21a mostra como adicionar um grupo de usuário e a Figura 21b mostra os campos que necessitam ser preenchidos. O campo *ramp-up* representa o tempo máximo para que todos os usuários iniciem, o campo *Number of Threads(users)* representa o número de usuários a serem simulados pelo teste e o campo *Name* é o nome de identificação do grupo de usuários.

Após a inserção do grupo de usuários é possível realizar a configuração da requisição Java a ser feita. Para isto é necessário a inserção de um *Sampler Java Request* dentro do grupo de usuários.




**(a) Adição de arquivo *ThreadGroup*** **(b) Configuração *ThreadGroup***

**Figura 21 – JMeter - Grupo de Usuários (*ThreadGroup*)**

**Fonte: Autoria Própria.**

Os passos para a criação do *Java Request* podem ser vistos na Figura 22. Nesta figura é possível ver que o *Java request* é adicionado ao grupo de usuários diretamente.

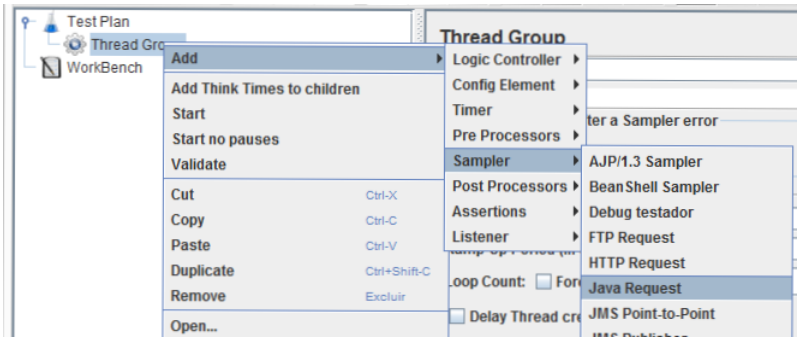



**Figura 22 – JMeter Adição de Requisição Java**

**Fonte: Autoria Própria.**

Os campos para configuração do *sample Java request* podem ser vistos através da Figura 23.

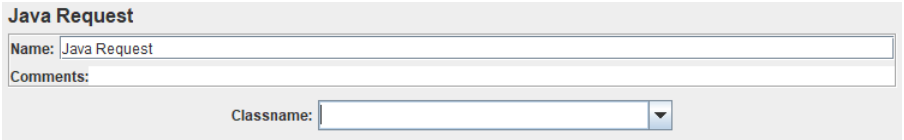



**Figura 23 – Configuração *Java Request***

**Fonte: Autoria Própria.**

Ao criar o *Java Request* é preciso indicar qual a interface Java a ser acessada pelo JMeter através do campo *ClassName*. As opções exibidas no campo *ClassName* são as classes Java que estendem a classe *AbstractJavaSamplerClient* e implementam o método *runTest* como citado na configuração do cliente Java. É atraves da indicação deste campo que será identificado a classe e o código a ser executado na análise do teste.

# 3 METODOLOGIA E DESENVOLVIMENTO

Neste capítulo serão discutidos a metodologia utilizada e o desenvolvimento do projeto. Também serão descritas as ferramentas utilizadas para o desenvolvimento.

## 3.1 ESCOPO

Este projeto tem como objetivo analisar o desempenho da camada de persistência de bancos de dados relacionais e não relacionais além de implementar um software para realizar estes testes sobre banco de dados relacional (MySQL) e NoSQL (MongoDB).

Para este projeto foram utilizados o Hibernate juntamente com o JPA para ao desenvolvimento desta aplicação. Além de persistir os dados o Hibernate também faz toda a manipulação de dados no banco, como consultar, recuperar e atualizar dados. O JPA abstrai ainda mais a comunicação da aplicação com o banco de dados, pois ele faz a comunicação com a camada de persistência de maneira transparente para o desenvolvedor, isto quer dizer que não é necessário saber o que há abaixo dele, podendo desta forma, por exemplo, trocar o *framework* de persistência sem que haja uma mudança de código do JPA ou das camadas superiores a ele.

Os bancos de dados sempre devem ter as mesmas informações devido aos software de testes da camada de persistência. Este software foi desenvolvido para realizar testes funcionais que mostraram se a camada de persistência é capaz de executar as mesmas operações em Mongo DB que executa no MySQL, e testes não funcionais que mediram o desempenho da camada de persistência para bancos de dados relacionais e não-relacionais sem que houvesse nenhuma mudança no código da aplicação a não ser a mudança na própria camada.

## 3.2 ETAPAS

Nesta seção serão descritas as atividades a serem realizadas para a execução deste projeto. As atividades foram divididas em etapas conforme descritas a seguir.

- **Etapa 1:** Estudo sobre base de dados relacionais e não relacionais

  Um estudo sobre softwares gerenciadores de banco de dados será realizado com o intuito de obter conhecimento sobre o funcionamento de cada um desses gerenciadores, ou seja, como cada um organiza os dados e como as operações de inserção, atualização, remoção e busca se diferenciam entre o MySQL e o MongoDB, assim como obter conhecimento sobre cada paradigma de armazenamento.

- **Etapa 2:** Projetar e mapear o banco de dados

  Será projetado um banco de dados MySQL, sendo todo o mapeamento feito para o paradigma relacional, utilizando-se de conceitos de entidade-relacionamento. As tabelas do MySQL serão geradas através de código Java utilizando o *framework* JPA.

- **Etapa 3:** Geração do banco de dados MongoDB

  Apesar da migração de paradigmas de armazenamento de dados em uma aplicação, a estrutura do novo banco de dados deve ser a mesma do original. Após o planejamento do banco de dados MySQL e efetuado sua geração como descrito na **Etapa 2**, os mesmos conceitos serão mantidos, e o código utilizado para geração das tabelas do MySQL será reutilizado para gerar as coleções do MongoDB.

- **Etapa 4:** Desenvolver uma aplicação de exemplo

  Será criada uma aplicação que realiza acesso ao banco de dados. Esta aplicação será feita de maneira simples, contendo apenas o necessário para a conexão da aplicação com o banco de dados e para realizar operações de criação, busca, atualização e remoção. A aplicação não conterá interface gráfica, pois o objetivo é verificar a eficiência apenas da camada de persistência evitando assim o uso desnecessário de recursos da máquina em que serão realizados os testes.

- **Etapa 5:** Configurar a camada de persistência

  Realizar a configuração da camada de persistência através de arquivo de configuração do Hibernate ORM para utilização de banco de dados relacional e Hibernate OGM para NoSQL. Fase necessária para proporcionar a comunicação da aplicação com o banco de dados, pois a camada de persistência é o enfoque principal deste projeto.

- **Etapa 6:** Popular banco de dados para os testes

  Para a realização dos testes será preciso que ambos os bancos de dados possuam dados armazenados previamente. Os bancos de dados gerados, como descritos em etapas anteriores, serão preenchidos com dados aleatórios.

  Uma aplicação será criada para gerar arquivos Java que utilizam-se das operações sobre os bancos de dados criadas no desenvolvimento do aplicação. Estes arquivos conterão os dados a serem inseridos nos bancos. Os dados serão gerados aleatoriamente utilizando o texto *loren-ipusum*. O *lorem-ipsum* nada mais é que um texto formado formado com palavras em latim. A partir desse texto serão selecionadas aleatoriamente palavas e assim serão criados os registros de cada tabela e documento que estarão contidos nos banco de dados.

- **Etapa 7:** Metodologia de testes.

  Será criada uma metodologia de testes para a camada de persistência visando apresentar de forma clara e objetiva a viabilidade ou não da migração de banco de dados em um sistema

já existente, baseado em utilização de camada de persistência acessando um banco de dados relacional.

A metodologia de testes irá se basear na utilização da ferramenta JMeter acessando a aplicação pré existente em Java por meio de requisições, sendo o JMeter o servidor e os objetos Java os clientes. O JMeter será configurado para realizar uma requisição Java e configurar a classe Java que utiliza a camada de persistência para realizar as operações de CRUD sobre os bancos MySQL e MongoDB.

Estipular quais os números de operações a serem realizadas, bem como o número de usuários a serem utilizados visando a análise de desempenho sobre efeito de concorrência.

- **Etapa 8:** Construção de gráficos e análise de resultados.

  Os testes criados na **Etapa 7** irão gerar dados que necessitam de análise. Esta análise visa obter informações relacionadas ao funcionamento adequado da camada de persistência em ambos os bancos de dados (testes funcionais) e também obter dados referentes ao desempenho das operações realizadas sobre os bancos através dessa camada (não funcionais).

### 3.2.1 Diagrama do Projeto

Um diagrama descritivo do projeto é apresentado na Figura 24, que apresenta as etapas de uma forma sucinta, sendo mostrado de forma a manter o foco nas etapas principais descritas nessa seção.

O diagrama tem por objetivo mostrar os pontos principais necessários para a implementação deste projeto, bem como questionamentos que geraram a demanda de algumas etapas utilizadas para o desenvolvimento da aplicação, a qual contou com uma rotina de testes da camada de persistência rodando sobre banco de dados com paradigmas de armazenamento distintos, e por fim foi realizada uma análise dos dados extraídos dos testes executados.

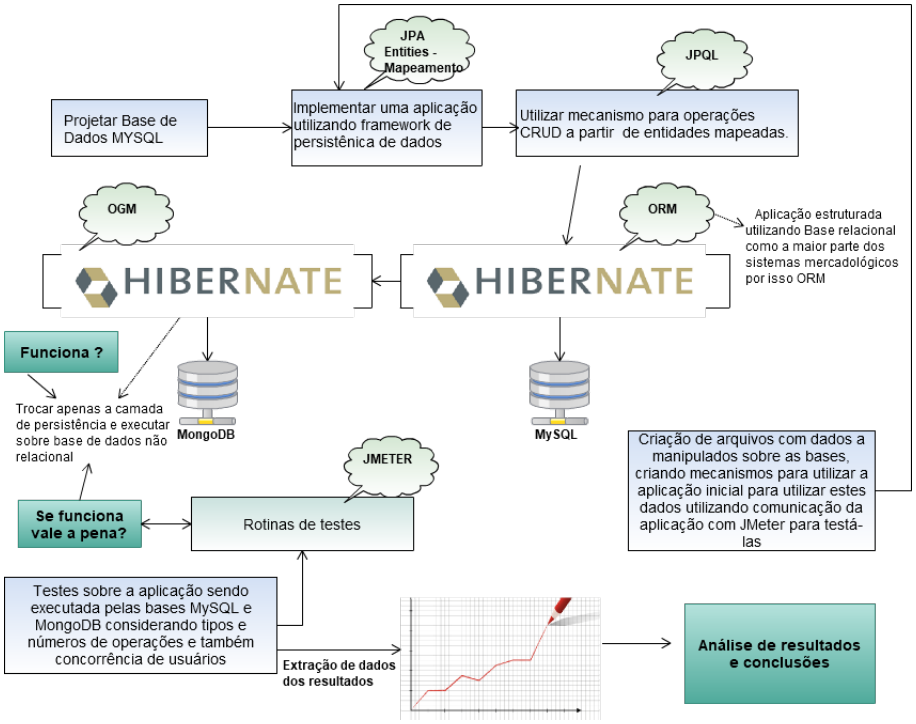



**Figura 24 – Diagrama do Projeto**

**Fonte: Autoria Própria.**

## 3.3 DESENVOLVIMENTO DO PROJETO

Este projeto foi subdividido em três partes, o banco de dados, uma aplicação para demonstrar a utilização de camadas de persistência e uma aplicação para gerar dados de preenchimento do banco.

Os dados a serem armazenados apresentam informações sobre filmes e livros, assim como informações pertinentes a eles, como por exemplo, quem são os autores, de que país vieram, qual o idioma, entre outras. Isso foi feito para mostrar como informações básicas se relacionam em um banco de dados. O desenvolvimento do banco de dados, bem como suas relações são apresentados na seção 3.3.1.

Todo o desenvolvimento da aplicação para testes foi feito em linguagem Java utilizando-se o *framework* JPA e Hibernate como camada de persistência. Foi através das anotações do JPA que foram criadas as entidades do banco de dados e seus relacionamentos. Cada entidade criada possui um gerenciador próprio, o *database manager* (DBM). O DBM é responsável pela configuração da conexão com o banco de dados, seja o MySQL ou o MongoDB. O DBM também é responsável por definir as *queries* de interação com ambos os bancos.

Por se tratar de uma aplicação que visa apenas mostrar o desempenho da camada de persistência em banco de dados de paradigmas distintos não foi desenvolvida interface gráfica. A implementação de um interface gráfica poderia interferir diretamente no resultado dos testes,

pois ela usa bastante recursos da máquina utilizada para os testes. Ao utilizar o processamento da máquina para executar a interface gráfica os dados obtidos nos testes poderiam não representar a realidade, pois o processador deve compartilhar os recursos entre a interface e o testes, o que pode ocasionar leituras falsas dos tempos de execução de cada operação.

### 3.3.1 Projeto do Banco de Dados

O desenvolvimento inicial teve como enfoque o paradigma relacional. Foram discutidos quais os dados a serem armazenados para poder estruturar as relações entre estes dados e então criar os relacionamento das entidades no banco.

O banco de dados foi projetado de modo que fosse possível a sua utilização em uma aplicação comercial. Desta forma, como citado anteriormente, foi estruturado um banco para a gerência de uma biblioteca em que foram criadas diversas entidades que se relacionam entre si. Essas interligações entre informações foram criadas para simular um banco de dados real que possui diversas relações entre os dados armazenados. Neste banco de dados duas tabelas se destacam pela quantidade de relacionamentos. A primeira tabela é a de filmes (*movies*) e seus relacionamentos são apresentados na Figura 25 e descritos na Tabela 4.

