



CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS  
CURSO DE ENGENHARIA DE COMPUTAÇÃO

## **PERSISTÊNCIA POLIGLOTA**

**JOSÉ FRANCISCO CAMPOS LIMONGI**

Orientador: Evandrino Barros  
Centro Federal de Educação Tecnológica de Minas Gerais – CEFET-MG

BELO HORIZONTE  
AGOSTO DE 2014

**JOSÉ FRANCISCO CAMPOS LIMONGI**

## **PERSISTÊNCIA POLIGLOTA**

Modelo canônico de trabalho monográfico acadêmico  
em conformidade com as normas ABNT apresentado à  
comunidade de usuários L<sup>A</sup>T<sub>E</sub>X.

Orientador:     Evandrino Barros  
                    Centro Federal de Educação Tecnológica  
                    de Minas Gerais – CEFET-MG

CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS  
CURSO DE ENGENHARIA DE COMPUTAÇÃO  
BELO HORIZONTE  
AGOSTO DE 2014

**JOSÉ FRANCISCO CAMPOS LIMONGI**

## **PERSISTÊNCIA POLIGLOTA**

Modelo canônico de trabalho monográfico acadêmico  
em conformidade com as normas ABNT apresentado à  
comunidade de usuários  $\text{\LaTeX}$ .

Trabalho aprovado. Belo Horizonte, 24 de novembro de 2014

---

**Evandrino Barros**  
Orientador

---

**Professor**  
Convidado 1

---

**Professor**  
Convidado 2

CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS  
CURSO DE ENGENHARIA DE COMPUTAÇÃO  
BELO HORIZONTE  
AGOSTO DE 2014

*“Seja realista, exija o impossível.” (Roland Castro)*

## Lista de Figuras

Figura 1 – Exemplo de um diagrama Unified Model Language (UML) para o modelo relacional . . . . .	7
Figura 2 – Exemplo da disposição dos dados no modelo relacional . . . . .	7
Figura 3 – Exemplo de um diagrama UML utilizando agregação . . . . .	9
Figura 4 – Exemplo da disposição dos dados utilizando agregação . . . . .	10
Figura 5 – Exemplo de um diagrama UML para utilizando agregação entre cliente e pedido . . . . .	11
Figura 6 – Exemplo da disposição dos dados no NoSQL, utilizando agregação entre cliente e pedido . . . . .	11
Figura 7 – Tela Inicial . . . . .	16
Figura 8 – Tela de cadastro de usuário . . . . .	17
Figura 9 – Tela Tweets do Usuário Logado . . . . .	17
Figura 10 – Tela de Login . . . . .	18
Figura 11 – Tela Cadastro de Tweet . . . . .	18
Figura 12 – Tela de Listagem de Usuários . . . . .	19
Figura 13 – Tela de Feed . . . . .	21
Figura 14 – Camada de modelo . . . . .	22
Figura 15 – Camada de controle . . . . .	23

## **Lista de Tabelas**

## Lista de Siglas

<b>SQL</b> Structured Query Language.....	1
<b>Redis</b> Remote Dictionary Server.....	1
<b>SGBD</b> Sistema de Gerenciamento do Banco de Dados .....	5
<b>UML</b> Unified Model Language .....	iv
<b>UoD</b> universo de discurso.....	4
<b>ACID</b> Atomicidade, Consistência, Isolamento e Durabilidade .....	12
<b>XML</b> <i>eXtensible Markup Language</i> .....	13
<b>MVC</b> <i>Model View Controller</i> .....	21

# Sumário

<b>1 – Introdução</b>	<b>1</b>
1.1 Motivação	2
<b>2 – Fundamentação Teórica</b>	<b>4</b>
2.1 Banco de Dados	4
2.2 Gêneros de Banco de dados	6
2.2.1 Banco de dados Relacional	6
2.2.2 Banco de dados não-relacional	8
2.2.3 Persistência Poliglota	12
<b>3 – Implementação</b>	<b>15</b>
3.1 Casos de Uso	15
3.1.1 Caso de Uso 1 - Cadastro do usuário	15
3.1.2 Caso de Uso 2 - Cadastro de <i>tweet</i>	17
3.1.3 Caso de Uso 3 - Usuário segue outro usuário	18
3.1.4 Caso de Uso 4 - Usuário para de seguir algum usuário	19
3.1.5 Caso de Uso 5 - Usuário visualiza a <i>feed</i>	20
3.2 Sistema Desenvolvido	21
3.2.1 Implementação com Persistência Monoglota	21
3.2.2 Implementação com Persistência Poliglota	25
<b>4 – Resultados</b>	<b>27</b>
4.1 Resultados do tempo de consulta da <i>feed</i> de <i>tweets</i>	27
4.2 Resultados do tempo de inserção de um <i>tweet</i>	28
4.3 Análise dos resultados	28
<b>5 – Conclusão</b>	<b>30</b>
<b>Referências</b>	<b>31</b>



# 1 Introdução

A necessidade de persistência de dados sempre esteve presente na computação. A medida que os sistemas evoluíram a complexidade da forma que os dados eram armazenados aumentou significativamente. Com isso, houve a necessidade da criação de um sistema computadorizado de manutenção de registros (DATE, 2004), o banco de dados.

O modelo relacional foi um dos primeiros modelos que surgiram, sua estrutura são tabelas de duas dimensões com linhas e colunas. Os dados armazenados são tipados, podendo variar a quantidade de tipos de acordo com o banco utilizado. Para interagir com esse modelo é necessário realizar consultas com a linguagem Structured Query Language (SQL). Alguns exemplos de banco de dados relacional são MySQL<sup>1</sup>, Oracle<sup>2</sup> e PostgreSQL<sup>3</sup>.

Durante anos, o banco de dados relacional tem sido considerado a melhor opção para os problemas de escalabilidade, porém surgiram novas soluções com novas alternativas de estruturas de dados, replicação simples, alta disponibilidade e novos métodos de consultas (REDMOND; WILSON, 2012). Essas opções são conhecidas como NoSQL ou banco de dados do modelo não-relacional

Existem diversos gêneros de banco de dados do modelo não-relacional, entre eles chave-valor, orientado a documento, orientado a coluna e orientado a nó. Com o surgimento dessas novas soluções, o questionamento sobre qual banco de dados é melhor para resolver certo tipo de problema, vem à tona. A partir disso, o conhecimento e compreensão sobre os bancos de dados em geral se torna necessário para realizar uma boa escolha.

Entendendo que cada banco se destaca em determinados tipos de problema, é nítido perceber que sistemas que trabalham com mais de um banco de dados podem oferecer um melhor desempenho, dando origem à persistência poliglota.

Este trabalho consiste na comparação de dois sistemas baseados em NoSQL. O primeiro utilizará apenas um banco de dados, que será do gênero orientado a documento. O segundo utilizará dois bancos de dados, sendo um do gênero orientado a documento e um outro banco do gênero chave-valor. O segundo sistema, por utilizar dois bancos de dados, caracteriza a persistência poliglota, que é alvo desse trabalho. Os bancos escolhidos para fazer esses sistemas foram o MongoDB e o Remote Dictionary Server (Redis). O autor escolheu esses bancos de dados por ter experiência na linguagem

---

<sup>1</sup> Sítio oficial <<http://www.mysql.com>>

<sup>2</sup> Sítio oficial <<http://www.oracle.com/technetwork/oem/db-mgmt/db-mgmt-093445.html>>

<sup>3</sup> Sítio oficial <<http://www.postgresql.org/>>

*Ruby on Rails*, que oferece excelentes bibliotecas para esses bancos e por serem utilizados em diversas aplicações. O MongoDB está sendo utilizado em grandes empresas, como Cisco, eBay, Codecademy, Microsoft, Craigslist, The Guardian e outras, conforme sítio oficial do MongoDB <sup>4</sup>. O Redis está sendo utilizado no Twitter, Github, Craigslist e outros conforme referência do sítio oficial <sup>5</sup>. O intuito desse trabalho é comparar desempenho do uso da persistência poliglota com a persistência monoglota.

MongoDB é um banco de dados do gênero orientado a documento e foi desenhado para ser gigante. O próprio nome é uma derivação da palavra inglesa *humongous* que significa gigantesco. O diferencial desse banco é a maneira que os registros são armazenados. Cada registro fica armazenado em um documento que é análogo à tupla do modelo relacional. O documento é composto por um identificador único e um conjunto de valores de tipos e estruturas aninhadas. Esse banco é bem flexível, pois não tem esquema pré-definido e permite valores multivalorados. Ao criar a arquitetura do sistema temos que identificar se as entidades criadas são expressivas como um documento (REDMOND; WILSON, 2012). Além disso, há soluções para tratar concorrência e foi desenhado para trabalhar em *clusters*. A linguagem utilizada para fazer consulta no MongoDB é JavaScript. Nesse trabalho utilizaremos o banco MongoDB nos dois sistemas que serão criados.