**Tabela 4 – Relacionamentos da Tabela *Movies***

| **Entidade 1** | **Tipo de Relação** | **Entidade 2** |
|---|---|---|
| Movies | Muitos para Muitos | Actors |
| Movies | Muitos para Muitos | Networks |
| Movies | Muitos para Um | Languages |
| Movies | Muitos para Muitos | Genres |
| Movies | Muitos para Muito | Writers |
| Movies | Muitos para Um | Countrys |
| Movies | Muitos para Muitos | Bookss |

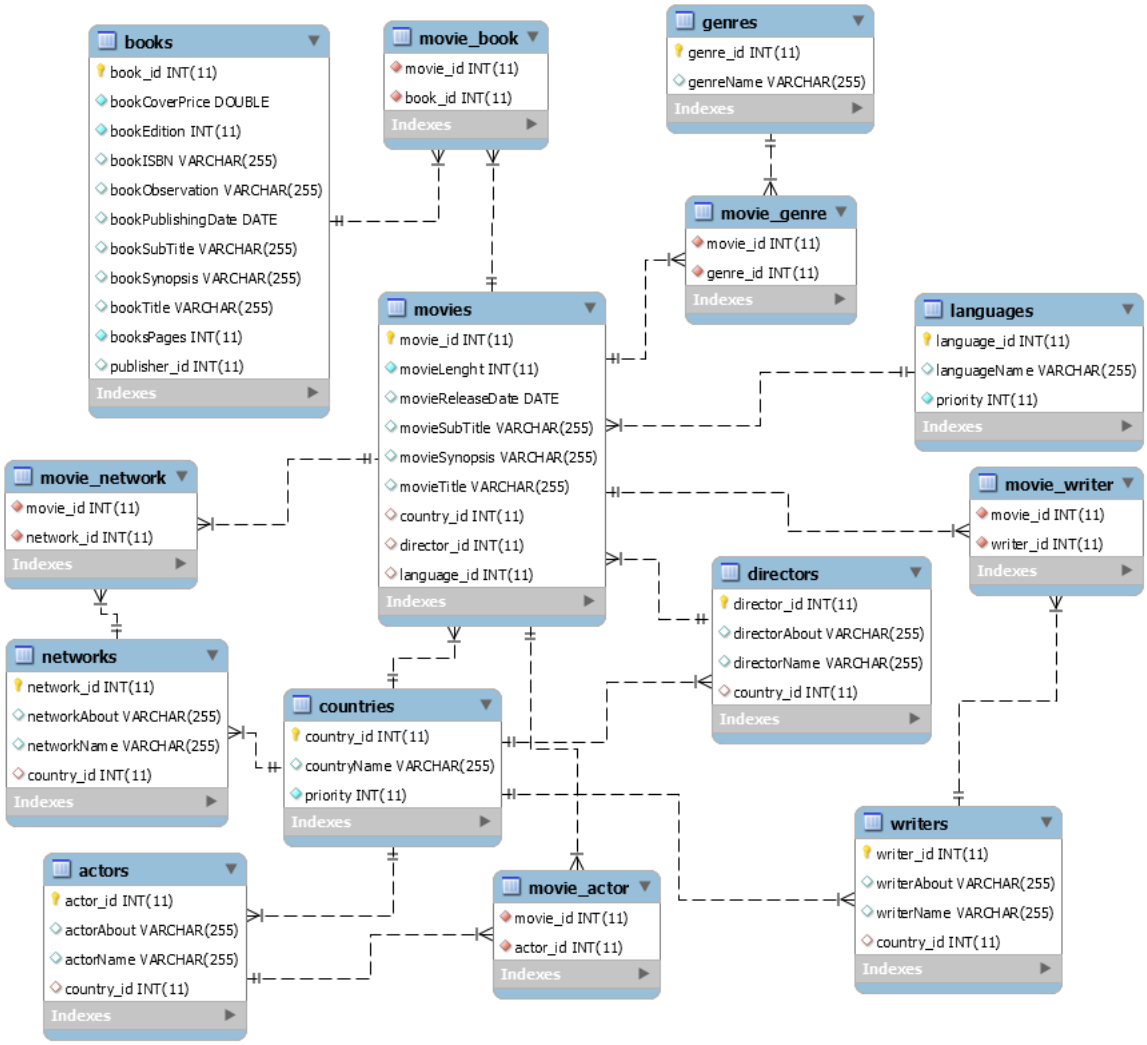



**Figura 25 – Diagrama De Relacionamento de Tabela *movies***

**Fonte: Autoria Própria.**

A segunda tabela que se destaca por seus relacionamentos é a tabela livros (*books*). Seus relacionamentos são apresentados na Figura 26 e são descritos na Tabela 5.

**Tabela 5 – Relacionamentos da Tabela *Books***

| Entidade 1 | Tipo de Relação | Entidade 2 |
|---|---|---|
| Books | Muitos para Muitos | Authors |
| Books | Muitos para Muitos | Languages |
| Books | Muitos para Muitos | Genre |
| Books | Muitos para Um | Publishers |
| Books | Muitos para Muitos | BookSeries |

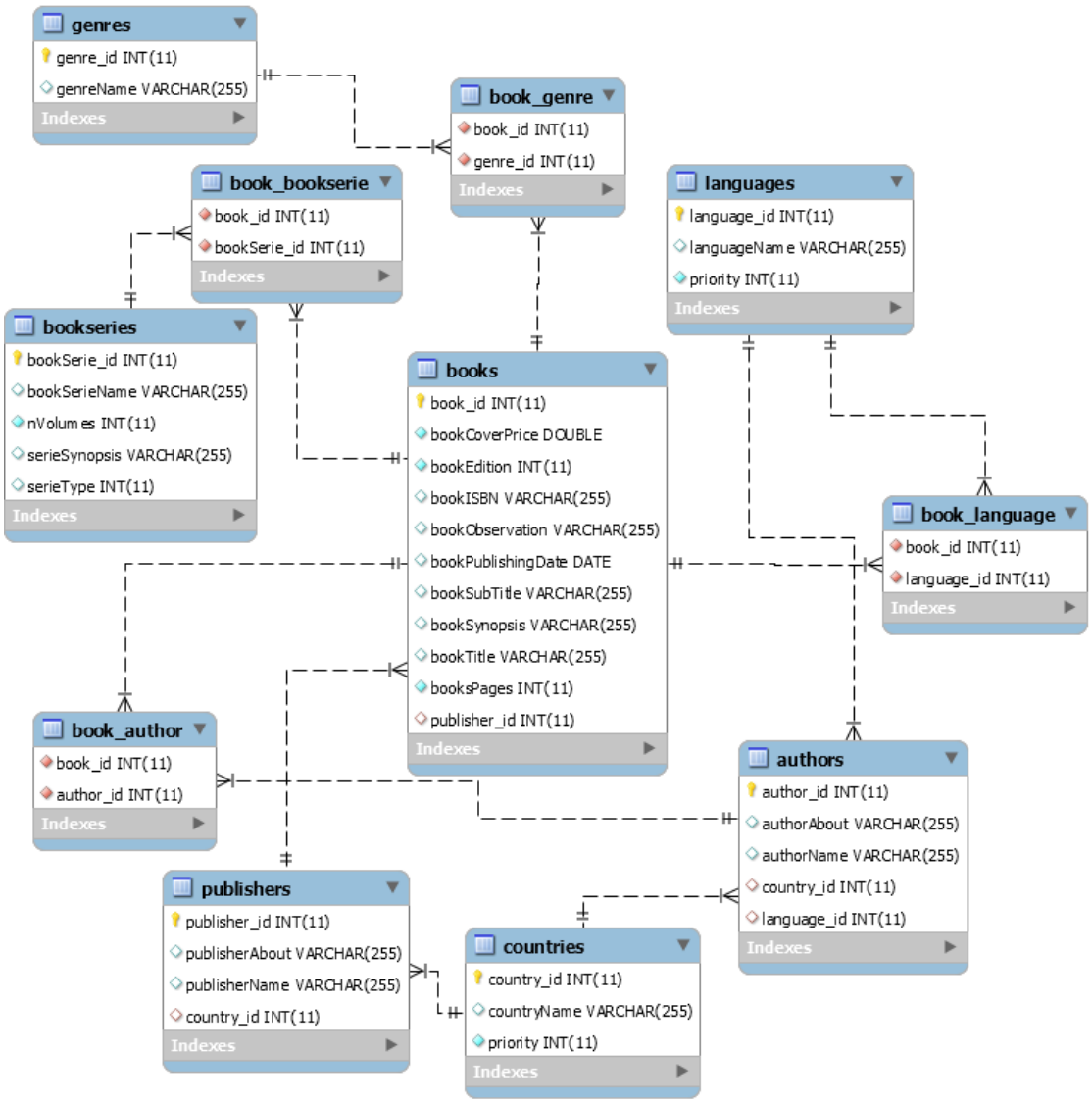



**Figura 26 – Diagrama De Relacionamento de Tabela *books***

**Fonte: Autoria Própria.**

As Figura 25 e 26 apresentam todos os relacionamentos contidos no banco de dados. Os demais relacionamentos entre as tabelas do banco de dados relacional são descritos na Tabela 6. O diagrama entidade-relacionamento completo é encontrado no Apêndice A.

**Tabela 6 – Relacionamentos Entre Tabelas**

| Entidade 1 | Tipo de Relação | Entidade 2 |
|---|---|---|
| Actors | Muitos para Um | Countries |
| Authors | Muitos para Um | Countries |
| Authors | Muitos para Um | Languages |
| Countries | Um para Muitos | Publishers |
| Countries | Um para Muitos | Directors |
| Countries | Um para Muitos | Writers |
| Countries | Um para Muitos | Networks |

### 3.3.2 Projeto da Aplicação de Teste

Nesta seção será discutido o desenvolvimento da aplicação que foi utilizada para a realização dos testes. Nela se encontram todas as informações necessárias para a conexão com os bancos de dados (configuração da camada de persistência), as entidades (mapeamento dos bancos de dados), os gerenciadores (classes que contém as *queries* de operações), etc.

Esta aplicação foi desenvolvida e configurada para acessar o banco de dados MySQL. Desta forma as tabelas apresentadas nas figuras 25 e 26 foram representadas, com seus atributos e relacionamentos, em forma de entidade utilizando as anotações do JPA e os relacionamentos foram feitos segundo os relacionamentos apresentados nas tabelas 4, 5 e 6.

Apesar de as entidades terem sido modeladas visando o banco de dados relacional, este tipo de abordagem na geração do banco possibilitou que o mesmo código fosse utilizado para a criação de ambos os bancos de dados, pois a forma de mapeamento das entidades foi a mesma, independente do banco de dados utilizado.

#### 3.3.2.1 Configuração da conexão

A configuração da conexão é a parte mais importante da aplicação, pois é nesta configuração que são definidas as propriedades e características da conexão.

A primeira etapa é a configuração do arquivo *persistence.xml*. A Tabela 7 apresenta os parâmetros utilizados para definir a unidade de persistência.

**Tabela 7 – Configuração *tag persistence-unit***

| ***Tag persistence-unit*** | | |
|---|---|---|
| **Parâmetro** | ***name*** | ***transaction-type*** |
| **MySQL** | mysql_PU | mongo_PU |
| **MongoDB** | RESOURCE_LOCAL | RESOURSE_LOCAL |

O provider deve ser configurado com o *framework* que será utilizado para a camada de persistência. Neste trabalho foi utilizado o Hibernate ORM para o MySQL e para o MongoDB foi utilizado o Hibernate OGM. Os parâmetros usados na *tag provider* é apresentado na tabela

**Tabela 8 – Configuração *tag provider***

| ***Tag privider*** |
|---|
| **MySQL** |
| org.hibernate.ejb.HibernatePersistence |
| **MongoDB** |
| org.hibernate.ogm.jpa.HibernateOgmPersistence |

Cada banco de dados necessita de configurações diferentes no arquivo *persistence.xml*. Os parâmetros de configuração da camada de persistência do MySQL são apresentados na Tabela 9.

**Tabela 9 – Propriedades da Conexão com o MySQL**

| | |
|---|---|
| **host** | localhost:3306 |
| **database** | newlibrary_v4 |
| **user** | root |
| **password** | rachel |
| **entitymanager_factory_name** | mysql_PU |
| **jdbc.driver** | mysql.jdbc.driver |
| **hbm2ddl.auto** | update |

Os parâmetros da configuração da camada de persistência do MongoDB são mostrados na Tabela 10.

**Tabela 10 – Propriedades da Conexão com o MongoDB**

| | |
|---|---|
| **provider** | mongodb |
| **database** | newlibrary_mongo |
| **create_database** | true |
| **host** | localhost:27017 |
| **user** | - |
| **password** | - |

A configuração detalhada do *persistence.xml* é apresentado no Apêndice B.

Para completar a conexão, foi criada a classe em Java que faz uso das unidades de persistência para acessar o banco de dados desejado. Para inciar uma conexão deve ser criada uma instância da *entity manager factory* e através dela é criado uma instância da *entity manager*. Com a *entity manager* configurada é possível realizar todas as operações desejadas no banco de dados. A Figura 27 apresenta a instanciação da conexão para o MySQL, porém para instanciar uma conexão para o MongoDB, basta colocar o nome da unidade de persistência do MongoDB no parâmetro do método *createEntityManagerFactory*.

```
public boolean setCon() {
    if (factory == null || em == null) {
        factory = Persistence
                .createEntityManagerFactory("mysql_PU");
        em = factory.createEntityManager();
        return true;
    }

    return false;
}
```

**Figura 27 – Método Java para Conexão com o Banco de Dados**

**Fonte: Autoria Própria.**

#### 3.3.2.2 Código gerenciador de banco de dados

Os gerenciadores de banco de dados são muito importantes para a aplicação, pois são eles que fazem a ligação entre as entidades e a conexão com o banco de dados. Os gerenciadores contém as operações de CRUD com o banco, ou seja, são classes Java que fazem a inserção, atualização e busca no banco de dados.

Para inserção de dados foi utilizado o código mostrado na Figura 28. Neste código o parâmetro *con* é a conexão com o banco de dados, ou seja, a *entity manager* que foi instanciada no início da conexão, como mostrado na Figura 27. O parâmetro *movie* indica o objeto que será inserido no banco de dados, ou seja, com a conexão ativa o objeto é persistido no banco de dados.

```
public boolean insertMovie(Movies movie, Connection con) {
    try {
        con.getCon().merge(movie);
        return true;
    } catch (Exception e) {
        con.getCon().getTransaction().rollback();
        return false;
    }
}
```

**Figura 28 – Método Java para Inserção e Atualização**

**Fonte: Autoria Própria.**

Este mesmo código pode ser utilizado para atualização de um registro. O método *merge* da conexão é capaz de adicionar um novo registro assim como atualizar um registro já existente no banco de dados. A diferenciação entre inserção e atualização é feita através do objeto recebido para persistir. Se este contém o parâmetro da chave primária definido, o *merge* faz uma busca por este registro e se encontrar atualiza-o com os dados do objeto. Contudo se nada for encontrado com esta chave, um novo registro é criado com esta chave primária. Existem também a possibilidade de o objeto vir sem o parâmetro de chave primária configurado, dessa forma um novo registro será criado, porém isso só ocorre se o banco de dados estiver configurado para gerar automaticamente a chave primária dos novos registros.

O método desenvolvido para pesquisa necessita além da conexão com o banco de dados, a *query* para a pesquisa que se deseja. Para pesquisar registros em um banco de dados existem inúmeras maneiras. Pode-se procurar um registro por seu identificador único, ou por qualquer campo contido na entidade. É possível também buscar um registro pelo valor de um campo de uma outra entidade relacionada a ela, utilizando-se o operador *join*. Devido a esta diversidade de pesquisa foram definidas duas *queries* para testar os bancos. A primeira apenas uma busca simples na entidade *movies*. A busca simples consiste na busca de filmes por seu título. A segunda *query* definida foi utilizando um *join* que é uma operação nativa de bancos de dados relacionais.

Ambas as *queries* de pesquisa foram escritas em linguagem JPQL nativo para o MySQL e são apresentadas na Figura 29.

```
public List retrieveMovieByTitle(String movieTitle, Connection con) {
    return con.getCon().createQuery("SELECT m FROM Movies m "
            + "WHERE m.movieTitle "
            + "LIKE CONCAT('%',:movieTitle,'%')")
            .setParameter("movieTitle", movieTitle)
            .getResultList();
}
```

**(a) *Query* de Pesquisa Simples**

```
public List retrieveOneJoin(String movieTitle,
        String director,String writer, String actor,
        String network, String book, Connection con) {

    return con.getCon().createQuery(
            "SELECT m.movieTitle "
            + "FROM Movies m "
            + "JOIN m.movieDirector d "
            + "WHERE m.movieTitle LIKE CONCAT('%',:movieTitle,'%')")
            .setParameter("movieTitle", movieTitle)
            .getResultList();
}
```

**(b) *Query* de Pesquisa com *join***

**Figura 29 – *Queries* de Pesquisa**

**Fonte: Autoria Própria.**

As figuras 28 e 29 referem-se ao gerenciador da entidade *movies*, porém todas as outras entidades possuem um gerenciador semelhante, diferenciando apenas no objeto que é passado para a realização da operação.