O segundo banco utilizado se chama Redis do gênero chave-valor. Esse tipo de armazenamento é mais simples. Como o próprio nome indica, é armazenado um valor para determinada chave. O valor armazenado pode ter uma estrutura variável. Escolhemos esse banco, pois antes de efetivar as operações em disco, ele mantém os valores na memória primária. Dessa forma, o banco tem um ganho muito alto em desempenho, porém poderá ocorrer perda de dados, caso ocorra uma falha de hardware (REDMOND; WILSON, 2012). A forma de como estruturar esse banco é muito parecida com um tipo estruturado chamado *hash* que são implementadas em algumas linguagens de computação, como Java e Ruby. Esse banco será utilizado no segundo sistema a ser desenvolvido.

## 1.1 Motivação

A persistência poliglota é uma alternativa para melhorar o desempenho de uma aplicação. Utilizando-a conseguimos adaptar cada tipo de problema a um gênero de banco de dados.

Existem duas variáveis opostas no ambiente de persistência de dados, consistência e disponibilidade. Quanto mais consistente um dado, menos disponível ele será e quanto

<sup>4</sup> <<http://www.mongodb.com/customers>>

<sup>5</sup> <<http://redis.io/topics/whos-using-redis>>

mais disponível um dado menos consistente ele estará ([SADALAGE; FOWLER, 2013](#)). Em aplicações é comum termos um conjunto de dados que deve ser sempre consistente e um outro conjunto de dados que deve estar sempre disponível. Logo, para atender a esses conjuntos de dados podem ser utilizados dois bancos de dados, um que garante disponibilidade e outro que garante consistência. Utilizando esse ambiente misto, é esperado que haja um ganho de desempenho.

Atualmente poucos trabalhos apresentam uma comparação entre sistemas que utilizam apenas um banco, persistência monoglota, e sistemas que utilizam persistência poliglota.

Com esse trabalho pretendemos ilustrar as diferenças entre a persistência monoglota e poliglota, destacando as vantagens que a persistência poliglota pode oferecer. Fizemos a implementação dos dois sistemas e coletamos resultados. Esses resultados indicaram que a persistência poliglota aumentou significativamente o desempenho em um ponto, mas devido ao modelo utilizado, piorou o sistema em outro ponto.

O trabalho está organizado em cinco capítulos, o próximo, [Capítulo 2](#), apresenta a fundamentação teórica necessária para fazer esse trabalho. O [Capítulo 3](#) explica os sistemas criados e a diferença de implementação entre eles. O [Capítulo 4](#) apresenta os resultados preliminares que demonstraram um melhor desempenho da persistência poliglota e, por fim, [Capítulo 5](#) apresenta a conclusão do trabalho.

## 2 Fundamentação Teórica

Para entendermos o porquê de utilizar mais de um banco de dados em uma mesma aplicação, temos que entender o que é um banco de dados, quais gêneros existem e para qual tipo de problema cada gênero se destaca.

Este capítulo está organizado em duas seções, a [Seção 2.1](#) apresenta o conceito de banco de dados e a [Seção 2.2](#) apresenta os gêneros de bancos de dados mais utilizados atualmente. A [Seção 2.2](#) se divide em três subseções, a [Subseção 2.2.1](#) apresenta o conceito do modelo relacional, a [Subseção 2.2.2](#) apresenta o conceito do modelo não-relacional comparada com o conceito do modelo relacional e a [Subseção 2.2.3](#) apresenta o conceito de persistência poliglota.

### 2.1 Banco de Dados

Banco de dados é um sistema computadorizado de manutenção de registros, análogo à um armário de arquivamento eletrônico. Podemos entendê-lo como um repositório para manter a coleção de arquivos de dados computadorizados ([DATE, 2004](#)). [Elmasri \(2005\)](#) define banco de dados como uma coleção de dados relacionados e que dados são fatos com um significado implícito. Porém, a definição de [Elmasri \(2005\)](#) é mais abrangente, logo ele aponta três propriedades implícitas para restringir a definição de banco de dados.

A primeira propriedade é que um banco de dados deve representar alguns aspectos do mundo real, chamado de *universo de discurso* (*UoD*). As alterações que ocorrem nesse universo são refletidas em um banco de dados. A segunda propriedade define que o banco de dados é uma coleção lógica e coerente de dados com algum significado inerente, ou seja, uma coleção de dados randômicos não pode ser considerado um banco de dados. A terceira propriedade afirma que banco de dados é projetado, construído e povoado com dados, atendendo a uma proposta específica. Além disso, possui um grupo de usuários definido e algumas aplicações preconcebidas, de acordo com o interesse desse grupo.

Os bancos de dados têm contribuído para o aumento do uso do computador ([ELMASRI, 2005](#)) e podemos afirmar que eles apresentam um papel crucial em quase todas as áreas em que os computadores são utilizados. Devido a essa importância o estudo sobre banco de dados é extremamente necessário para os profissionais da computação.

Antes da existência dos bancos de dados, a aplicação devia gerenciar e pro-

cessar arquivos para manter os dados persistidos. Para justificar o uso de banco de dados, [Elmasri \(2005\)](#) cita quatro características: natureza autodescritiva, abstração de dados, suporte para as múltiplas visões de dados e processamento de transações de multiusuários.

A primeira característica, natureza autodescritiva do banco de dados, apresenta o catálogo do banco de dados como uma grande vantagem sobre o processamento tradicional dos arquivos, pois o catálogo identifica as estruturas dos arquivos, formato e tipo de dados. Logo, não é necessário conhecer a aplicação para trabalhar com os dados. Já o processamento tradicional dos arquivos, mantém essas definições de estrutura na própria aplicação ([ELMASRI, 2005](#)).

Em relação à característica abstração de dados, [Elmasri \(2005\)](#) afirma que não é feita no processamento tradicional de arquivos, pois é a aplicação que define a estrutura dos dados. Por exemplo, suponha que tenhamos diversos programas utilizando o mesmo arquivo para armazenar uma coleção de dados. Se um desses programas precisar de acrescentar algum campo novo, todos os outros programas que acessam esse arquivo, devem ser modificados para contemplar o novo campo adicionado. Já quando utilizamos banco de dados, a alteração da estrutura dos dados pode não influenciar no funcionamento dos outros programas.

Em relação à característica suporte para múltiplas visões dos dados, [Elmasri \(2005\)](#) diz que quando é utilizado o banco de dados, é possível ter diferentes visões sobre os dados, fazendo o cruzamento das tabelas. Com a abordagem de processamento de arquivo tradicional, isso não é usual.

A última característica citada por [Elmasri \(2005\)](#) é o processamento de transações multiusuários, essa característica é essencial para que várias aplicações possam acessar e alterar os dados a partir de usuários diferentes e simultâneos. Porém, o Sistema de Gerenciamento do Banco de Dados ([SGBD](#)) deve ter implementado um controle de concorrência para garantir a atomicidade das transações e a consistência dos mesmos.

[Elmasri \(2005\)](#) não cita a existência dos bancos de dados NoSQL, que são bancos que não têm esquema pré-definido, chamados de *schemaless*. Apesar de não ter a declaração do tipo de estruturas de dados contidas no banco, os bancos de dados *schemaless* fazem a abstração dos dados da mesma maneira que os bancos de dados tradicionais, têm suporte para múltiplas visões e multiusuários.

Como revelado acima, a utilização do banco de dados facilita o desenvolvimento das aplicações, permite a abstração entre aplicação e dados e, além disso, faz o controle de concorrência. Após verificarmos que o uso de banco de dados é imprescindível, nos deparamos com uma outra dificuldade, qual banco de dados utilizar. Os bancos de dados, chamados de NoSQL, chamam a atenção da comunidade científica, depois da

publicação de dois artigos BigTable ([CHANG et al., 2006](#)) e Dynamo ([DECANDIA et al., 2007](#)). Essas publicações apresentam bancos de dados NoSQL que têm um desempenho superior ao modelo relacional. O BigTable apresenta um banco de dados orientado a coluna e o Dynamo apresenta um banco de dados chave-valor.