### 3.3.3 Projeto da Aplicação de Criação dos Arquivos de Teste

As entidades que se relacionam com *movies* devem estar previamente populadas para quando os testes de inserção, por exemplo, forem executados existam dados para serem adicionados a esta entidade. Com esse intuito foi criada uma aplicação paralela a aplicação de testes para gerar todos os dados de preenchimento das entidades e criação dos arquivos de teste de *movies*. Como estas entidades são fixas para qualquer tipo de teste de operação realizados uma vez criados não foi mais necessário nenhuma modificação posterior.

A geração dos dados de preenchimento do banco de dados foi realizada através da criação de arquivos Java. Esses arquivos foram adicionados em um pacote na aplicação de teste e foram executados utilizando-se dos gerenciadores de cada entidade. Por exemplo, o arquivo que insere elementos na entidade *actors* cria diversos objetos desta entidade e ao fim utiliza-se do método de inserção no gerenciador de *actors*. A Figura 30 mostra um objeto criado para esta entidade.

Os dados inseridos nos bancos foram criados através de um gerador randômico de palavras *Lorem Ipsum*. Este gerador, como o nome diz, gera palavras aleatórias em latim. Ele foi

```
c = new Countries();
c.setCountry_id(109);
Actors actors = new Actors(1, "vivamus scelerisque aliquam",
        c, "feugiat hendrerit dictum auctor vesti");
```

**Figura 30 – Objeto da Entidade *Actors* Usado para Preenchimento do Banco de Dados**

**Fonte: Autoria Própria.**

utilizado pela facilidade de criação de palavras e frases aleatórias.

Na Figura 30 nota-se que a entidade *actors* tem um relacionamento com a entidade *countries*, pois o objeto de *actor* contem um objeto *countries* em sua construção. O relacionamento entre as entidades é definida aleatoriamente, ou seja, qualquer *actor* pode ser relacionado com qualquer *countrie*.

Para inserção no banco de dados seguiu a ordem de entidades que não possuem relacionamentos com nenhuma outra entidade até a entidade com o maior número de relacionamentos, evitando assim inconsistência nos dados inseridos.

Os arquivos de testes foram criados da mesma forma, utilizando-se de paravras aleatórias *Loren Ipsum*, porém não foi criado um arquivo Java, mas um arquivo simples contendo apenas as informações de cada campo a ser inserido. A entidade *movies* possui diversos campos que não possuem relacionamentos com outras entidades, foram informações para estes campos que foram criadas e adicionados a um arquivo de texto. Os campos que possuem relacionamentos são aleatoriamente selecionados a partir dos registros contidos nas entidades relacionadas.

## 3.4 DESENVOLVIMENTO DO TESTE

Os testes foram realizados para as operações de CRUD como inserção, atualização e busca. Para determinar a capacidade da camada de persistência de gerenciar concorrência foram desenvolvidos testes com múltiplos usuários e cada usuário realiza diversas vezes cada operação. A Tabela 11 apresenta número de usuários em cada teste realizado para inserção, atualização e busca em ambos os bancos de dados. Estes números foram escolhidos para mostrar o comportamento da camada de persistência e do banco de dados com uma grande variação na quantidade de usuários gerando concorrência no acesso e realização de operações no banco.

**Tabela 11 – Número de Usuários Para os Testes**

| Numero de Usuários | | |
|---|---|---|
| **Inserção** | **Atualização** | **Busca** |
| 1 | 1 | 1 |
| 100 | 100 | 50 |
| 1000 | 1000 | 100 |
| 2500 | 2500 | - |
| 5000 | 5000 | - |

A quantidade de operações que cada usuário realizou nos testes é apresentada na Tabela 12. Esses valores foram escolhidos para mostrar o comportamento dos bancos de dados de acordo com o aumento de operações, e consequentemente do tempo de execução das *queries* de CRUD.

**Tabela 12 – Número de Operações por Usuário.**

| | | | | | |
|---|---|---|---|---|---|
| Inserção | 10 | 25 | 50 | 75 | 100 |
| Atualização | | | | | |
| Busca | | | | | |

Nos testes realizados com mais de um usuário foi estipulado que cada um desses usuários iria iniciar sua respectiva operação no banco a cada intervalo fixo de tempo. Para os testes de inserção de informações no banco e de atualização esse intervalo entre o início de cada usuário foi de 700 milissegundos. Para os testes de busca esse intervalo foi estendido para 1000 milissegundos. Esse atraso no início de cada usuário é importante pois se todos os usuários iniciarem a execução ao mesmo tempo. Esse tipo de comportamento não é esperado nos testes, pois o intuito é mostrar o desempenho dos bancos de dados com o surgimento gradual de usuários realizando operações sobre os bancos.

Os bancos de dados foram utilizados com a configuração padrão de cada um. Diferenças nos resultados poderiam ser obtidas através da modificação de parâmetros específicos de cada banco de dados. Por exemplo, a configuração para a geração automática de índice no MySQL pode afetar diretamente o tempo de busca no banco.

### 3.4.1 Configuração do Ambiente de Teste

Os testes foram criados no JMeter utilizando grupos de usuários e o *sample Java request*.

Cada *Java request* foi configurado com a interface Java referente ao teste realizado, ou seja, para os testes que que insere 25 registros no banco de dados o *Classname* do *Java request* deve conter o método que realiza esta inserção especifica. A Tabela 13 apresentam as interfaces Java utilizadas para a realização dos testes. Nesta tabela é apresentado o padrão de nomenclatura

destas interfaces, em que **X** assume os valores 10, 25, 50, 75 e 100 e representam o número de operações que serão realizadas no banco de dados por cada usuário.

**Tabela 13 – Interfaces Java Utilizadas Pelo JMeter para Realização dos Testes**

| Classname | |
|---|---|
| Inserção | TestInsertMovieX* |
| Atualização | TestUpdateMovieX* |
| Busca simples | TestRetrieveSimpleX* |
| Busca com join | TestRetrieveJoinX* |

***X = {10, 25, 50, 75, 100}**

O grupo de usuários necessita apenas de 2 parâmetros. O número de usuários que irão realizar os testes, que foi apresentado na Tabela 11, e o tempo de início de cada usuário. Como citado anteriormente para os testes de inserção e atualização, que são operações mais rápidas, foi utilizado um usuário iniciando sua operação a cada 0,7 segundo. Para os testes de busca foi utilizado um usuário a cada segundo por ser uma operação mais lenta. Esse valor foi alterado para os testes de busca pois com o valor de 0,7 segundo nenhum banco foi capaz de realizar totalmente os testes, portanto foi elevado o tempo de início de cada usuário para tornar viável os testes de busca.

O apêndice C apresenta mais detalhes da configuração do ambiente de testes JMeter.

Todos os testes foram executados em um computador *desktop* exclusivo para esta finalidade. As configurações de hardware e software são apresentadas a seguir:

- **Processador:** Intel Core I5 2,8 GHz.
- **Memoria RAM:** 8 GB.
- **Sistema Operacional:** Windows 8.1 Pro e arquitetura de 64 bits.
- **Versão Java:** 8.0.1310.11.
- **Versão MySQL Server:** 5.7.18.
- **Versão MongoDB:** 3.4.4.
- **Versão Hibernate ORM:** *Hibernate-core* 4.3.x.
- **Versão Hibernate OGM:** *Hibernate-ogm-core* 5.1.0.
- **Versão JPA:** 2.1.

O JMeter, por padrão, é configurado para utilizar apenas 512 Mb de memória *heap* para o Java, isso quer dizer que mesmo com 8 Gb de memória disponível na máquina não mais que 515 Mb são utilizados nos testes. Desse modo foi modificada a configuração do *script* de

inicialização do JMeter, pois é neste *script* que a informação de memória *heap* é definida. No campo *"set HEAP"* do *script* foi substituído o valor 512m para 6144m, o que indica 6 Gb para o *heap* do Java.

### 3.4.2 Implementação do Método *runTest*

Para a execução dos testes o JMeter utiliza de uma requisição Java através do método *runTest*. Este método foi implementado para cada teste.

Para testar o desempenho da camada de persistência para a operação de inserção foi utilizados os arquivos de inserção criados na aplicação de preenchimento do banco, apresentada na seção 3.3.3. São diversos arquivos de inserção, cada um contém o número de registros a serem inseridos no banco de cados, ou seja, para o testes em que cada usuário deve inserir 10 registros o arquivo correspondente contém 10 linhas, cada uma delas com todas as informações necessárias para a inserção de um novo registro de filme no banco de dados. Esses dados são carregados do arquivo e são inseridos no banco de dados, como mostrado no código da Figura 31. Neste código nota-se que o tempo é contado apenas entre o início da transação e o seu final, ou seja, apenas o tempo da operação em si é calculado, toda a preparação dos dados não entra no cálculo do tempo de operação.

```
result.sampleStart();
con.getCon().getTransaction().begin();

moviesToBeInserted.forEach((Movies iterator) -> {
    mdbm.insertMovie(iterator, con);
});

con.getCon().getTransaction().commit();
result.sampleEnd();
```

**Figura 31 – Código de Teste de Inserção**

**Fonte: Autoria Própria.**

A Figura 32 apresenta o código de cálculo de atualização de dados no banco de dados. Assim como para o teste de inserção, os testes de atualização utilizaram um arquivo com as informações necessárias para realizar os testes. Neste caso as linhas dos arquivos continham informações para modificação de alguns campos de registros já existentes no banco de dados. O registro a ser atualizado é escolhido aleatoriamente entre os 55000 de filmes contidos no banco de dados, que foram adicionados previamente aos testes.

```
result.sampleStart();
con.getCon().getTransaction().begin();

moviesToBeUpdated.forEach((iterator) -> {
    moviesDBM.insertMovie(iterator, con);
});

con.getCon().getTransaction().commit();
result.sampleEnd();
```

**Figura 32 – Código de Teste de Atualização**

**Fonte: Autoria Própria.**

Por fim o teste de busca seguiu o mesmo padrão dos testes anteriores. O arquivo de testes de busca foram os mesmos utilizados para o testes de inserção, pois estes contém os dados disponíveis no banco, assim é realizada uma busca que sempre retornará resultados. A Figura 33 apresenta o código de contagem de tempo para a busca simples de informações (Figura 33a) e busca com *join* (Figura 33b) no banco de dados.

```
result.sampleStart();

retrieveInfo.forEach((iterator) -> {
    m = moviesDBM.retrieveMovieByTitle(iterator[1], con);
    movies.add(m);
});

result.sampleEnd();
```

**(a) Pesquisa Simples**

```
result.sampleStart();

retrieveInfo.forEach((iterator) -> {
    m = moviesDBM.retrieveOneJoin(iterator[1],
            iterator[6], iterator[9], iterator[11],
            iterator[12], iterator[13], con);
    movies.add(m);
});

result.sampleEnd();
```

**(b) Pesquisa com Join**

**Figura 33 – Código de Teste de Pesquisa**

**Fonte: Autoria Própria.**

Todos os testes realizados utilizam-se dos códigos do *database manager* que contém as *queries* de inserção, atualização e busca. As figuras 31, 32 e 33 apresentam apenas o teste que utilizam essas *queries*. O *database manager* e suas *queries* são apresentados na seção 3.3.2. O banco de dados foi previamente configurado para cada operação. Para a operação de inserção, a entidade *movies* não continha registro no início dos testes. Para as operações de atualização e busca, foi inserido previamente 55000 registros na entidade *movies*. As outras entidades, em todos os testes, foram preenchidas previamente e não sofreram modificações durante os testes.

# 4 RESULTADOS E DISCUSSÕES

Neste capítulo serão apresentados e discutidos os resultados obtidos nos testes da camada de persistência realizados sobre os gerenciadores de banco de dados MySQL e MongoDB para as operações de inserção, atualização e busca.

## 4.1 RESULTADOS

Nesta seção os resultados dos testes de inserção, atualização e busca são apresentados para cada banco de dados.

### 4.1.1 Inserção

Os testes de inserção foram realizados na entidade *movies*. Essa entidade inicialmente não continha nenhum registro salvo em nenhum dos bancos de dados. Foram realizados quatro testes em cada banco, sendo o primeiro com um usuário e o último com 5000 usuários. Cada teste foi subdivido em 5 sub-testes, cada um com um número de operações diferentes, iniciando em 10 operações e finalizando com 100 operações por usuário.

O primeiro teste realizado foi com apenas um usuário e o tempo de execução en cada banco de dados é mostrado na Tabela 14.

**Tabela 14 – Tempos para Inserção com Um Usuário no MySQL e MongoDB**

| Operações | MySQL (s) | MongoDB (s) | Diferença (s) |
|---|---|---|---|
| **10** | 0,130 | 0,101 | 0,036 |
| **25** | 0,180 | 0,251 | 0,071 |
| **50** | 0,168 | 0,226 | 0,058 |
| **75** | 0,166 | 0,237 | 0,071 |
| **100** | 0,172 | 0,256 | 0,084 |

A Figura 34 apresenta graficamente o resultado da execução do teste de inserção para um usuário.

Para um usuário executando operações sobre o banco de dados observa-se um comportamento bastante estável do tempo de execução. Com o número baixo de operações o MongoDB se mostrou mais veloz para inserir dados no banco, porém ao aumentar o número de operações o MongoDB se mostrou mais lento que o MySQL.

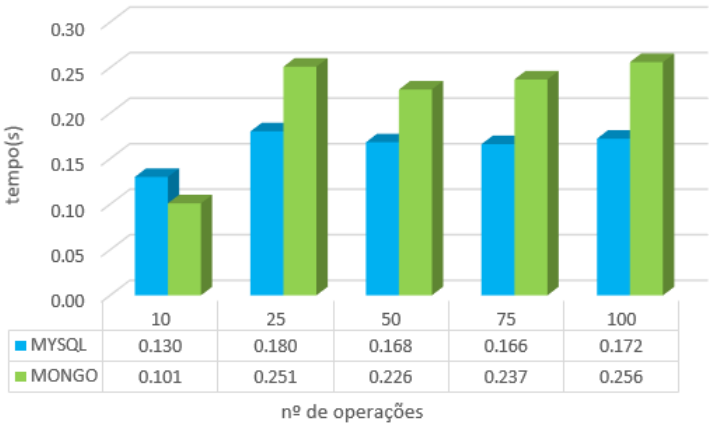



**Figura 34 – Resultados Inserção Um Usuário**

**Fonte: Autoria Própria.**

Ao executar os testes com 100 usuários realizando operações de inserção, obteve-se o tempo médio de cada execução conforme apresentado na tabela.

**Tabela 15 – Tempos Médios para Inserção com 100 Usuários no MySQL e MongoDB**

| Operações | MySQL (s) | MongoDB (s) | Diferença (s) |
|---|---|---|---|
| **10** | 0,086 | 0,063 | 0,098 |
| **25** | 0,151 | 0,115 | 0,036 |
| **50** | 0,158 | 0,157 | 0,001 |
| **75** | 0,188 | 0,215 | 0,027 |
| **100** | 0,189 | 0,298 | 0,109 |

A Figura 35 apresenta o resultado dos testes de inserção realizados com 100 usuários.

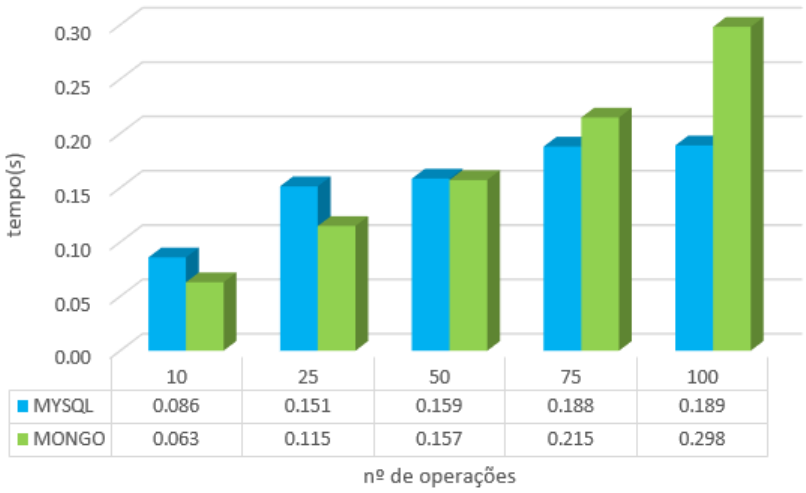



**Figura 35 – Resultados Inserção 100 Usuários**

**Fonte: Autoria Própria)**

A Tabela 16 apresenta os tempos médios de execução dos testes utilizando 1000 usuários. Neste teste notou-se que o MongoDB não conseguiu concluir o teste quando cada usuários realizou 100 operações. O valor médio apresentado é referente apenas a 5% dos usuários previstos para realizar o teste.

**Tabela 16 – Tempos Médios para Inserção com 1000 Usuários no MySQL e MongoDB**

| Operações | MySQL (s) | MongoDB (s) | Diferença (s) |
|---|---|---|---|
| **10** | 0,139 | 0,132 | 0,007 |
| **25** | 0,144 | 0,248 | 0,104 |
| **50** | 0,172 | 0,388 | 0,215 |
| **75** | 0,207 | 2,236 | 2,029 |
| **100** | 0,279 | 2447,650* | 2447,370* |

***55 dos 1000 usuários concluíram a operação de inserção.**

A Figura 36 apresenta o gráfico do testes de inserção com 1000 usuários. É importante ressaltar que neste gráfico o valor referente ao tempo médio de execução de 100 operações é baseado apenas nos 55 usuários que conseguiram terminar a operação de inserção, ou seja, é preciso cuidado ao comparar com o tempo de execução do MySQL, em que todos os usuários foram capazes de terminar suas operações em poucos segundos. O gráfico foi plotado desta maneira para mostrar como o banco de dados NoSQL se comporta bem abaixo do ideal nestas condições.

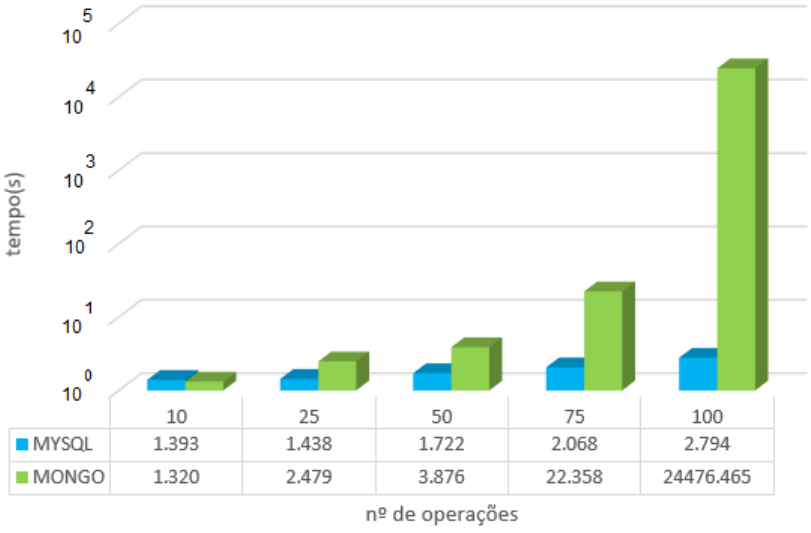



**Figura 36 – Resultados Inserção 1000 Usuários**

**Fonte: Autoria Própria.**

A Tabela 17 apresenta os resultados para o teste de inserção com 5000 usuários. Neste teste o problema visto no teste com 1000 usuários realizando 100 operações no MongoDB se repete, muitos usuários tentam inserir no banco de dados mas nenhum consegue concluir a ope-

ração. Nenhum teste foi concluído com 5000 usuários no MongoDB. O MySQL por sua vez, é capaz de executar todas as operações em cerca de um quarto de segundo.

**Tabela 17 – Tempos Médios para Inserção com 5000 Usuários no MySQL e MongoDB**

| Operações | MySQL (s) | MongoDB (s) | Diferença (s) |
|---|---|---|---|
| **10** | 0,123 | 66,317* | 66,194* |
| **25** | 0,140 | - | - |
| **50** | 0,174 | - | - |
| **75** | 0,215 | - | - |
| **100** | 0,208 | - | - |

***179 dos 5000 usuários concluíram a operação de inserção**

### 4.1.2 Atualização

Os testes de atualização, assim como o teste de inserção, foram realizados sobre a entidade *movies*. Para essa operação a entidade foi populada com 55000 registros e os registros atualizados foram escolhidos aleatoriamente no momento da realização da operação. Foram realizados quatro testes com um, cem, mil e cinco mil usuários respectivamente, cada usuário executando entre dez e cem operações.

O primeiro teste foi realizado simulando apenas um usuário atuando na aplicação que executa sobre os bancos de dados MySQL e MongoDB, afim de analisar o desempenho em tempo real sem levar em conta concorrência.

A Tabela 18 mostra os tempos de execução para um usuário ao realizar as operações.

**Tabela 18 – Tempos de Atualização para Um Usuário no MySQL e MongoDB**

| Operações | MySQL (s) | MongoDB (s) | Diferença (s) |
|---|---|---|---|
| **10** | 0,084 | 0,018 | 0,097 |
| **25** | 0,064 | 0,220 | 0,156 |
| **50** | 0,064 | 0,225 | 0,191 |
| **75** | 0,096 | 0,341 | 0,245 |
| **100** | 0,083 | 0,367 | 0,284 |

A Figura 37 mostra o gráfico dos tempos de execução de um usuário executando operações de atualização.

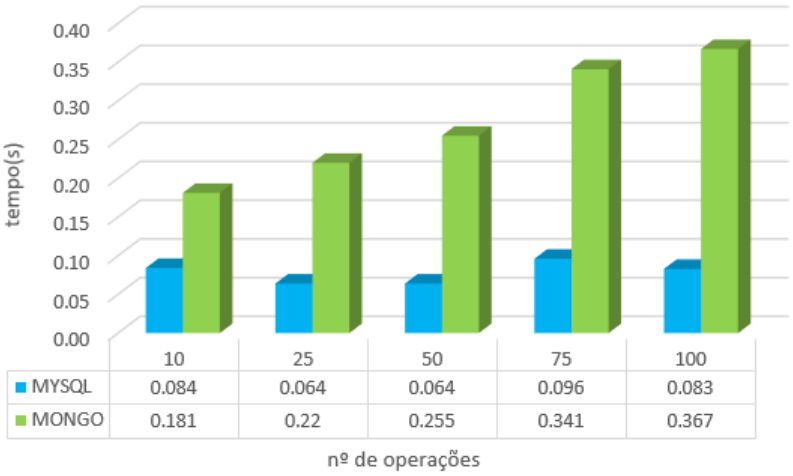



**Figura 37 – Resultados Atualização Um Usuário**

**Fonte: Autoria Própria.**

A Tabela 19 apresenta os tempos médios de execução dos testes com 100 usuários.

**Tabela 19 – Tempos Médios de Atualização para 100 Usuário no MySQL e MongoDB**

| Operações | MySQL (s) | MongoDB (s) | Diferença (s) |
|---|---|---|---|
| **10** | 0,039 | 0,055 | 0,016 |
| **25** | 0,037 | 0,100 | 0,063 |
| **50** | 0,045 | 0,203 | 0,158 |
| **75** | 0,052 | 0,281 | 0,230 |
| **100** | 0,057 | 0,631 | 0,304 |

A Figura 38 mostra em forma de gráfico os tempos de execução contidos na Tabela 19.

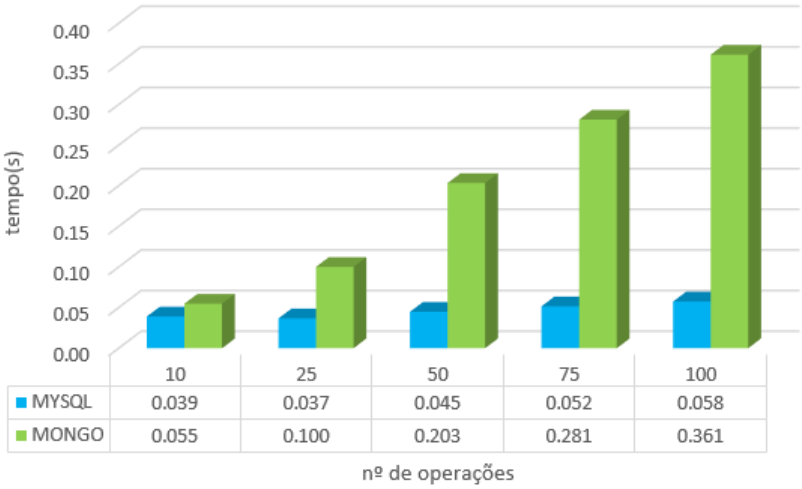



**Figura 38 – Resultados Atualização 100 Usuários**

**Fonte: Autoria Própria.**

Os resultados dos testes com 1000 usuários realizando operações nos bancos de dados são apresentados na Tabela 20.

**Tabela 20 – Tempos Médios de Atualização para 1000 Usuário no MySQL e MongoDB**

| Operações | MySQL (s) | MongoDB (s) | Diferença (s) |
|---|---|---|---|
| **10** | 0,029 | 0,046 | 0,017 |
| **25** | 0,039 | 0,103 | 0,064 |
| **50** | 0,047 | 0,204 | 0,157 |
| **75** | 0,053 | 0,282 | 0,229 |
| **100** | 0,060 | 0,368 | 0,308 |

Esses resultados foram apresentados em forma de gráfico para uma melhor visualização dos dados. Este gráfico é apresentado na Figura 39.

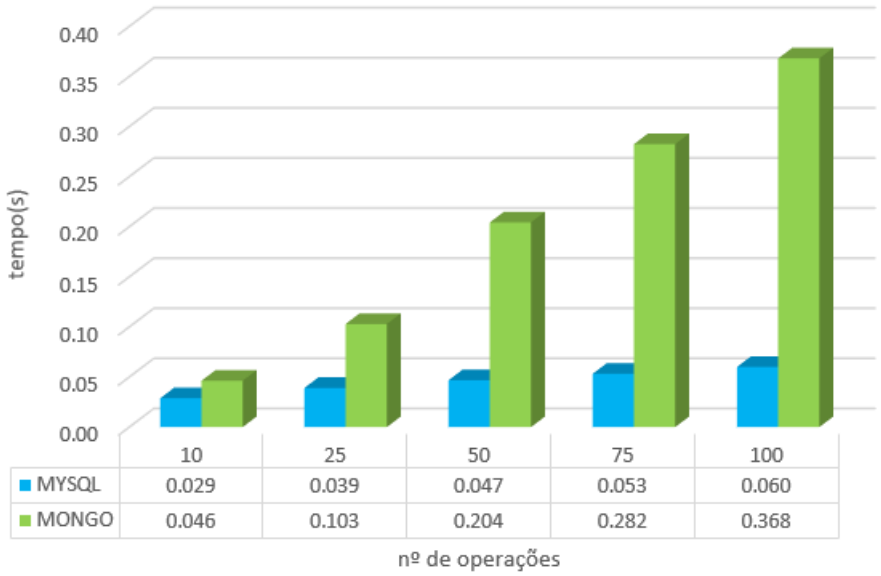



**Figura 39 – Resultados Atualização 1000 Usuários**

**Fonte: Autoria Própria.**

Os resultados dos testes com 5000 usuários atualizando registros no MongoDB e no MySQL através de suas camadas de persistência, são apresentados na Tabela 17.

**Tabela 21 – Tempos Médios de Atualização para 5000 Usuário no MySQL e MongoDB**

| Operações | MySQL (s) | MongoDB (s) | Diferença (s) |
|---|---|---|---|
| **10** | 0,028 | 0,046 | 0,018 |
| **25** | 0,036 | 0,108 | 0,072 |
| **50** | 0,045 | 0,198 | 0,154 |
| **75** | 0,053 | 0,286 | 0,232 |
| **100** | 0,062 | 0,369 | 0,307 |

A Figura 40 representa graficamente os resultados dos testes de atualização realizados com 5000 usuários.

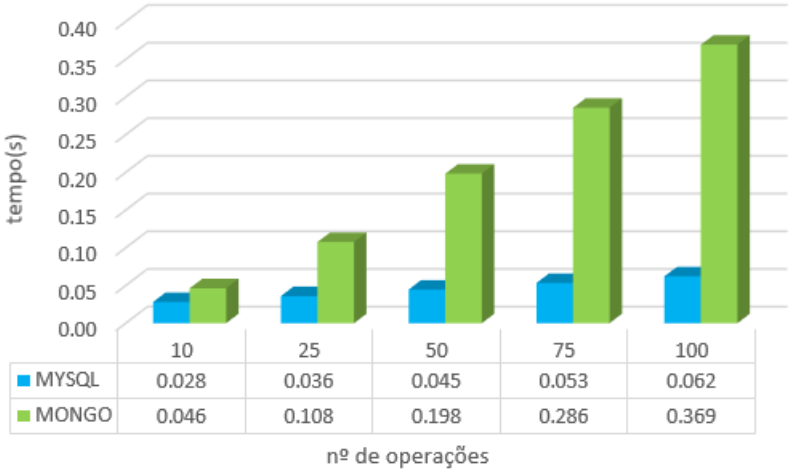



**Figura 40 – Resultados Atualização 5000 Usuários**

**Fonte: Autoria Própria.**

### 4.1.3 Busca

Buscas são operações custosas, ou seja, levam bastante tempo para executar. Foram realizados testes com um, cinquenta e cem usuários. Essa redução do número de usuários foi motivado pois os testes com números elevados de usuários, em nenhum banco de dados, não foram concluídos. Portanto foi reduzidos os usuários para montar uma base viável de resultados para comparação entre os bancos de dados.

Foram realizados dois testes de buscas. O primeiro um teste de busca simples, utilizando a *query* apresentada na Figura 29a e o segundo realiza uma busca com *join*, utilizando a *query* da Figura 29b.

#### 4.1.3.1 Busca Simples

Foram realizados os testes com 1, 50 e 100 usuários simultâneos. Cada usuário, assim como nas operações anteriores, realizam testes com 10, 25, 50, 75 e 100 operações.

A Tabela 22 apresenta os tempos do teste com um único usuário realizando operações nos bancos de dados. Os tempos de execução com um usuário representa o tempo real das operações nos bancos, pois não existe concorrência.

A Figura 41 apresenta graficamente os dados contidos na Tabela 22 para a execução do

**Tabela 22 – Tempos de Busca Simples para Um Usuário no MySQL e MongoDB**

| Operações | MySQL (s) | MongoDB (s) | Diferença (s) |
|---|---|---|---|
| **10** | 0,084 | 0,018 | 0,097 |
| **25** | 0,064 | 0,220 | 0,156 |
| **50** | 0,064 | 0,225 | 0,191 |
| **75** | 0,096 | 0,341 | 0,245 |
| **100** | 0,083 | 0,367 | 0,284 |

teste de busca simples com um usuário.