## 2.2 Gêneros de Banco de dados

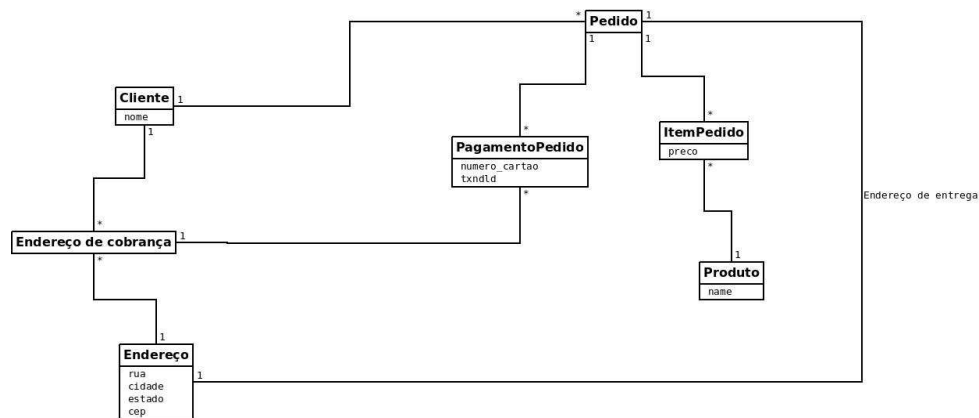
Durante anos, o banco de dados relacional tem sido considerado a melhor opção para a maioria dos problemas de pequeno ou grande volume de dados. No entanto, o aumento do volume de dados fez com que os especialistas buscassem novas soluções, que permitissem o armazenamento paralelo dos dados, pois o modelo relacional não foi desenhado para funcionar em *clusters*. Logo, os bancos de dados NoSQL se destacaram por funcionarem bem nesse ambiente e terem um melhor desempenho com grande volume de dados ([SADALAGE; FOWLER, 2013](#)).

Os bancos de dados NoSQL têm as seguintes características: não usam o modelo relacional, foram desenhados para funcionar em *clusters*, são *open source* e não tem catálogos (*schemaless*) ([SADALAGE; FOWLER, 2013](#)). Carlo Strozzi foi o primeiro a utilizar o nome NoSQL, mas não no sentido que a palavra tem hoje. Strozzi denominou um banco de dados relacional, *open source* de NoSQL, pois não usava **SQL** como linguagem de consulta. O sentido, que a palavra tem hoje, veio de uma conferência realizada em São Francisco nos Estados Unidos, em Junho de 2009. Johan Oskarsson que organizou essa conferência, escolheu esse nome, porque era uma boa *hashtag* no Twitter, pois era pequeno, memorável e tinha poucos resultados no Google. Isso facilitaria os interessados a encontrar a conferência. Apesar do termo não significar explicitamente o que são esses bancos de dados, atendeu bem a intenção de Oskarsson ([SADALAGE; FOWLER, 2013](#)).

### 2.2.1 Banco de dados Relacional

O modelo relacional armazena os dados em tabelas de duas dimensões em linhas e colunas. A interação com esse banco é feito por um **SGBD** que utiliza a **SQL** como linguagem de consulta de dados. Os dados armazenados são valores tipados e podem ser numéricos, texto, data e outros tipos, que são configurados e forçados pelo sistema. É possível fazer junção das tabelas e obter diferentes perspectivas das informações de maneira simples. MySQL, Oracle e PostgreSQL são alguns exemplos de banco de dados relacional ([REDMOND; WILSON, 2012](#)). A representação do modelo relacional pode ser feita com o diagrama **UML**. Por exemplo, um sistema de *e-commerce* poderia ser desenhado conforme o diagrama da [Figura 1](#) e as tuplas ficariam dispostas conforme [Figura 2](#).

Figura 1 – Exemplo de um diagrama UML para o modelo relacional



Fonte: (SADALAGE; FOWLER, 2013)

Figura 2 – Exemplo da disposição dos dados no modelo relacional

Customer		Order		
Id	Name	Id	CustomerId	ShippingAddressId
1	Martin	99	1	77

Product		BillingAddress		
Id	Name	Id	CustomerId	AddressID
27	NoSQL Distilled	55	1	77

OrderItem				Address	
Id	OrderId	ProductId	Price	Id	City
100	99	27	37.45	77	Chicago

OrderPayment				
Id	OrderId	CardNumber	BillingAddressId	txnid
33	99	1000-1000	55	abelif879rft

Fonte: (SADALAGE; FOWLER, 2013)

O modelo relacional funciona muito bem para diversas aplicações, pois é bem flexível em relação às consultas, permite concorrência, transações e pode ser integrado com várias aplicações. Porém, há uma desvantagem que causa frustração em muitos desenvolvedores, chamada de Impedância de Correspondência ou *Impedance Mismatch* (ELMASRI, 2005; SADALAGE; FOWLER, 2013). Isso ocorre, pois nem sempre o tipo



do campo no banco de dados irá corresponder com o tipo esperado da linguagem utilizada, então é necessário criar uma forma de associação entre o tipo da variável da linguagem com o tipo do valor da tabela. Outra desvantagem é que esse não aceita valores multivalorados, distanciando a aplicação ainda mais do modelo relacional.

### 2.2.2 Banco de dados não-relacional

Os bancos de dados NoSQL foram construídos para suprir a necessidade de se trabalhar com grande quantidade de dados e em *clusters*. NoSQL abrange diversos gêneros de banco de dados, entre eles o orientado a documento, chave-valor, orientado a coluna e orientado a nó.

Todos esses gêneros não possuem catálogo, ou seja, não se define previamente em qual estrutura os dados serão armazenados. Isso permite uma flexibilidade no sistema, pois a estrutura dos dados pode ser alterada facilmente. É possível adicionar um novo campo, sem ter que se preocupar com a atualização da base legada, pois os objetos de uma mesma coleção podem ter diferentes campos. Da mesma maneira, para remover um campo, basta parar de armazená-lo. Os registros antigos, que tinham esse campo, continuarão com eles e os objetos novos não irão armazenar esse campo, que já não faz parte da aplicação (SADALAGE; FOWLER, 2013).

Com isso é possível trabalhar com dados não uniformes, isto é, dados que para cada registro há um conjunto diferente de atributos. Para que o banco de dados relacional lide com um objeto dessa natureza, é necessário uma tabela com os campos de todos os objetos e conseqüentemente, isso traria uma quantidade grande de campos vazios.

Basicamente, os bancos de dados NoSQL deslocam a definição do esquema para a aplicação que acessa o banco. Isso pode se tornar problemático quando há muitas aplicações acessando o mesmo banco, mas existem soluções para resolver isso. Uma delas seria encapsular toda a interação com o banco de dados, fazendo esse funcionar como um *web service* (SADALAGE; FOWLER, 2013).

Além dessa flexibilidade, a maioria dos gêneros NoSQL permite estruturar dados multivalorados<sup>1</sup>. Esses dados são modelados como agregação, que é uma coleção de objetos relacionados que desejamos tratar como um único objeto (EVANS ERIC/FOWLER, 2003). A agregação facilita a manipulação e a consistência desses dados, pois é tratado como uma unidade, ou seja, é lido e escrito sempre todo o conjunto de dados. Como, geralmente, ao utilizar esse relacionamento buscamos, operações atômicas, essa abordagem se adapta perfeitamente com a aplicação (SADALAGE; FOWLER, 2013). Para exemplificar, iremos modelar o exemplo de *e-commerce* anterior de duas maneiras

---

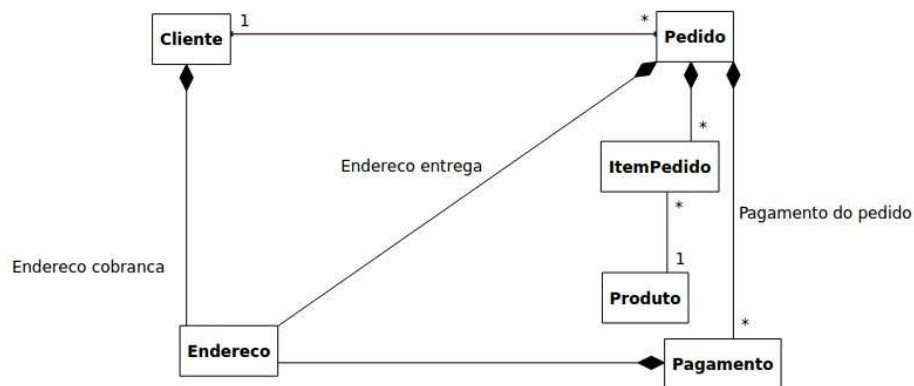
<sup>1</sup> O banco de dados orientado a nó, é um dos gêneros que não implementam agregação.



diferentes.