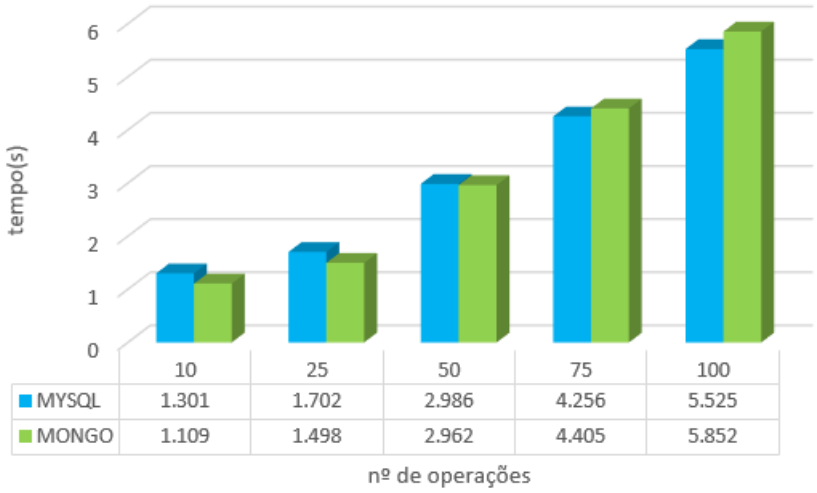



**Figura 41 – Resultados Busca Simples com Um Usuário**

**Fonte: Autoria Própria.**

A Tabela 23 contém os tempos dos testes realizados com 50 usuários tentando buscar registros através do título no banco de dados.

**Tabela 23 – Tempos Médios de Busca Simples para 50 Usuários no MySQL e MongoDB**

| Operações | MySQL (s) | MongoDB (s) | Diferença (s) |
|---|---|---|---|
| **10** | 2,529 | 0,811 | 1,718 |
| **25** | 1,980 | 1,172 | 0,708 |
| **50** | 15,363 | 3,436 | 11,930 |
| **75** | 44,447 | 20,073 | 24,370 |
| **100** | 61,842 | 35,709 | 26,130 |

A Figura 42 apresenta o gráfico dos resultados e evidencia o comportamento dos tempos de busca nos bancos de dados.

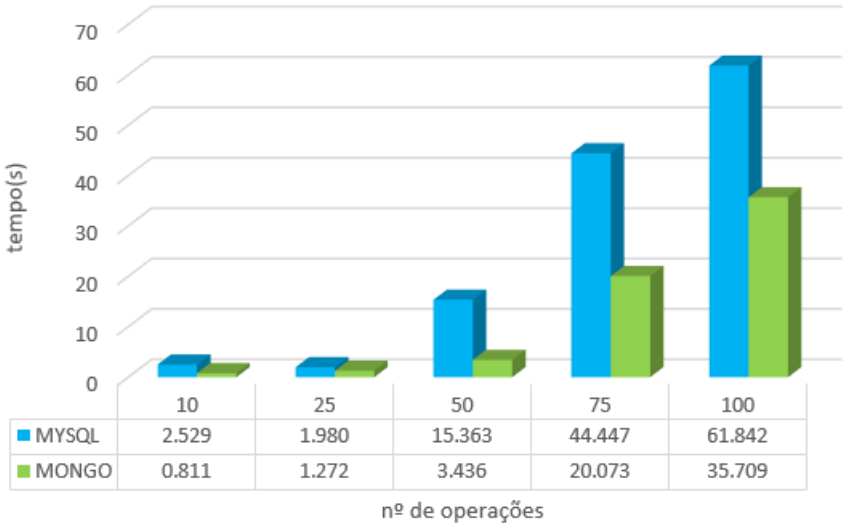



**Figura 42 – Resultados Busca Simples com 50 Usuários.**

**Fonte: Autoria Própria.**

Por fim foi realizado o teste com 100 usuários realizando buscas simples no MySQL e no MongoDB. Os tempos médios deste teste são apresentados na Tabela 24. Para 100 operações o teste não foi concluído quando executado sobre o MySQL, apenas 32 dos 100 usuários conseguiram completar suas operações.

**Tabela 24 – Tempos Médios de Busca Simples para 100 Usuários no MySQL e MongoDB**

| Operações | MySQL (s) | MongoDB (s) | Diferença (s) |
|---|---|---|---|
| **10** | 0,888 | 0,672 | 0,216 |
| **25** | 1,855 | 1,478 | 0,377 |
| **50** | 53,484 | 3,524 | 49,960 |
| **75** | 219,845 | 24,759 | 195,086 |
| **100** | 1702,063* | 35,595 | 1666,468* |

***32 dos 100 usuários concluíram a operação de busca simples.**

A Figura 43 apresenta de forma gráfica o resultado desse teste. Ela é apresentada em escala logarítmica pois a variação no tempo de pesquisa nos testes é muito elevado. Apesar de o teste com 100 operações por usuário não ter sido completado, o tempo médio de busca dos 32 usuários que conseguiram concluir as operações foi calculado e o resultado foi plotado para efeito de observação do tempo, porém a comparação com o tempo de execução da busca no MongoDB deve ser feita com ressalvas.

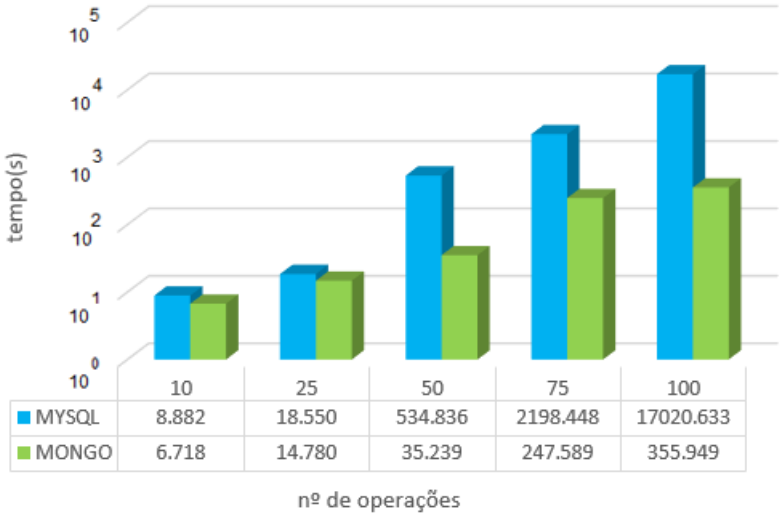



**Figura 43 – Resultados Busca Simples com 100 Usuários**

**Fonte: Autoria Própria.**

#### 4.1.3.2 Busca com *Join*

Para o segundo teste realizado com a operação de busca foi utilizado uma *query* contendo uma operação de *join*. Este teste foi idealizado pois em toda aplicação rodando com banco de dados relacional possui operações *join* em suas *queries*, tornando assim este teste muito importante para veriaficar a viabilidade da migração de base de dados em sistemas legados.

A Tabela 25 apresenta os tempos obtidos na realização de operação de busca com um usuário nos bancos relacional e NoSQL.

**Tabela 25 – Tempos de Busca com *join* para Um Usuário no MySQL e MongoDB**

| Operações | MySQL (s) | MongoDB (s) | Diferença (s) |
|---|---|---|---|
| **10** | 0,919 | 1,038 | 0,119 |
| **25** | 1,573 | 1,635 | 0,062 |
| **50** | 2,479 | 3,271 | 0,792 |
| **75** | 2,949 | 4,931 | 1,982 |
| **100** | 3,958 | 6,487 | 2,529 |

A Figura 44 apresenta o gráfico dos tempos de busca com *join* em ambos os bancos de dados com apenas um usuário.

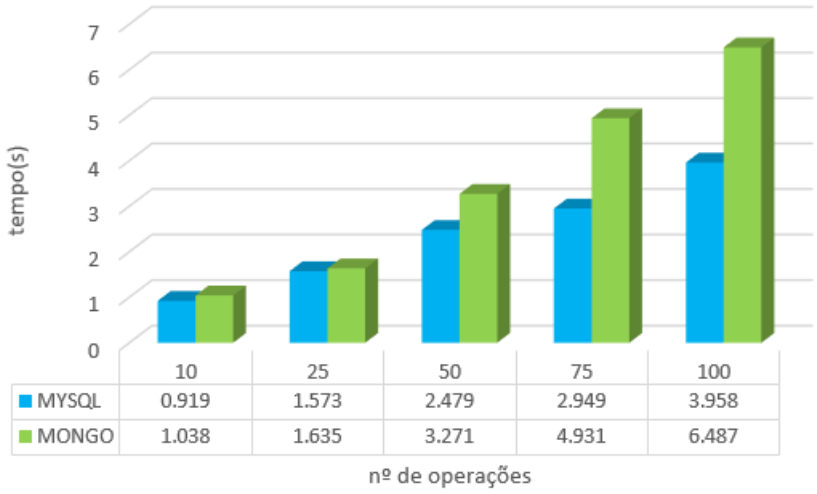



**Figura 44 – Resultados Busca com *join* com Um Usuário**

**Fonte: Autoria Própria.**

O segundo teste foi com 50 usuários realizando buscas com *join* no MySQL e no MongoDB. A Tabela 26 apresenta os resultados desses testes.

**Tabela 26 – Tempos Médios de Busca com *join* para 50 Usuários no MySQL e MongoDB**

| Operações | MySQL (s) | MongoDB (s) | Diferença (s) |
|---|---|---|---|
| **10** | 0,857 | 0,734 | 0,122 |
| **25** | 1,223 | 1,246 | 0,022 |
| **50** | 2,857 | 3,543 | 0,686 |
| **75** | 9,926 | 21,302 | 11,376 |
| **100** | 12,337 | 36,012 | 23,675 |

A Figura 45 apresenta o gráfico dos resultados desse teste.

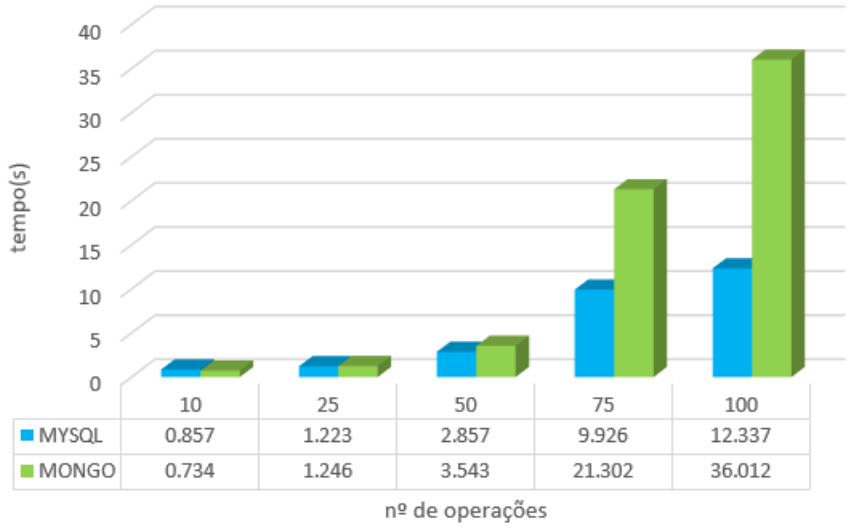



**Figura 45 – Resultados Busca com *join* com 50 Usuários**

**Fonte: Autoria Própria.**

A Tabela 27 apresenta os tempos médios de execução do terceiro teste para buscas com *join* em que 100 usuários compartilham o acesso ao banco de dados.

**Tabela 27 – Tempos Médios de Busca com *join* para 100 Usuários no MySQL e MongoDB**

| Operações | MySQL (s) | MongoDB (s) | Diferença (s) |
|---|---|---|---|
| **10** | 0,709 | 0,713 | 0,004 |
| **25** | 1,344 | 1,812 | 0,468 |
| **50** | 2,449 | 3,361 | 0,912 |
| **75** | 8,326 | 35,944 | 27,618 |
| **100** | 26,174 | 77,380 | 51,206 |

A Figura 46 apresenta os resultados para os testes com 100 usuários de forma gráfica, permitindo uma melhor visualização do comportamento do tempo médio de execução da *query* de busca com o operador *join*.

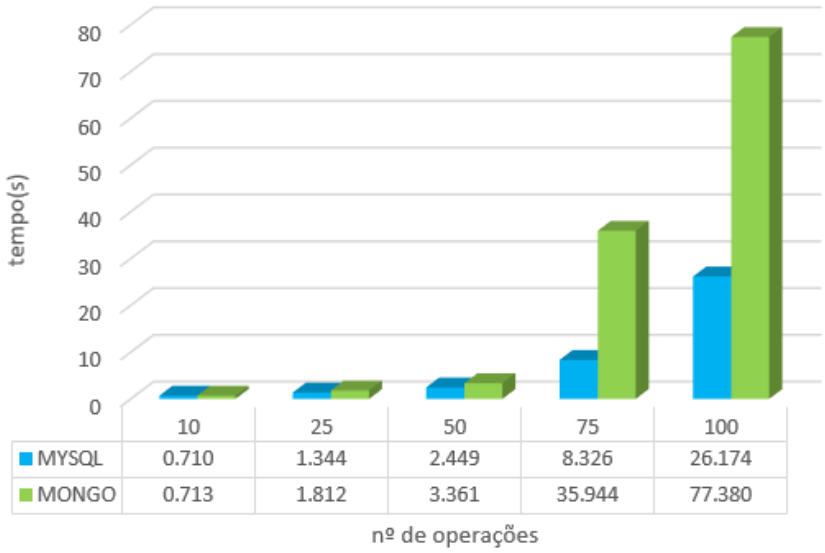



**Figura 46 – Resultados Busca com *join* com 100 Usuários**

**Fonte: Autoria Própria.**

## 4.2 DISCUSSÕES

Nesta seção são apresentadas as discussões referentes aos resultados de todos os testes realizados sobre as camadas de persistência que atuam sobre o MySQL e o MongoDB.

### 4.2.1 Inserção

O teste realizado com apenas um usuário mostra o comportamento do banco de dados quando não existe nenhuma concorrência, ou seja, mostra o tempo realmente necessário para concluir a operação. Quando adicionado mais usuários, os tempos de inserção aumentam pois quando um usuário tenta inserir algo no banco ele pode ter que esperar outro usuário concluir sua operação para então conseguir de fato inserir o dado. Esta espera causa o aumento no tempo quando os testes são realizados com múltiplos usuários.

Com apenas um usuário o comportamento de ambos os bancos é estável quando o número de operações é maior que 25. Com apenas 10 operações o MongoDB se mostrou melhor, porém nos demais testes é mais lento sendo 1,4 vezes mais lento, em média, que o MySQL.

Quando 100 usuários realizam inserções no banco de dados o MongoDB tem melhor desempenho quando o número de operações é menor, porém ao aumentar o número de operações que cada usuário realiza o MongoDB perde, e muito, o desempenho de inserção e para cem operações o MongoDB chega a ficar 60% mais lento comparado ao MySQL. Esse comportamento fica claro ao observar o gráfico da Figura 35.

Com 1000 usuários inserindo dados no banco de dados o tempo para conclusão da operação pelo MongoDB se torna extremamente alto, até o ponto em que, com 100 operações por usuário, o teste não é concluído devido a falta de memória do computador em que o teste foi realizado. Isso ocorreu pois o tempo para inserção se tornou alto demais, impedindo que o usuário finalize seu processo e por se tratar de uma operação com muitos usuários, outros usuários também não conseguem terminar suas operações devido a concorrência no banco de dados e cada usuário utiliza-se de recursos da máquina. Eventualmente, quando há um acumulo de usuários que não conseguem concluir suas operações a memória disponível acaba e todo o processo finaliza sem que o teste seja concluído inteiramente.