A primeira iremos utilizar agregação entre as relações de *Pedido-Endereço*, *Pedido-ItemPedido*, *Pedido-Pagamento*, *Cliente-Endereço*, e por fim *Endereço-Pagamento*. No diagrama iremos utilizar o símbolo de composição em UML para demonstrar como a informação se adapta na estrutura de agregação (SADALAGE; FOWLER, 2013). O diagrama ficaria como a Figura 3 e a disposição de dados ficaria como a Figura 4. É usual a utilização de JSON<sup>2</sup> para mostrar os dados em NoSQL (SADALAGE; FOWLER, 2013).

Figura 3 – Exemplo de um diagrama UML utilizando agregação



Fonte: (SADALAGE; FOWLER, 2013)

<sup>2</sup> JSON é um tipo de estrutura de dados muito utilizado em JavaScript que representa os dados estruturados de uma maneira simples para as pessoas lerem e escreverem e para as máquinas fazerem a conversão.

Figura 4 – Exemplo da disposição dos dados utilizando agregação

```
// Em clientes
{
  "id": 1,
  "nome": "Martin",
  "endereco_cobranca": [{"cidade": "Chicago"}],
}

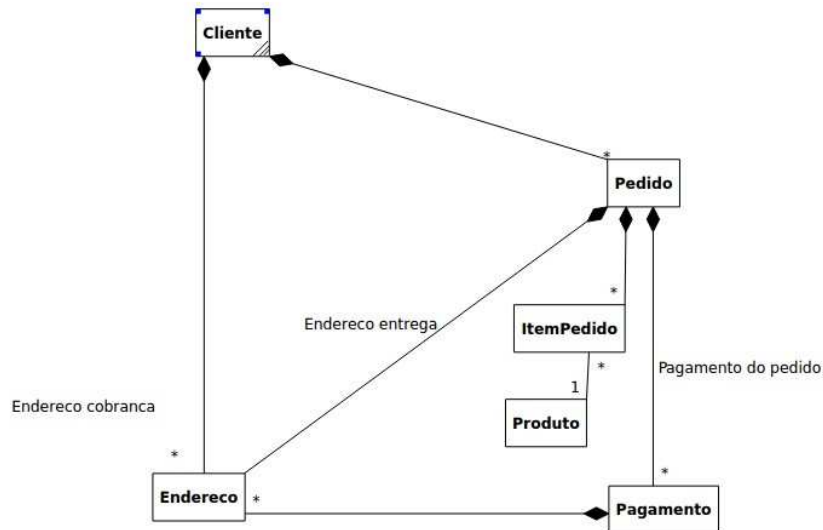
//Em pedidos
{
  "id": 99,
  "cliente_id": 1,
  "itens_pedido":[
    {
      "produto_id": 27,
      "preco": 32.45,
      "nome_produto": "NoSQL Distilled"
    }
  ],
  "endereco_entrega": [{"cidade": "Chicago"}],
  "pedido_pagamento": [
    {
      "ccinfo": "1000-1000-1000-1000",
      "txid": "abellf879rft"
      "endereco_cobranca" {"cidade": "Chicago"}
    }
  ]
}
```

Fonte: (SADALAGE; FOWLER, 2013)

Observamos que as duas principais agregações são cliente e pedido. O cliente contém apenas uma lista, que é a de endereços. Já o pedido contém uma lista de itens de pedido, endereços e pagamentos. E por fim, o pagamento contém uma lista de endereços. Endereço aparece três vezes, mas ao invés de utilizar uma referência com um identificador, como no modelo relacional, o valor é copiado. Essa replicação de dados se adapta ao problema, pois, nesse caso, não queremos que o endereço em pagamento seja alterado caso o usuário altere o endereço de entrega. Utilizando o modelo relacional temos duas maneiras de lidar com o problema. A primeira é tratar para que não seja permitido alterar o endereço, após desse ser vinculado à pedido ou pagamento. A segunda e mais usada, seria duplicar o endereço e associar à pedido ou à pagamento, ou seja, os dados são replicados da mesma maneira. As relações entre *Pedido-Cliente* e *ItemPedido-Produto* funcionam da mesma maneira que o modelo relacional, um lado da relação armazena o identificador do outro lado, por exemplo, o pedido mantém o identificador do cliente relacionado.

A segunda maneira de modelar o exemplo de *e-commerce* em NoSQL, seria utilizar as agregações do exemplo anterior (*Pedido-Endereço*, *Pedido-ItemPedido*, *Pedido-Pagamento*, *Cliente-Endereço*, e por fim *Endereço-Pagamento*) mais a agregação entre cliente e pedido. O modelo em UML ficaria como a Figura 5 e os dados ficariam dispostos como a Figura 6.

Figura 5 – Exemplo de um diagrama UML para utilizando agregação entre cliente e pedido



Fonte: (SADALAGE; FOWLER, 2013)

Figura 6 – Exemplo da disposição dos dados no NoSQL, utilizando agregação entre cliente e pedido

```

// Em clientes
{
  "id": 1,
  "nome": "Martin",
  "endereco_cobranca": [{"cidade": "Chicago"}],
  "pedidos": [{
    "id": 99,
    "cliente_id": 1,
    "itens_pedido": [
      {
        "produto_id": 27,
        "preco": 32.45,
        "nome_produto": "NoSQL Distilled"
      }
    ],
    "endereco_entrega": [{"cidade": "Chicago"}],
    "pedido_pagamento": [
      {
        "ccinfo": "1000-1000-1000-1000",
        "txdId": "abelif879rft",
        "endereco_cobranca": {"cidade": "Chicago"}
      }
    ]
  }
]
}

```

Fonte: (SADALAGE; FOWLER, 2013)

Como na maioria dos problemas de modelagem não existe a melhor solução, mas sim uma que se adapta melhor para um certo tipo de problema, devemos analisar o objetivo da aplicação para fazer uma melhor escolha.

Nesse exemplo se for interessante para a aplicação listar o histórico dos pedidos do mês corrente, o modelo da [Figura 5](#) não é o ideal, pois será necessário entrar em cada cliente para ler os pedidos. Já utilizando o modelo da [Figura 3](#), essa consulta do histórico se torna trivial ([SADALAGE; FOWLER, 2013](#)), basta fazer uma busca dos pedidos que contém a data igual a do mês corrente.

Por outro lado, ao buscarmos os pedidos de um cliente, o modelo da [Figura 5](#) irá apresentar um melhor desempenho. Pois, será necessário apenas buscar o cliente que essa consulta irá trazer todos os pedidos. Já o modelo da [Figura 3](#) será necessário percorrer pedido por pedido e comparar o identificador do cliente.

Outra razão para utilizar agregação é a facilidade proporcionada para colocar o funcionamento em *clusters*. Pois, utilizando esse modelo temos a informação de quais dados devem ser manipulados juntos e se devem estar no mesmo nó do *cluster*.

O modelo relacional por não tratar o conceito de agregação é chamado de *aggregate-ignorant*. Apesar disso, o modelo relacional tem suas vantagens por não tratar. A principal é que ao utilizar o modelo relacional, podemos analisar os dados em diversas perspectivas, já no banco de dados NoSQL algumas consultas podem não ser complexas ou até mesmo mais lentas. Por exemplo, o modelo da [Figura 5](#) ficará mais lento para buscar o histórico de pedidos.

Outra desvantagem do NoSQL, é não implementar transações do tipo Atomicidade, Consistência, Isolamento e Durabilidade ([ACID](#)). Esses gêneros de banco de dados conseguem implementar essas transações apenas sobre uma agregação como dito anteriormente. Caso seja necessário o controle dessas transações, deverá ser feito no código da aplicação.

### 2.2.3 Persistência Poliglota

O objeto desse trabalho são duas aplicações que utilizam dois gêneros de banco de dados, orientado a documento e chave-valor.

O gênero chave-valor funciona como uma simples tabela *hash* que dado uma chave única encontrará um valor. O sistema desse banco não tem conhecimento algum sobre esse valor, logo poderá ser um texto, um objeto multivalorado, um valor binário. Qualquer tipo de informação, apenas o tamanho é limitado, dependendo do banco. Alguns exemplos de bancos desse gênero são [Redis](#)<sup>3</sup> e [Riak](#)<sup>4</sup>. A consistência utilizando

<sup>3</sup> Sítio oficial do Redis <<http://redis.io/>>

<sup>4</sup> Sítio oficial do Riak <<http://basho.com/riak/>>

esse gênero existe apenas para operação realizada com uma chave. Então não temos consistência para um grupo de chaves. Isso pode ser implementado, mas é muito custoso (SADALAGE; FOWLER, 2013). Em relação à consistência no ambiente paralelo, cada banco implementa de uma maneira. O Riak, por exemplo, utiliza o chamado *Quorums*, que é uma abordagem de consistência que elege o valor mais atual fazendo uma consulta entre os nós. Aquele valor que tiver na maioria dos nós será eleito como mais atualizado.

Como o banco não tem conhecimento da estrutura armazenada, qualquer filtro necessário deverá ser feito pela aplicação. A aplicação faz a busca no banco utilizando a chave, o banco retorna o valor referente a essa chave e em seguida, a aplicação trabalha com esse valor fazendo o filtro dos registros que deseja. Também é possível programar o tempo de expiração da chave. Essa maneira que o banco chave-valor trabalha faz com que ele se adeque bem para manter dados da sessão de um usuário, carrinhos de compra de um *e-commerce*, perfil de usuários e outros. Não é indicado para armazenar estruturas que a aplicação fará buscas comparando os dados do valor. Por exemplo, suponha que tenhamos o modelo da Figura 5 para armazenar a estrutura utilizando o modelo chave-valor. Para realizar a busca de todos os pedidos do mês corrente, teremos que carregar para a aplicação todos os clientes, acessar os pedidos de cada um e verificar se a data do pedido é igual ao do mês corrente (SADALAGE; FOWLER, 2013).

O gênero orientado a documento trabalha com o conceito de documento. Documento pode ser um *eXtensible Markup Language* (XML), JSON, BSON ou outros formatos. Esses documentos são armazenados em coleções. Documentos da mesma coleção são semelhantes, mas não precisam ser idênticos. Se fizermos uma analogia, a coleção seria a tabela no modelo relacional e um documento seria uma tupla no modelo relacional. Exemplos desse gênero são MongoDB<sup>5</sup> e CouchDB<sup>6</sup> (SADALAGE; FOWLER, 2013). Existem dois tipos de relacionamento, um que funciona semelhante ao modelo relacional que é referenciando um documento de uma coleção pelo identificador, análogo a chave estrangeira no modelo relacional. Outro tipo é embutir documentos dentro de um documento, ou seja, agregação. Consistência existe apenas para os objetos da agregação, como relatado anteriormente. Quando distribuído, o MongoDB pode trabalhar com uma série de políticas de como irá se comportar no ambiente paralelo, que são configuráveis. Esse gênero não implementa transações<sup>7</sup>. O MongoDB permite que seja feito consultas com filtros e a linguagem utilizada é o JavaScript. Consultas mais complexas que utilizam os documentos embutidos em outros documentos são realizadas com *MapReduce* (SADALAGE; FOWLER, 2013). Por exemplo, se modelarmos esse banco com a o modelo da Figura 5 para fazer a busca dos pedidos do mês corrente,

<sup>5</sup> Sítio oficial do MongoDB <<http://www.mongodb.com/>>

<sup>6</sup> Sítio oficial do CouchDB <<http://couchdb.apache.org/>>

<sup>7</sup> Operações de inserir, atualizar ou deletar com a opção de emitir o *commit* ou *rollback*

diferente do [Redis](#), o banco é capaz de fazer esse filtro e retornar apenas os pedidos que têm a data igual a do mês corrente. Para isso ele utiliza *MapReduce*.

Diferentes bancos de dados foram desenhados para resolver diferentes problemas ([SADALAGE; FOWLER, 2013](#)). Conseguir identificar diferentes problemas de armazenamento, representação ou escalabilidade dentro de uma aplicação pode ser um indicativo que mais de um banco de dados deve ser utilizado. Da mesma maneira que diferentes paradigmas de linguagem de programação são utilizados em um desenvolvimento Web, diferentes gêneros de banco de dados podem ser usados em uma mesma aplicação ([WAMPLER; CLARK, 2010](#)). É considerado persistência poliglota quando uma aplicação utiliza mais de um banco de dados.

## 3 Implementação

Para comparar os sistemas que utilizam apenas um banco de dados (persistência monoglota) com os sistemas que utilizam mais de um banco de dados (persistência poliglota) implementamos e testamos o desempenho desses.

Escolhemos fazer uma aplicação semelhante ao Twitter. Nessa aplicação, o usuário poderá se cadastrar, escrever *tweets*, seguir outros usuários e listar os *tweets* dos usuários que ele segue.

Este capítulo é dividido em duas seções, a primeira, [Seção 3.1](#) descreve os casos de uso da aplicação. A [Seção 3.2](#) descreve os detalhes da implementação com persistência monoglota e com persistência poliglota.

### 3.1 Casos de Uso

Essa seção irá descrever os casos de uso da aplicação. Cinco subseções foram criadas para detalhar os cinco casos de uso criados. A primeira [Subseção 3.1.1](#), detalha o caso de uso que descreve a funcionalidade de cadastro do usuário. A [Subseção 3.1.2](#) descreve o caso de uso de cadastro de um *tweet*. A [Subseção 3.1.3](#) apresenta o caso de uso que descreve como o usuário segue outro usuário. A [Subseção 3.1.4](#) apresenta o caso de uso que descreve como o usuário para de seguir outro e por fim, a [Subseção 3.1.5](#) descreve o último caso de uso que detalha como o usuário visualiza a *feed*, página que apresenta todos os *tweets* dos usuários que o usuário logado segue.

#### 3.1.1 Caso de Uso 1 - Cadastro do usuário

Sumário: Usuário usa o sistema para se cadastrar

Ator primário: Usuário

Ator secundário: Sistema

Precondições: O usuário deverá ter uma email

Fluxo Principal

1. O usuário acessa o sistema;
2. O sistema exibe a tela de entrada, [Figura 7](#);
3. O usuário clica em *Sign up*;
4. O sistema retorna a tela com o formulário de cadastro de usuário, [Figura 8](#);

5. O usuário preenche os campos de nome, email, senha e confirmação de senha;
6. O sistema cria o usuário e redireciona para tela de *My Tweets*, [Figura 9](#).

Fluxo de Exceção (5): Email inválido

1. Se o usuário não digitou um email válido, o sistema reporta o campo incorreto e retorna ao passo 5.

Fluxo de Exceção (5): Campos obrigatórios não preenchidos

1. Se o usuário não preencher todos os campos, o sistema informa que os campos são obrigatórios e o usuário retorna ao passo 5.

Fluxo de Exceção (5): Senha e confirmação de senha não conferem

1. Se o usuário não preencher as senhas corretamente, o sistema retorna ao passo 5, informando que as senhas não conferem.

Figura 7 – Tela Inicial



Fonte: Autoria própria



Figura 8 – Tela de cadastro de usuário

Twitter/MongoDB About

### Sign up

\* Name

\* Email

\* Password

\* Password confirmation

[Sign up](#)

[Sign in](#)

Fonte: Autoria Própria

Figura 9 – Tela Tweets do Usuário Logado

Twitter/MongoDB Feed My Tweets Users Followers Following About Zico Limongi

### My Tweets

Tweet	Date		
asdasdasd asdasdasd asdasdasd asdasdasd asdasdasd asdasdasd asdasdasd asdasdasd asdasdasd asdasdasd asdasdasd asdasdasd	2014-08-19 02:19:18 UTC	<a href="#">Edit</a>	<a href="#">Destroy</a>
asdasdasd asdasdasd asdasdasd asdasdasd asdasdasd asdasdasd asdasdasd asdasdasd asdasdasd asdasdasd asdasdasd asdasdasd	2014-08-19 02:19:07 UTC	<a href="#">Edit</a>	<a href="#">Destroy</a>
asdasdasd asdasdasd asdasdasd asdasdasd asdasdasd asdasdasd asdasdasd asdasdasd asdasdasd asdasdasd asdasdasd asdasdasd	2014-08-19 02:18:58 UTC	<a href="#">Edit</a>	<a href="#">Destroy</a>

[New Tweet](#)

Fonte: Autoria Própria

### 3.1.2 Caso de Uso 2 - Cadastro de *tweet*

Sumário: Usuário usa o sistema para cadastrar um *tweet*

Ator primário: Usuário

Ator secundário: Sistema

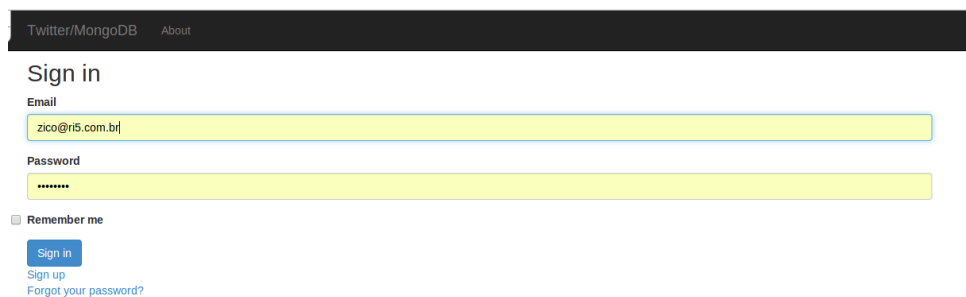
Precondições: O usuário deverá ser cadastrado e deve estar logado no sistema

Fluxo Principal

1. O usuário acessa o sistema e faz o *login*, [Figura 10](#);
2. O sistema abre a listagem dos *tweets* do usuário, [Figura 9](#);

3. O usuário clica no botão *New Tweet*;
4. O sistema retorna com o formulário para cadastrar *tweet*, [Figura 11](#);
5. O usuário escreve o *tweet* no campo indicado;
6. O sistema cria um novo *tweet*, registrando que o usuário é o autor do *tweet*, data e hora que foi criado.