Mesmo com apenas 55 usuários concluindo a operação de inserção no MongoDB para 100 operações, fica visível, através da Figura 36, que o tempo para inserção neste banco de dados é muito mais elevado em comparação com o MySQL. O gráfico foi apresentado em escala logarítmica para uma melhor visualização dos resultados, pois a variação nos tempos dos testes foi extremamente alta, dificultando assim, a visualização na escala decimal.

Os testes com 5000 usuários não gerou dados para comparação entre os bancos de dados, pois o MongoDB com Hibernate OGM não conseguiu concluir nenhum teste de inserção.

### 4.2.2 Atualização

Através dos dados extraídos no teste com um usuário, que podem ser visualizados na Tabela 18, pode-se verificar que o tempo para realização das operação pela camada de persistência agindo sobre o MongoDB é cerca de 3 vezes mais demorado ao comparado com o tempo de execução sobre o MySQL. Ao aumentar o número de operações a média da diferença do tempo de execução entre os bancos é de 0,19 segundos. No gráfico mostrado na Figura 37, pode-se verificar de forma clara que a camada de persistência sobre o MongoDB tem um desempenho inferior ao MySQL no quesito tempo de execução.

Com base nos tempos obtidos no teste de atualização com 100 usuários percebeu-se que a camada de persistência sobre o MongoDB executa as *queries* com tempos superiores aos obtidos executando sobre o MySQL. A execução de atualização no MongoDB foi aproximadamente 4 vezes mais demorada em relação a execução sobre o MySQL. Com o aumento das operações a média da diferença de execução entre os bancos foi de 0,15 segundos.

Com base na visualização do gráfico da Figura 38 verifica-se que a camada de persistência sobre o MongoDB tem um desempenho inferior ao MySQL, e também que o tempo de execução é altamente prejudicado conforme o aumento do número de operações, pois a diferença entre o tempo de média de execução para 10 operações e para 1000 para o MySQL é de 18,38 milissegundos enquanto para o MongoDB a diferença é de 306 milissegundos. Através dos resultados deste teste, notou-se que a camada de persistência juntamente com o MySQL apresenta-se bem mais estável durante sua execução independente do aumento do número de

operações.

Observou-se que para 1000 e 5000 usuários os tempos de execução assemelham-se muito aos obtidos com 100 usuários, sendo que a maior diferença de tempo na execução destes testes foi de aproximadamente 10,38 milissegundos, ou seja, ao realizar atualização sobre o MySQL e sobre o MongoDB praticamente não houve diferença de resultado ao aumentar o número de usuários.

### 4.2.3 Busca Simples

Ao analisar os dados da Tabela 22 nota-se que a camada de persistência consegue manter um equilíbrio no tempo de busca em ambos os bancos de dados quando apenas um usuário está realizando operações. Em todos os testes com um usuário a diferença no tempo de busca entre o MySQL e o MongoDB não passa de 15%. A média da variação do tempo de pesquisa entre o MongoDB e o MySQL é de 0,96 segundos. O que percebe-se, porém, é um aumento significativo do tempo de busca quando comparado o número de operações realizadas. Entre o teste com busca de 10 registros e a busca com 100 registros houve um aumento quase 5 vezes no tempo de execução da *query* em ambos os bancos.

O gráfico da Figura 41 deixa claro o comportamento dos bancos de dados ao realizar uma busca, ambos possuem tempos similares de execução, porém a cada mudança na quantidade de buscas realizadas é visível o aumento no tempo de execução. O gráfico também ajuda a visualizar quem quando o número de operações é mais baixa o MongoDB é ligeiramente mais rápido que o MySQL para realizar uma pesquisa por nome, porém ao final o MySQL tem a vantagem, apesar de pequena, no tempo de pesquisa.

Com os resultados dos testes com 50 usuários, mostrados na Tabela 23, é possível ver que os tempos de pesquisa não são mais semelhantes entre os banco de dados. A diferença média entre os tempos de execução entre o MySQL e o MongoDB, que no teste anterior com apenas um usuário, não passou de um segundo, neste teste está na casa dos 12 segundos. Outra informação que se obtém desta tabela é que o tempo de busca começa aumentar, principalmente para o MySQL, a partir de 50 usuários e o tempo praticamente triplica quando o teste é realizado com 100 operações por usuário.

O gráfico mostrado na Figura 42 chama bastante atenção pelo comportamento do MySQL em relação a buscas simples. O tempo de busca é extremamente alto quando são executados muitos usuários e cada um realizando 100 operações. A diferença de tempo entre o MySQL e o MongoDB neste teste chega a ser de 26 segundos, uma diferença de 43% no tempo da operação de busca.

Os testes com 100 usuários, assim como o teste anterior com 50 usuários, mostrou-se bastante ruim no tempo de busca para o MySQL em relação ao MongoDB principalmente

quando o número de operações é mais alto. Quando realizado com 100 operações o teste sobre o MySQL não foi concluído, apenas 32% dos usuários conseguiram finalizar suas operações. O problema encontrado, foi o mesmo já discutido nos testes de inserção. O tempo para buscar os registros são muito altos e a cada segundo um novo usuário também realiza buscas no banco de dados até o limite em que muitos usuários estão tentando realizar suas operações o e sistema fica sem memória disponível, o que faz com que o teste seja finalizado antes do esperado. O que diferencia este teste do teste de inserção é que na inserção o banco de dados que não consegue realizar todas as operações é o MongoDB. No teste de busca simples é o MySQL que não suporta as operações e o teste é encerrado.

A escala logarítmica do gráfico mostrado na Figura 43 deixa claro o aumento no tempo de execução da *query* de pesquisa simples. Desconsiderando a diferença entre os tempos de execução com 100 operações, pois este teste não foi 100% concluído no MySQL, a média da diferença de tempo de execução entre o MongoDB e o MySQL é de 61,4 segundos. Porém, ao levar em consideração o teste com 100 operações, mesmo que não tenha sido concluído no MySQL, a diferença média entre os tempos dos bancos de dados foi de 382,4 segundos, ou seja, em média, o MySQL levou 6 minutos a mais para executar as buscas quando comparado ao MongoDB.

### 4.2.4 Busca com *join*

Os resultados do teste de busca com inserção, que foram apresentados na Tabela 25, mostram a evolução dos tempos de acordo com a quantidade de operações. Diferentemente do teste de busca simples com um usuário, neste não existe uma similaridade nos tempos de busca. Com o número de operações baixo a diferença entre os dois bancos não passa de 0,1 segundo, porém a partir de 50 operações esta diferença aumenta bastante e chega a 2 segundos, isto quer dizer que o MongoDB chega a ser 60% mais lento ao fazer 100 pesquisas no banco de dados utilizando uma *query* com *join*. Essa variação é visível no gráfico apresentado na Figura 44. O gráfico mostra também a grande variação no tempo de execução da busca no MongoDB quando aumenta-se o número de operações. O tempo de execução da busca no MySQL também aumenta, porém este tempo no MongoDB cresce cerca de 6 vezes entre 10 e 100 operações, enquanto o MySQL tem um aumento de aproximadamente 4 vezes na mesma variação de operações.

A Tabela 26 mostra que a diferença entre os tempos de execução entre o MongoDB e o MySQL aumentam consideravelmente quando os testes são realizados com 50 usuários, principalmente quando cada um executa 100 operações. A diferença no tempo de execução chaga a 23 segundos entre os bancos. Outra informação importante obtida nesta tabela é o grande aumento de tempo de execução entre a quantidade de operação. O MongoDB tem uma proporção de aumento de 50 vezes entre o primeiro e o último teste realizado com 50 usuários. O MySQL também apresenta um aumento no tempo, porém em uma proporção muito menor, de aproxi-

madamente 14 vezes. A Figura 45 apresenta de forma visual como o tempo de execução do MongoDB aumenta muito. Um aumento no tempo de execução era esperado, pois o número de operações por usuário é maior, porém a proporção de aumento visto no MongoDB foi extremamente alto e não linear, ou seja, não aumenta proporcionalmente ao aumento de operações. O MySQL, como esperado em uma busca com *join*, tem um aumento menor do tempo de execução com o aumento de operações.

Os testes com 100 usuários obtiveram o mesmo comportamento observado nos testes com 50 usuários. O MongoDB com Hibernate OGM aumenta drasticamente o tempo de execução para 100 operações. A proporção de aumento do tempo de execução entre 10 e 100 operações com 100 usuários para o MongoDB foi de 83 vezes, enquanto no MySQL a proporção foi aproximadamente 36 vezes. O gráfico da Figura 46 evidencia o aumento drástico no tempo de execução no Hibernate OGM com MongoDB. Até 50 operações o tempo de pesquisa é baixo e muito próximo ao tempo de execução no MySQL, porém com 75 e 100 o banco NoSQL fica muito mais lento chegando a até 27 segundos de diferença em relação a busca no banco relacional.

# 5 CONCLUSÕES

A realização de testes sobre camada de persistência de banco de dados envolvendo diversas operações foi concluída completamente e os resultados obtidos foram consistentes e permitiram uma análise do desempenho do MySQL rodando sob o Hibernate ORM e do MongoDB sob o Hibernate OGM.

Os testes de inserção mostraram como a camada de persistência lidou com a variação do número de usuários e de operações sobre os bancos de dados utilizados. Observou-se com apenas um usuário ambos os bancos têm um comportamento estável de acordo com o aumento das operações. Porém o MongoDB mostrou-se eficiente somente com poucas operações por usuário. Quando o número de operações cresce seu desempenho cai, se mostrando inferior ao MySQL. Com o aumento significativo de usuários realizando as operações de inserção, o MongoDB com Hibernate OGM se mostra extremamente lento, apresentando uma grande diferença no tempo de execução quando comparado ao MySQL. O tempo de execução se mostra tão elevado que em determinado ponto da execução o teste não é executado por completo por falta de memória da máquina utilizada para teste. Isso ocorre pois a cada 700 milissegundos um novo usuário é iniciado para realizar inserções no banco e como o tempo para a inserção é muito alto cada vez mais usuários iniciam sua execução sem que os anteriores finalizem. Apesar do teste com 1000 usuários não ter completado as 100 operações notou-se que o desempenho daqueles que concluíram foi muito ruim. Enquanto os usuários concluíram as 100 inserções em um quinto de segundo em média no MySQL para o MongoDB com apenas 55 usuários concluídos dos 1000 previstos a média de tempo para as inserções foi superior a 40 minutos. O MongoDB teve um desempenho ainda pior quando o executado com 5000 usuários em que não concluiu nenhum teste por completo, completando apenas 179 usuários dos 5000 previstos que tentaram inserir 10 registros neste banco. O MySQL por sua vez concluiu todos os testes com todos os usuários com tempo extremamente satisfatório não levando mais que 0,2 segundo para a conclusão do teste com maior número de operações por usuário.

Os tempos de execução dos testes de atualização sobre as camada de persistência do MySQL se mostraram bastante estáveis durante a execução dos testes. A variação do tempo de atualização para esta banco é muito baixa com a variação do numero de operações. Isso ocorreu em todos os testes realizados independente do número de usuários realizando as operações. O MongoDB por sua vez tem uma elevação no tempo de atualização quando o número de operações aumenta. No teste com apenas um usuário a a taxa de aumento entre 10 e 100 operações é de aproximadamente 3 vezes. Quando com 100 usuários também existe uma aumento no tempo de atualização, porém em proporção muito elevado, sendo 100 inserções cerca de 11 vezes o tempo de atualização para dez operações. Para 1000 e 5000 mil usuários também percebe-se uma aumento proporcional no tempo, porém esta é maior proporção de crescimento no tempo dentre todos os testes. Ambas as camadas de persistência foram capazes de concluir todas as

operações em todos os testes não ocorrendo grandes problemas para realizar estas operações.

Os testes de busca foram realizados com menos usuários, sendo cem o maior número de usuários. Essa mudança na quantidade de usuários se deu devido ao tempo extremamente alto dos testes com muitos usuários. Este tempo elevado acarretou na parada dos testes em ambos os bancos de dados, por isso foi decidido pela redução do número de usuários para que uma comparação do desempenho das camadas de persistência fosse realizada para os testes de busca.

Os testes de busca utilizando a camada de persistência tiveram resultados muito interessantes, pois cada teste se mostrou mais eficiente em determinada camada de persistência. Para o teste utilizando uma pesquisa simples, ou seja, uma pesquisa por título sem considerar nenhuma relação, o MongoDB executando sob o Hibernate OGM se mostrou muito mais rápido que o MySQL, porém no teste com pesquisa com o operador *join* o MySQL executando sob o Hibernate ORM tem um melhor desempenho.

Com os resultados para os testes de busca simples pode-se concluir que a camada de persistência utilizada com o MySQL não consegue lidar bem com a grande quantidade de usuários realizando muitas operações. Em determinado ponto observou-se que o teste de busca simples no MySQL não foi concluído. Analisando apenas o teste com um usuário é possível afirmar que os tempos de execução são muito semelhantes, não passando de 0,3 segundos a diferença máxima entre as buscas com apenas um usuário. Ao aumentar o número de usuários buscando um registro por título no banco de dados o MySQL tem uma queda brusca de desempenho, principalmente quando cada usuário realiza mais de 50 operações. Com cinquenta usuários simultâneos o tempo de execução do MySQL já é bastante alto, mas quando o teste é realizado com cem usuários o tempo de busca cresce de maneira extremamente alta até que o teste com cem usuários e 100 operações é interrompido por falta de memória da máquina de testes. Concluiu-se que a interrupção do teste foi devido a má gerência da camada de persistência e do banco de dados para este tipo de busca. Mesmo com menos usuários e cada um iniciando com menos frequência que os testes anteriores houve um acúmulo de usuários que não conseguiam terminar suas operações causando a falta de memória da máquina. Neste mesmo cenário de testes o MongoDB também teve um aumento no tempo de execução com o aumento do número de usuários e operações, contudo este aumento não é tão significativo comparando com o tempo de execução da busca simples no MySQL. Este foi o único teste realizado em que o MongoDB teve um desempenho melhor que o MySQL utilizando a mesma aplicação com apenas a camada de persistência diferente.

O teste de busca utilizando o operador *join* foi concluído o que já era esperado: o MySQL com sua camada de persistência teve um desempenho elevado quando comparado com o MongoDB. Esta conclusão já era esperada pois o *join* é um operador nativo do paradigma relacional. Este operador é utilizado para realizar ligações entre tabelas relacionadas, e relação, como o nome já diz, é um conceito de banco de dados relacionais. Por ter que realizar diversas conversões da *query* JPQL para o dialeto nativo utilizado pelo MongoDB, o Hibernate acaba perdendo desempenho ao executar esse tipo de busca. Em todos os testes, com um, cinquenta e

cem usuários o desempenho do MongoDB com Hibernate OGM é inferior ao do MySQL, porém na execução com cem usuários a proporção entre os tempos de busca dos dois bancos chega a ser de 2,3 vezes em seu pior resultado. Para poucas operações (até 50) a diferença entre MySQL e MongoDB não é tão grande, mas com grande número de operações por usuário esta diferença cresce muito.

Através dos resultados obtidos a partir dos testes executados conclui-se que a migração para um banco de dados de paradigma NoSQL de uma aplicação totalmente planejada e estruturada para utilizar um banco de dados relacional não é viável sem que grandes modificações sejam feitas no código da aplicação ou mesmo realizando modificações a migração não seja possível.

Apesar da facilidade sugerida pelo Hibernate OGM de reutilizar toda a codificação de estruturação de acesso a um banco de dados relacional, e realmente conseguir realizar esse acesso seu desempenho é bastante prejudicado, tendo em vista que a camada de persistência terá sempre que realizar conversões internas tanto da estrutura de relacionamentos quanto das *queries* para a linguagem nativa do banco de dados NoSQL que se pretende migrar o sistema.