Figura 10 – Tela de Login



Twitter/MongoDB About

### Sign in

Email

zico@r15.com.br

Password

\*\*\*\*\*

☐ Remember me

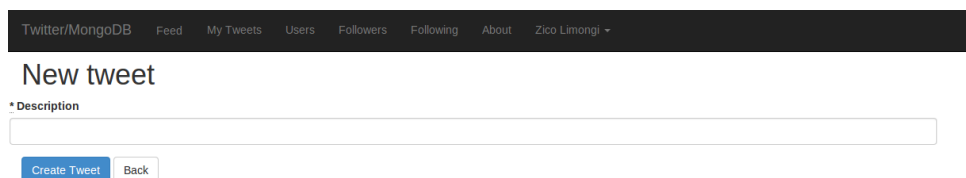
[Sign in](#)

[Sign up](#)

[Forgot your password?](#)

Fonte: Autoria Própria

Figura 11 – Tela Cadastro de Tweet



Twitter/MongoDB Feed My Tweets Users Followers Following About Zico Limongi

### New tweet

\*Description

[Create Tweet](#) [Back](#)

Fonte: Autoria Própria

### 3.1.3 Caso de Uso 3 - Usuário segue outro usuário

Sumário: Usuário segue outro usuário

Ator primário: Usuário

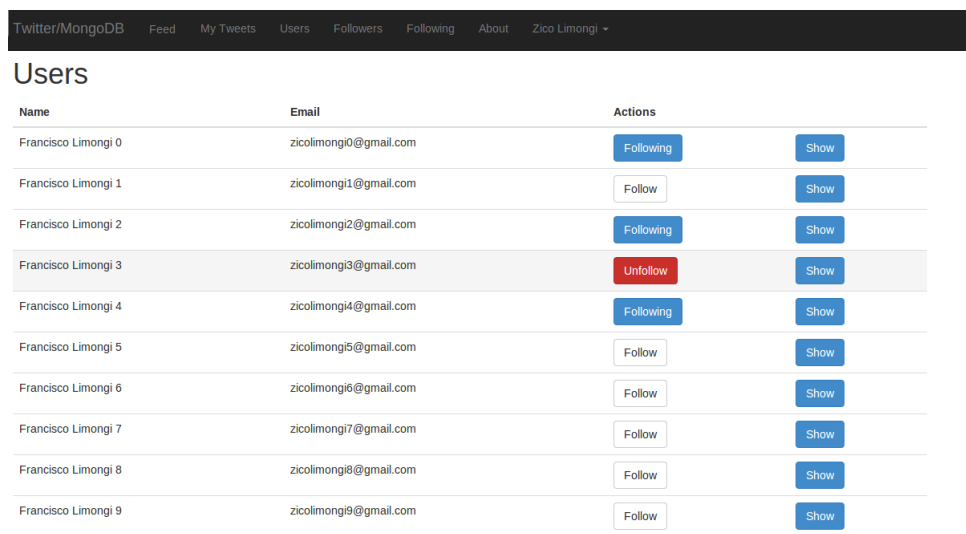
Ator secundário: Sistema

Precondições: O usuário deverá ser cadastrado e deve estar logado no sistema

Fluxo Principal

1. O usuário acessa o sistema e faz o *login*, [Figura 10](#);
2. O sistema abre a listagem dos *tweets* do usuário, [Figura 9](#);
3. O usuário clica no menu *Users*;
4. O sistema retorna a listagem paginada com todos os usuários cadastrados, [Figura 12](#);
5. O usuário escolhe algum usuário que deseja seguir e clica no botão *Follow*;
6. O sistema armazena essa informação.

Figura 12 – Tela de Listagem de Usuários



Name	Email	Actions
Francisco Limongi 0	zicolimongi0@gmail.com	<button>Following</button> <button>Show</button>
Francisco Limongi 1	zicolimongi1@gmail.com	<button>Follow</button> <button>Show</button>
Francisco Limongi 2	zicolimongi2@gmail.com	<button>Following</button> <button>Show</button>
Francisco Limongi 3	zicolimongi3@gmail.com	<button>Unfollow</button> <button>Show</button>
Francisco Limongi 4	zicolimongi4@gmail.com	<button>Following</button> <button>Show</button>
Francisco Limongi 5	zicolimongi5@gmail.com	<button>Follow</button> <button>Show</button>
Francisco Limongi 6	zicolimongi6@gmail.com	<button>Follow</button> <button>Show</button>
Francisco Limongi 7	zicolimongi7@gmail.com	<button>Follow</button> <button>Show</button>
Francisco Limongi 8	zicolimongi8@gmail.com	<button>Follow</button> <button>Show</button>
Francisco Limongi 9	zicolimongi9@gmail.com	<button>Follow</button> <button>Show</button>

Fonte: Autoria Própria

### 3.1.4 Caso de Uso 4 - Usuário para de seguir algum usuário

Sumário: Usuário para de seguir algum usuário

Ator primário: Usuário

Ator secundário: Sistema

Precondições: O usuário deverá ser cadastrado e deve estar logado no sistema. Além disso, o usuário logado deve estar seguindo o usuário que ele deseja parar de seguir

#### Fluxo Principal

1. O usuário acessa o sistema e faz o *login*, [Figura 10](#);
2. O sistema abre a listagem dos *tweets* do usuário, [Figura 9](#);
3. O usuário clica no menu *Users*;
4. O sistema retorna a listagem paginada com todos os usuários cadastrados, [Figura 12](#);
5. O usuário encontra o usuário que deseja parar seguir e clica no botão *Unfollow*;
6. O sistema armazena essa informação.

### 3.1.5 Caso de Uso 5 - Usuário visualiza a *feed*

Sumário: Usuário visualiza os *tweets* dos usuários que ele segue

Ator primário: Usuário

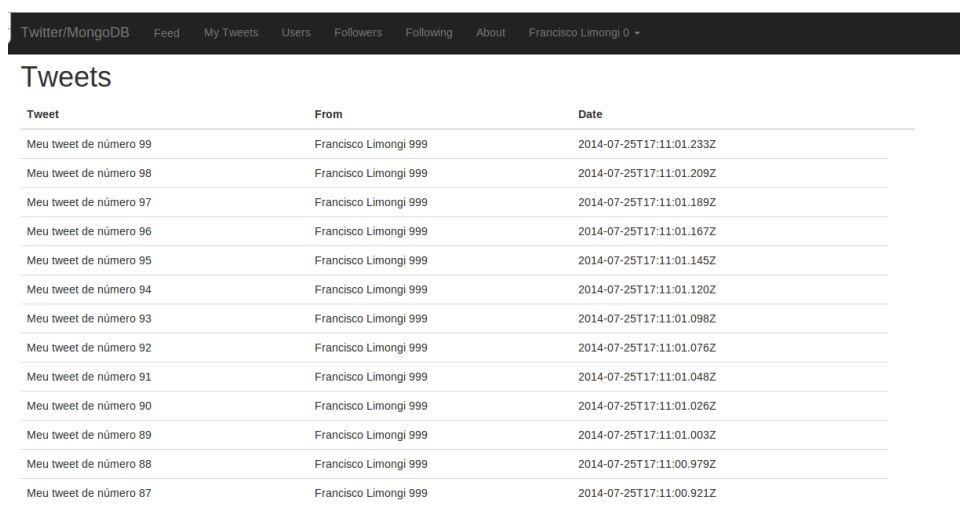
Ator secundário: Sistema

Precondições: O usuário deverá ser cadastrado e deve estar logado no sistema. Além disso, o usuário logado deve estar seguindo algum usuário

#### Fluxo Principal

1. O usuário acessa o sistema e faz o *login*, [Figura 10](#);
2. O sistema abre a listagem dos *tweets* do usuário, [Figura 9](#);
3. O usuário clica no menu *Feed*;
4. O sistema retorna a listagem paginada com todos os *tweets* dos usuários que o ator primário segue, [Figura 13](#).

Figura 13 – Tela de Feed



Tweet	From	Date
Meu tweet de número 99	Francisco Limongi 999	2014-07-25T17:11:01.233Z
Meu tweet de número 98	Francisco Limongi 999	2014-07-25T17:11:01.209Z
Meu tweet de número 97	Francisco Limongi 999	2014-07-25T17:11:01.189Z
Meu tweet de número 96	Francisco Limongi 999	2014-07-25T17:11:01.167Z
Meu tweet de número 95	Francisco Limongi 999	2014-07-25T17:11:01.145Z
Meu tweet de número 94	Francisco Limongi 999	2014-07-25T17:11:01.120Z
Meu tweet de número 93	Francisco Limongi 999	2014-07-25T17:11:01.098Z
Meu tweet de número 92	Francisco Limongi 999	2014-07-25T17:11:01.076Z
Meu tweet de número 91	Francisco Limongi 999	2014-07-25T17:11:01.048Z
Meu tweet de número 90	Francisco Limongi 999	2014-07-25T17:11:01.026Z
Meu tweet de número 89	Francisco Limongi 999	2014-07-25T17:11:01.003Z
Meu tweet de número 88	Francisco Limongi 999	2014-07-25T17:11:00.979Z
Meu tweet de número 87	Francisco Limongi 999	2014-07-25T17:11:00.921Z

Fonte: Autoria Própria

## 3.2 Sistema Desenvolvido

Ambos os sistemas foram desenvolvidos na linguagem *ruby* com o *framework rails*. Esse *framework* é próprio para Web e utiliza a arquitetura de *software Model View Controller (MVC)*. Além disso, é uma ferramenta de desenvolvimento ágil, que facilita o desenvolvimento Web. Foram utilizado outras *gems* <sup>1</sup>, a lista utilizada em cada aplicação criada se encontra no arquivo chamado Gemfile.lock. Podemos destacar a *gem Devise* <sup>2</sup> e a *gem Mongoid* <sup>3</sup> como principais. A Devise faz o controle de sessão do usuário e a Mongoid deixa transparente para o desenvolvedor a comunicação com o banco de dados MongoDB. Também utilizamos a *gem bootstrap-sass* para melhorar usabilidade da aplicação.

Esta seção foi dividida em duas subseções. A [Subseção 3.2.1](#) descreve os detalhes de implementação do sistema monoglota. e a [Subseção 3.2.2](#) descreve os detalhes da implementação poliglota.

A seguir apresentamos os detalhes de implementação de cada sistema.

### 3.2.1 Implementação com Persistência Monoglota

Essa implementação utiliza apenas o banco de dados MongoDB que é orientado a documento. O código da aplicação está disponível no GitHub <sup>4</sup>.

<sup>1</sup> Gem é análogo a biblioteca nas outras linguagens

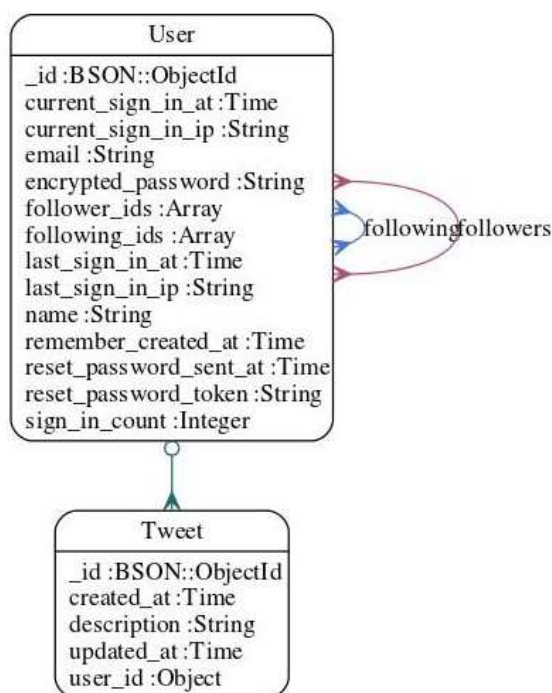
<sup>2</sup> <<https://github.com/plataformatec/devise>>

<sup>3</sup> <<http://mongoid.org/>>

<sup>4</sup> <<https://github.com/zicolimongi/Twitter-Mongo>>

Na camada de modelos temos apenas duas classes, `Tweet` e `User`. A classe `Tweet` representa o *tweet* e a classe `User` representa o usuário. Ambos são armazenados como coleção no MongoDB. Cada objeto de uma dessas classes é um documento. Na [Figura 14](#) podemos visualizar o relacionamento entre as classes.

Figura 14 – Camada de modelo



Fonte: ([SADALAGE; FOWLER, 2013](#))

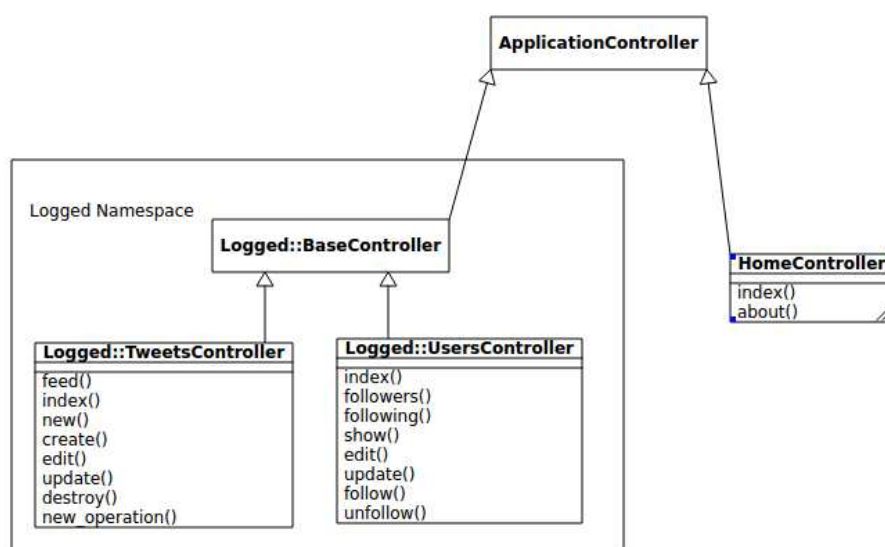
A classe `Tweet` armazena identificador do próprio *tweet*, data de criação, data da última atualização desse objeto, descrição do *tweet* e o identificador do usuário que criou o *tweet*. Não escolhemos usar agregação nessa relação, pois uma das principais perspectivas é visualizar os `Tweets` de diferentes usuários. Nessa classe temos duas validações: uma de presença do campo descrição e outra que limita o tamanho desse campo em cento e quarenta caracteres.

A classe `User` armazena o identificador do usuário, email, nome, uma série de campos usados para autenticação, data de criação do objeto, data da última modificação do objeto, lista de usuários seguidos e a lista de usuários seguidores. A classe `User` tem três relacionamentos, um com `Tweet` que já foi citado e dois autorrelacionamentos para armazenar a lista de usuários que seguem e que são seguidos. Esse autorrelacionamento no banco de dados orientado a documento funciona de uma maneira diferente do banco de dados relacional. Enquanto no modelo relacional é necessário criar uma nova tabela, no modelo não-relacional armazenamos apenas as listas com os identificadores dos usuários, pois é permitido valores multivalorados. Logo, basta armazenar os identi-

ficadores da relação em uma lista embutida no documento do usuário. Então temos uma lista que armazena os identificadores dos usuários que são seguidos, chamada de `following` e outra lista com os identificadores dos usuários que são seguidores, chamada de `followers`. Essa classe tem três validações de presença: uma para o campo nome, outra para o campo email e a terceira para o campo de senha.

Na camada de controle temos uma hierarquia de classes conforme mostra a [Figura 15](#). `ApplicationController` é a classe de controle que herda da classe do *framework*. Em seguida, temos o `HomeController` e `Logged::BaseController` que herda da classe `ApplicationController`. `HomeController` é a classe responsável por implementar o controle das páginas de `about` e `index`. Já o `Logged::BaseController` é a classe responsável por carregar o layout do usuário logado e verificar a autenticidade do usuário. Abaixo do `Logged::BaseController` foram criadas mais duas classes: `TweetsController` e `UsersController` que implementam os controles de *tweets* e usuários respectivamente.