O Hibernate OGM é capaz de realizar a maior parte das operações como inserção, atualização e busca, porém ao realizar buscas com operadores específicos esta camada não é capaz de traduzir *queries* JPQL em *queries* nativas do MongoDB. Esse fato foi observado ao executar uma *query* com o operador *join* e seleções agregadas. Este tipo de busca é bastante comum e usada em todos os sistemas que utilizam bancos de dados relacionais, pois é através desta *query* que se busca informações de registros em tabelas que possuem relações com outras tabelas. Este se torna um grande empecilho para a migração. Além da mudança na camada de persistência isso implica em uma mudando em todo o sistema de pesquisa da aplicação, que reescrever todas as *queries* que realizam *select* agregado juntamente com o operador *join*.

Mudando somente a camada de persistência foi concluído que a nova camada para banco de dados não-relacionais é capaz de realizar parcialmente as operações JPQL que a camada de persistência para bancos relacionais é capaz de realizar.

Outro problema encontrado na migração das bases de dados modificando-se apenas a camada de persistência é a incapacidade da utilização de tipos nativos de determinado banco de dados. Por exemplo, ao migrar uma aplicação do MySQL para o MongoDB, o MySQL não pode conter nenhuma tabela com um campo do tipo *BLOB* para imagens. Este é um tipo específico do MySQL, portanto o MongoDB não o reconhece e a camada de persistência não é capaz de traduzi-lo para um tipo conhecido desta base. Para uma migração completa seria necessária a modificação da estrutura de todas as entidades que contenham um campo do tipo *BLOB* e, ao realizar essa modificação torna a migração inviável de acordo com o escopo deste trabalho.

O uso de camadas de persistência sempre é recomendado para conexão com bancos de dados, mesmo bancos não-relacionais, pois realizam um interfaceamento completo e eficaz com o banco de dados que seria muito custoso para o desenvolvedor. Porém o seu uso deve ser feito de forma especifica para o banco de dados será usado. Por exemplo se a conexão for com

o MySQL, ou outro banco de dados relacional, o projeto deve ser feito com este fim, ou seja, as *queries* para acesso ao banco devem ser criadas respeitando o paradigma a ser utilizado. Se o acesso a ser realizado for um uma base não relacional, a aplicação deve ser desenvolvida visando este paradigma. Ao ser desenvolvida para o paradgma relacional uma aplicação pode executar em alto desempenho utilizando camadas de persistência pois não irá encontrar limitações geradas pela diferença de paradigma.

Para a utilização de um banco de dados como o MongoDB com Hibernate OGM o ideal é que a aplicação seja toda desenvolvida para este banco, pois ao criar as entidades de forma que as coleções são criadas e utilizando recursos nativos o desempenho do banco sempre será superior ao tentar adaptar este paradigma a uma aplicação que não foi estruturada para ele. Os resultados apresentados neste trabalho deixaram claro que a mudança de paradigma não compensa se não é desejado realizar uma reestruturação da aplicação.

Portanto conclui-se que a troca de paradigmas de banco de dados deve ser realizada mediante a mudança na aplicação como um todo. A camada de persistência não consegue realizar um bom trabalho sem que haja a demanda de retrabalho de desenvolvimento, sendo atualmente como está estruturada a camada isto geraria custos e demanda de tempo para reescrita de codificação para que possa ter um melhor desempenho da aplicação.

## 5.1 DIFICULDADES E LIMITAÇÕES

O Hibernate OGM, em sua página na internet, descreve que seu produto é capaz de executar *queries* JPQL nas linguagens nativas de bancos de dados não-relacionais. Porém algumas dificuldades foram encontradas, principalmente ao criar operações de busca. Não foi possível criar *queries* com *select* agregados, ou seja, fazer uma busca com o operador *join* por um dado que esta é uma tabela relacionada.

Outra dificuldade encontrada para a realização do projeto foi a limitação de hardware para realizar os testes. Isso causou impacto direto nos resultados obtidos. Ao utilizar uma única máquina para executar a aplicação e gerenciar os usuários que a utilizam parte dos recursos da máquina serão utilizados em outras operações que não o teste em si. Cada usuário criado utiliza memória RAM e processamento, com o excesso de usuários ativos, que ocorre quando uma operação sobre o banco de dados demanda um alto tempo para conclusão esses recursos se esgotam,pois cada processos demora bastante tempo para liberar o recurso, sendo assim a aplicação testada fica sem os recursos mínimos para um funcionamento perfeito, alterando assim alguns resultados.

Criar uma aplicação que reflete a realidade de aplicações comerciais é um desafio, pois existem inúmeras formas de criar e gerenciar bancos de dados, portanto a escolha da estrutura de dados a serem armazenados (o que armazenar no banco e que relações possuem) foi feita com bastante cuidado. Esta estrutura não deveria ser simples demais pois não refletiria a realidade do

mercado e também não poderia ser extremamente complexa inviabilizando sua construção no tempo previsto para a realização do trabalho.

## 5.2 TRABALHOS FUTUROS

A realização de testes utilizando equipamentos apropriados pode ser realizado para evitar problemas com falta de recursos ao rodar os testes. Utilizando uma máquina simples para os testes pode ter causado alguns erros que podem ser evitados usando servidores mais adequados para testes. Muitos usuários rodando na mesma máquina provoca um utilização de recursos muitos grande, que não são utilizados corretamente no teste. Um uma ação real de multi-usuários cada requisição realizar operações em um banco de dados virá de computadores de fora, e assim não utilizaram processamento e memória da máquina em que o banco de dados está rodando.

Outra melhoria a ser feita no projeto é a separação do teste da camada de persistência e do banco de dados. Para isso seria necessário realizar os testes sobre a camada de persistência e o mesmo teste ser realizado diretamente no banco de dados. Os resultados desses dois testem tem como objetivo definir qual o impacto da camada de persistência sobre o banco de dados. É possível definir um teste ruim é devido a uma má gerência do banco de dados pela camada de persistência ou se o banco de dados não é capaz de tal operação.

Em sistemas reais o acesso a banco de dados não são feitos por usuários requisitando serviços em espaços de tempos fixos. Um avanço para um resultado mais preciso nos testes sobre a camada de persistência seria usuários iniciando operações no banco de dados em intervalos de tempo aleatórios, causando assim diversas situações distintas, por exemplo, momentos em que muitos usuários tentam realizar operações sobrecarregando o banco e momentos em que o banco não requisições, deixando-o ocioso. Esse comportamento é similar ao que ocorrem em sistemas reais, grande quantidade de acesso em determinado momento e ociosidade em outros. Testes seguindo esse padrão podem retornar resultados mais próximos da realizada do sistema testado.

Para complementar ainda mais o teste, uma nova aplicação poderia ser desenvolvida sobre um banco de dados NoSQL, ou seja, toda a construção baseada neste paradigma, desde as entidades (construídas de acordo com o paradigma) até as *queries* de manipulação do banco todas em sua linguagem nativa. Desta forma uma comparação entre o desempenho da camada de persistência dos dois paradigmas poderia ser feita considerando a execução otimizada para cada banco de dados. Esses dados podem complementar os dados já encontrados para analisar a viabilidade de migração de paradigmas de banco de dados ao utilizar-se de camadas de persistência.

# REFERÊNCIAS


ALEGROGRAPH. 2017. Https://allegrograph.com/. Acessado em: 06/12/2017. Disponível em: <https://allegrograph.com/>.

APACHE Cassandra. 2017. Http://cassandra.apache.org. Acessado em: 12/06/2017. Disponível em: <http://cassandra.apache.org>.

APACHE CouchDB. 2017. Http://couchdb.apache.org. Acessado em: 12/06/2017. Disponível em: <http://couchdb.apache.org>.

APACHE JMeter. 2017. Http://jmeter.apache.org/. Acessado em: 22/08/2017. Disponível em: <http://jmeter.apache.org/>.

APACHE JMeter. 2017. Http://jmeter.apache.org/. Acessado em: 01/11/2017. Disponível em: <http://jmeter.apache.org/api/org/apache/jmeter/protocol/java/sampler/JavaSamplerClient.html>.

APACHE JMeter Java Request. 2017. Http://jmeter.apache.org/usermanual/component_reference.html#Java_Request. Acessado em: 01/11/2017. Disponível em: <http://jmeter.apache.org/>.

ARANGODB. 2017. Https://www.arangodb.com/. Acessado em: 06/12/2017. Disponível em: <https://www.arangodb.com/>.

BAUER, C.; KING, G. **Hibernate in Action**. [S.l.]: Dreamtech Press, 2007. ISBN 9788177225594.

BRITO, R. W. Bancos de dados nosql x sgbds relacionais: Análise comparativa. **III Congresso Tecnológico TI e Telecom – InfoBrasil (26-28 de Maio, 2010, Fortaleza, CE)**, 2010. Disponível em: <http://www.infobrasil.inf.br/userfiles/27-05-S4-1-68840-Bancos%20de%20Dados%20NoSQL.pdf>.

CHANG, F. *et al.* Bigtable: A distributed storage system for structured data. **Symposium on Operation System Design and Implementation (November 6-8 2006, Seattle, United States)**, 2006.

CHEN, P. P.-S. The entity-relationship model - toward a unified view of data. **ACM Transactions on Database Systems**, v. 1, n. 1, p. 9–36, 1976. Disponível em: <http://www.lbd.dcc.ufmg.br/colecoes/wtdbd/2010/sbbd_wtd_12.pdf>.

COELHO, H. **JPA Eficaz: As melhores práticas de persistência de dados em Java**. 1. ed. [S.l.]: Casa do Código, 2013. ISBN 9788566250312.

DATE, C. **Introdução a Sistemas de Banco de Dados**. 8. ed. [S.l.]: Campus, 2004. ISBN 8535212736.

DATE, C. J. **Introdução a Sistemas de Banco de Dados**. 7. ed. [S.l.]: Campus, 2000.

DB-Engine. 2017. Http://db-engines.com/en/ranking. Acessado em: 05/11/2017. Disponível em: <http://db-engines.com/en/ranking>.

DEBNATH, M. **How to Manage Data Persistence with MongoDB and JPA**. 2016. Disponível em: <http://www.developer.com>.

DECANDIA, G. *et al.* Dynamo: Amazon's highly available key-value store. **21st ACM Symposium on Operating Systems Principles (October 14-17 2007, Stevenson, United States)**, 2007.

DIANA, M.; GEROSA, M. Nosql na web 2.0: Um estudo comparativo de bancos não-relacionais para armazenamento de dados na web 2.0. **X Workshop de Teses e Dissertações em Banco de Dados**, 2010. Disponível em: <http://www.lbd.dcc.ufmg.br/colecoes/wtdbd/2010/sbbd_wtd_12.pdf>.

HAUSER, C. A. **Projeto de banco de dados**. [S.l.]: Porto Alegre: Sagra Luzzato, 2000.

HECHT, R.; JABLONSKI, S. Nosql evaluation: A use case oriented survey. in cloud and service computing (csc). p. 336 – 341, 2011. Disponível em: <http://dsc.inf.furb.br/arquivos/tccs/monografias/TCC2012-2-22-VF-PauloABugmann.pdf>.

HIBERNATE OGM. 2017. Http://hibernate.org/ogm/. Acessado em: 11/11/2017. Disponível em: <www.hibernate.org>.

HIBERNATE OGM Documentation. 2017. Http://hibernate.org/ogm/. Acessado em: 11/11/2017. Disponível em: <http://hibernate.org/ogm/documentation/>.

HIBERNATE ORM. 2017. Http://hibernate.org/orm/. Acessado em: 11/11/2017. Disponível em: <www.hibernate.org>.

IBM MongoDB. 2017. Https://www.ibm.com. Acessado em: 13/06/2017. Disponível em: <https://www.ibm.com/developerworks/br/library/os-mongodb4/>.

LAKSHMAN, A.; MALIK, P. Cassandra - a decentralized structured storage system. **The 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware (October 10-11 2009, Big Sky, United States)**, 2009.

LEITE, G. S. **Análise Comparativa do Teorema CAP Entre Bancos de Dados NoSQL e Bancos de Dados Relacionais**. Dissertação (Trabalho de Conslusão de Curso) — FACULDADE FARIAS BRITO, 2010. Disponível em: <http://www.ffb.edu.br/sites/default/files/tcc-20102-gleidson-sobreira-leite.pdf>.

MARKET Research Media. 2017. Http://www.marketresearchmedia.com/?p=568. Acessado em: 15/11/2017. Disponível em: <http://www.marketresearchmedia.com/?p=568>.

MIILLER, M. G.; BONNETI, T. P. Mapeamento objeto relacional com hibernate em aplicaÇÕes java web. 2015. Disponível em: <http://web.unipar.br/~seinpar/2015/_include/artigos/Miguel_Gustavo_Miiller.pdf>.

MOHAMED, M. A.; ALTRAFI, O. G.; ISMAIL, M. O. Relational vs. nosql databases: A survey. **International Journal of Computer and Information Technology (May 2014)**, v. 3, 2014. ISSN 2279–0764. Disponível em: <https://www.researchgate.net/profile/Mohamed_Mohamed69/publication/263272704_Relational_vs._NoSQL_Databases_A_Survey/links/00b7d53a495312ad22000000.pdf>.

ORACLE. 2017. Http://docs.oracle.com/javaee/6/tutorial/doc/bnbtg.html. Acessado em: 27/08/2017. Disponível em: <http://docs.oracle.com/javaee/6/tutorial/doc/bnbtg.html>.

PANIZ, D. **NoSQL: Como armazenar os dados de uma aplicação moderna**. 1. ed. [S.l.]: Casa do Código, 2016. ISBN 9788555191930.

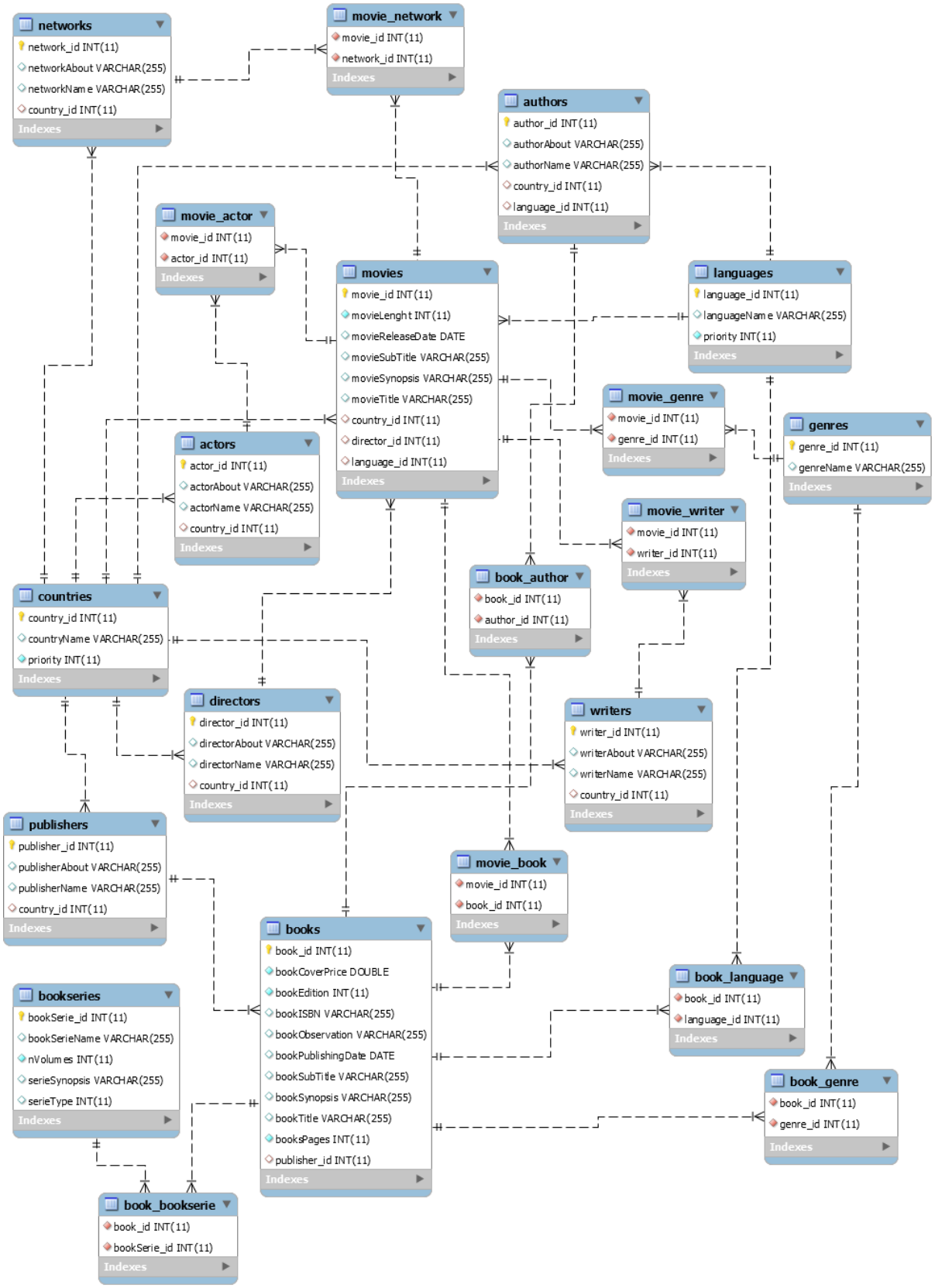
PERSISTENCE Oracle Documentation. 2017. Https://docs.oracle.com/html/E13946_06/ejb3_overview_persistence.html. Acessado em: 21/11/2017. Disponível em: <https://docs.oracle.com/en/>.

STöRL, U. *et al.* Schemaless nosql data stores – object-nosql mappers to the rescue? 2015. Disponível em: <http://btw-2015.de/res/proceedings/Hauptband/Ind/Stoerl-Schemaless_NoSQL_Data_Stores.pdf>.

TAKAI, O.; ITALIANO, I.; FERREIRA, J. Introdução a banco de dados. 2005. Disponível em: <http://dsc.inf.furb.br/arquivos/tccs/monografias/TCC2012-2-22-VF-PauloABugmann.pdf>.

# APÊNDICES

# A – DIAGRAMA ENTIDADE-RELACIONAMENTO DO BANCO DE DADOS RELACIONAL



# B – CONFIGURAÇÃO DO *PERSISTENCE.XML* DO PROJETO

A conexão entre a aplicação e o banco de dados com o JPA é feita em duas etapas, como mostrado na Figura 47. Na primeira etapa é instanciado o *entity manager factory* que da suporte as instâncias do *Entity Manager*. É papel do *factory* fazer a conexão com a unidade de persistência configurada para uso do banco de dados. Já a *entity manager* é quem da suporte a operações que exigem transações ativas. Essas transações são gerenciadas pelo *entity transaction* e são criadas a partir da *Entity Manager*.

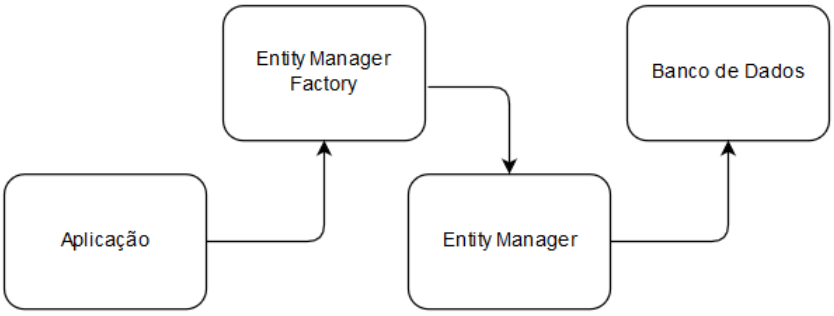



**Figura 47 – Conexão Banco de Dados com o JPA**

**Fonte: Autoria Própria.**

A primeira etapa corresponde à configuração da camada de persistência para acesso a base de dados. Nesta etapa é gerado um arquivo *persistence.xml*. Como este projeto visou a conexão em dois bancos de dados distintos, foram criadas duas unidades de persistência, cada uma corresponde a conexão em um banco de dados. Cada unidade é configurada separadamente dentro do arquivo xml e para isso são utilizadas *tags* que representam as informações necessárias para o acesso e manipulação dos dados nos bancos.

A primeira configuração é definir a unidade de persistência. O primeiro parâmetro desta *tag* é o nome que irá definir a unidade. É através deste nome que a *entity manager factory* faz a ligação entre aplicação e a camada de persistência. A Figura 48 mostra esta informação para os dois bancos de dados.

```
<persistence-unit name="mongo_PU" transaction-type="RESOURCE_LOCAL">

<persistence-unit name="mysql_PU" transaction-type="RESOURCE_LOCAL">
```

**Figura 48 – *Tag persistence-unit* da Unidade de Persistência**

**Fonte: Autoria Própria.**

O parâmetro *transaction-type* = *RESOURCE_LOCAL* indica que o programador é responsável pela criação da instância da *entity manager* e deve ser realizada explicitamente através da *entity manager factory* (PERSISTENCE..., 2017). Todas as informações pertinentes a conexão com o banco de dados está contido dentro desta *tag persistence-unit*.

Com a unidade de persistência criada, deve-se definir o provedor da unidade, ou seja, quem irá gerenciar a camada. Existem diversos *framaworks* para gerência de camada de persistência, neste trabalho foram escolhidos o Hibernate ORM para a gerência da camada de persistência do MySQL e o Hibernate OGM para a gerência da camada para o MongoDB. A definição deste provedor na unidade de persistência é feito através da *tag provider*, como mostra a Figura 49. A primeira linha refere-se ao provedor do MongoDB, pois utiliza o Hibernate OGM e a segunda ao provedor do MySQL.

```
<provider>org.hibernate.ogm.jpa.HibernateOgmPersistence</provider>

<provider>org.hibernate.ejb.HibernatePersistence</provider>
```

**Figura 49 – *Tag provider* da Unidade de Persistência**

**Fonte: Autoria Própria.**

Outra *tag* importante é a <*class*>. Esta *tag* explicita quais as classes com anotações de entidade serão de fato mapeados pela unidade de persistência, ou seja, todos as entidades devem estar declaradas na unidade de persistência para que ela seja representada como uma tabela no MySQL ou um documento no MongoDB. Como em ambos os bancos de dados as entidades mapeadas são as mesmas, esta configuração é igual nas duas unidades criadas, como mostra a Figura 50.

```
<class>entities.Actors</class>
<class>entities.Authors</class>
<class>entities.BookSeries</class>
<class>entities.Books</class>
<class>entities.Countries</class>
<class>entities.Directors</class>
<class>entities.Genres</class>
<class>entities.Languages</class>
<class>entities.Movies</class>
<class>entities.Networks</class>
<class>entities.Publishers</class>
<class>entities.Writers</class>
```

**Figura 50 – *Tag class* das Unidades de Persistência**

**Fonte: Autoria Própria.**

Por fim, são adicionadas as propriedades da conexão ao arquivo xml. Estas propriedades são específicas para cada banco de dados e contém as informações pertinentes ao banco, como o nome da base que será usada para armazenas os registros, o *driver* utilizado para conexão, o nome de usuário e senha do banco, etc (PERSISTENCE..., 2017). A Figura 51 apresenta as propriedades da configuração de ambos os bancos de dados.

A Figura 51a apresenta as configurações do MySQL e são descritas abaixo:

- ***javax.persistence.jdbc.url***

```
<properties>
    <property name="javax.persistence.jdbc.url"
              value="jdbc:mysql://localhost:3306/
              newlibrary_v4?zeroDateTimeBehavior=convertToNull"/>
    <property name="javax.persistence.jdbc.user" value="root"/>
    <property name="javax.persistence.jdbc.password" value="rachel"/>
    <property name="hibernate.ejb.entitymanager_factory_name" value="mysql_PU"/>
    <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
    <property name="hibernate.hbm2ddl.auto" value="update"/>
</properties>
```

**(a) Propriedades do MySQL**

```
<properties>
    <property name="hibernate.ogm.datastore.provider" value="mongodb"/>
    <property name="hibernate.ogm.datastore.database" value="newlibrary_mongo"/>
    <property name="hibernate.ogm.datastore.create_database" value="true"/>
    <property name="hibernate.ogm.datastore.host" value="localhost:27017"/>
    <property name="hibernate.ogm.datastore.username" value="" />
    <property name="hibernate.ogm.datastore.password" value="" />
</properties>
```

**(b) Propriedades do MongoDB**

**Figura 51 – *Tag proprieties* da Unidade de Persistência**

**Fonte: Autoria Própria.**

Define o endereço do banco de dados, neste caso, um banco local conectado na porta 3306 e qual o nome da base a se conectar.

- ***javax.persistence.jdbc.user***

  Define o usuário que ira acessar o banco.

- ***javax.persistence.jdbc.password***

  Define a senha de acesso ao banco.

- ***hibernate.ejb.entitymanager_factory_name***

  Define o nome da unidade de persistência que será utilizado pelo Hibernate.

- ***javax.persistence.jdbc.driver***

  Define o *driver* de conexão entre o Java e o banco de dados.

- ***hibernate.hbm2ddl.auto***

  Define a ação do Hibernate no banco de dados, neste caso sempre irá fazer uma atualização das tabelas quando executado.

As propriedades do MongoDB, mostradas na Figura 51b são mais simples, todas utilizam o *hibernate.ogm.datastore* e são descritas abaixo:

- ***provider***

  Determina o *driver* a ser utilizado pelo Java para a conexão.

- ***database***

  Define o nome da base de dados em que serão armazenados os registros.

- ***create_database***

  Esta propriedade indica para o Hibernate que ele deve criar uma base de dados no MongoDB com o nome *newlibrary_mongo* caso este banco ainda não exista.

- ***host***

  Define o endereço do banco de dados, neste caso, um banco local conectado na porta 27017.

- ***username***

  Define o usuário que ira acessar o banco, caso tenha um usuário especificado. O MongoDB é capaz de rodar sem ter nenhum usuário registrado.

- ***password***

  Define a senha de acesso ao banco, caso exista. Da mesma forma que o usuário o MongoDB pode não ter nenhuma senha configurada para acesso.

Este apêndice é referente apenas as configurações utiliziadas neste trabalho. Estas configurações são o mínimo necessário para uma conexão. Existem diversas formas de configurar o arquivo *persistence.xml* e é possível configurar inúmeros recursos dos bancos de dados através deste arquivo.

# C – CONFIGURAÇÃO DO JMETER

Para a criação de um teste no JMeter é necessário criar um plano de teste (*test plan*). O plano de teste contém os grupos de usuários onde são definidas as características dos testes. É no grupo de usuários que define-se o número de usuários que irão realizar a operação e também o tempo para inicialização de cada *Thread* (usuário). A Figura 52 mostra a interface de um plano de testes com um grupo de usuários no JMeter.

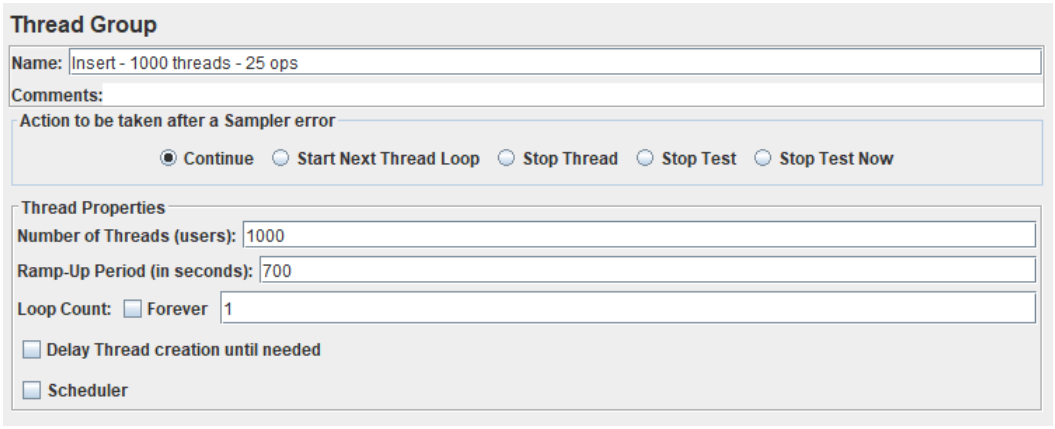



**Figura 52 – Configuração do Grupo de Usuários do JMeter**

**Fonte: Autoria Própria.**

O tempo para início de um usuário é definido pela rampa de início (*ramp-up period*). Essa rampa define o tempo máximo para que todos os usuários sejam iniciados, ou seja, em um grupo com 1000 usuários e uma rampa de 700 segundos, conforme o exemplo mostrado na Figura 52, um usuário será iniciado a 700 milissegundos.

Após definido o número de usuários em cada grupo foi necessário definir o que cada usuário deveria fazer. Neste trabalho cada usuário realizou um *Java request*, ou seja, acessa uma interface criada em Java e executa o código contido nesta interface. Em cada grupo de usuário foi adicionado um *sampler* do tipo *Java request*. Neste *sampler* é selecionada a interface que os usuários irão acessar para realizar os testes. A Figura 53 apresenta a seleção da interface a ser acessada.

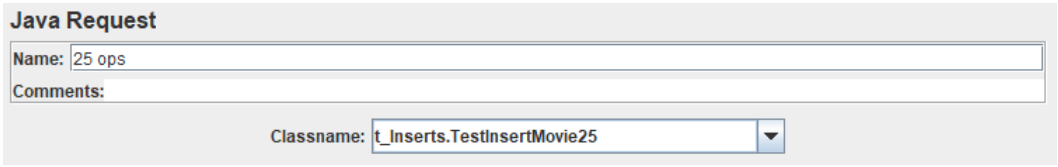



**Figura 53 – *Sampler Java request* do JMeter**

**Fonte: Autoria Própria.**

O campo *classname* visto na Figura 53 é o local que se define a interface a ser acessada, neste caso o JMeter realiza o acesso a classe Java que faz a inserção de 25 registros no banco.

A Figura 54 mostra a configuração básica feita para os testes. Todos os testes seguem o mesmo padrão com um grupo de usuários para cada teste, de acordo com a quantidade de operações. Em cada teste é modificado o número de usuários, o tempo de início e a requisição Java, conforme mostrado nas figuras 52 e 53.

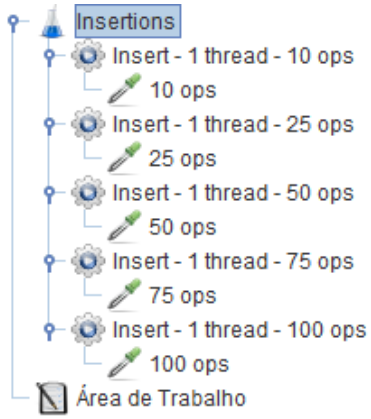



**Figura 54 – Configuração do Grupo de Usuários do JMeter**

**Fonte: Autoria Própria.**