Figura 15 – Camada de controle



Fonte: (SADALAGE; FOWLER, 2013)

Toda consulta que resulte em mais de trinta registros são paginadas, isto é, colocamos um limite para que o banco não busque mais que 30 registros de uma só vez. Isso funciona manipulando duas variáveis `offset` e `limit`. Também devemos ressaltar que antes de todas ações é executado uma consulta, no banco, que carrega para variável `current_user` o usuário logado, caso exista. Essa busca é feita pela *gem* `Devise` e é transparente para o desenvolvedor.

O controle de usuário implementa as ações `index`, `followers`, `following`, `show`, `edit`, `update`, `follow` e `unfollow`, todas descritas a seguir:

A ação `index` lista todos os usuários com exceção do usuário logado. Na página renderizada, o usuário logado poderá seguir os usuários que ele ainda não segue, ou poderá parar de seguir os usuários que ele segue. É realizada uma consulta para ler os usuários do sistema.

A ação `followers` lista todos os usuários que seguem o usuário logado. Na página renderizada, o usuário logado poderá seguir os usuários que ele ainda não segue, ou poderá parar de seguir os usuários que ele já segue. É realizada uma consulta no banco de dados que busca todos os usuários que o usuário logado segue.

A ação `followings` lista todos os usuários que o usuário logado segue. Na página renderizada, o usuário logado poderá parar de seguir os usuários que ele segue. Da mesma maneira que a ação `followers` é realizado uma consulta, porém busca os usuários que seguem o usuário logado.

A ação `show` recebe um identificador como parâmetro e localiza o usuário que possui esse identificador. A página renderizada mostra o nome e email do usuário localizado e os *tweets* que ele fez. Nessa página, são realizadas duas consultas, uma que busca o usuário e outra que busca os *tweets* desse usuário.

A ação `edit` renderiza o formulário de edição para que o usuário logado possa mudar algum campo, como nome, email ou senha. Nessa ação nenhuma consulta adicional é feita.

A ação `update` recebe os parâmetros das alterações que o usuário fez no próprio perfil e registra isso no banco de dados. É realizado uma escrita no banco de dados para atualizar esses dados passados por parâmetro.

A ação `follow` recebe como parâmetro o usuário que será seguido pelo usuário logado e registra esse relacionamento. Para isso é necessário uma consulta para ler qual usuário será seguido e duas escritas: uma para atualizar o usuário que está sendo seguido com o identificador do usuário seguidor e outra para atualizar o usuário seguidor com o identificador do usuário seguido.

A ação `unfollow` recebe como parâmetro o usuário que deixará de ser seguido pelo usuário logado e apaga o registro desse relacionamento. Para isso é necessário uma consulta para ler o usuário que deixará de ser seguido e duas escritas no banco de dados: uma para atualizar a lista de seguidores do usuário que deixou de ser seguido e outra para atualizar a lista de seguindo do usuário seguidor que deixou de seguir.

O controle de *tweets* implementa as ações `feed`, `index`, `new`, `edit`, `update` e `destroy`, descritas a seguir:



A ação `feed` é responsável por fazer a busca no banco de dados dos *tweets* de todos os usuários que o usuário logado segue, ordenado-os pela data de criação de forma descendente, isto é, do *tweet* mais recente para o mais antigo.

A ação de `index` lista todos os *tweets* do usuário logado, para isso foi necessário fazer uma consulta no banco de dados que busca esses *tweets*.

A ação `new` renderiza o formulário de cadastro de *tweet*, não é feito nenhuma consulta adicional.

A ação `create` recebe como parâmetro o campo de descrição do *tweet*, relaciona esse *tweet* com o usuário logado e faz a escrita no banco de dados, caso seja válido.

A ação `update` recebe como parâmetro o campo de descrição e o identificador do *tweet* que será alterado, faz a leitura no banco de dados desse *tweet* com o identificador passado. Em seguida, altera o valor da descrição e faz a escrita no banco de dados.

A ação `destroy` recebe como parâmetro o identificador do *tweet* que será excluído, em seguida faz a leitura desse *tweet* para verificar se o usuário logado é o autor do *tweet* e caso seja confirmado, faz exclusão desse do banco de dados.

### 3.2.2 Implementação com Persistência Poliglota

Para fazer a aplicação com persistência poliglota foi feito uma cópia da aplicação com persistência monoglota e em seguida feito as alterações necessárias para que o segundo banco de dados fosse usado. O código da aplicação está disponível no GitHub<sup>5</sup>. Na aplicação monoglota a única maneira de consultar os *tweets* na *feed* é buscar todos os *tweets*, cujos o autores estejam na lista de *followers* do usuário logado. Para isso, era necessário varrer *tweet* por *tweet* e verificar se o autor está na lista. Pensando nessa abordagem utilizamos a persistência poliglota para melhorar o tempo de leitura da *feed* de *tweets*. Então utilizamos um segundo banco de dados, do gênero chave-valor, chamado [Redis](#), que armazena os cem *tweets* mais recentes, cujos autores estão na lista de *followers* do usuário logado. O [Redis](#) irá armazenar esses cem *tweets* em uma chave com o identificador do usuário concatenada com a string `"_feed"`, ou seja, o usuário que tem o identificador igual a um, a chave será `"1_feed"`. Todo usuário terá uma chave que armazenará os *tweets* da *feed*. Quando ele for acessar a página de *feed*, ao invés do sistema fazer a consulta no MongoDB e varrer *tweet* por *tweet*, ele irá apenas consultar o [Redis](#) com a chave e será retornado o valor com os cem *tweets* mais recentes que o usuário logado segue.

Para persistir esses dados no [Redis](#), precisamos alterar o sistema na camada de controle e na camada de dados. Inicialmente foi necessário instalar outra *gem*,

<sup>5</sup> <<https://github.com/zicolimongi/Twitter-Mongo-Redis>>

chamada `redis`<sup>6</sup>, que faz a comunicação com o banco de dados chave-valor. Em seguida, implementamos três métodos na classe `User`: `feed_key`, `remake_feed` e `update_tweet_hash`, descritos a seguir:

O método `feed_key` foi criado apenas para retornar a chave que será usada para armazenar no [Redis](#).

O método `remake_feed` foi criado para refazer o valor da `feed` de algum usuário quando necessário. Após fazer a implementação poliglota, utilizamos esse método para popular o [Redis](#).

O método `update_tweet_hash` foi implementado para adicionar mais um *tweet* na `feed` de *tweets* do usuário.

Também precisamos adicionar três métodos na classe `Tweet`: o método `feed_of`, `to_redis_json`, `update_hash`.

O método `feed_of` é estático, pois irá trazer uma coleção de *tweets*. Esse método recebe como parâmetro um usuário, acessa o `redis` com a chave desse usuário e em seguida, faz um parse do valor lido, que foi armazenado como JSON, para aplicação.

O método `to_redis_json` é usado para converter o objeto *tweet* no formato JSON para que possa ser armazenado no [Redis](#).

Por fim, o terceiro método criado é o `update_hash`, privado, que é responsável por atualizar todas as chaves com o *tweet* que foi criado. Esse método é executado por um *callback*, chamado `after_create`, com isso, toda a vez que for criado um *tweet* pela aplicação o método `update_hash` será rodado.

Ainda foi necessário modificar o controle de *tweet*, para que a ação `feed` leia os *tweets* do [Redis](#) e não do MongoDB.

---

<sup>6</sup> <<https://github.com/antirez/redis>>

## 4 Resultados

Este capítulo irá mostrar os resultados da comparação dos sistemas criados, um que utiliza a persistência monoglota e outro que utiliza a persistência poliglota. O modelo de persistência poliglota tinha como alvo melhorar o desempenho da página *feed*, mas sabemos que para isso a escrita do *tweet* iria ser mais lenta. Pois, ao usuário publicar o *tweet* é necessário atualizar a *feed* de todos os seguidores desse usuário. Então iremos comparar o tempo de consulta da *feed* de *tweets* e o tempo de inserção de um *tweet*.

O capítulo foi dividido em três seções, a [Seção 4.1](#) que descreve como foram feito os testes para medir tempo de consulta da *feed* de *tweets*, a [Seção 4.2](#) descreve como foram feito os testes para medir o tempo de inserção de um *tweet* e a [Seção 4.3](#) irá analisar os resultados encontrados.

Para medir o tempo das operações utilizamos uma classe do próprio *Ruby*, chamada *Benchmark*<sup>1</sup>. Dessa classe utilizamos o método chamado *measure* que mede o tempo de execução. Esse tempo medido está incluído o tempo da operação no banco de dados e o tempo do carregamento dos valores para as variáveis do sistema.

Os testes foram realizados na máquina Asus, modelo N82J<sup>2</sup>, porém foi adicionado, nessa máquina, mais 4GB de RAM e o sistema operacional é Ubuntu 14.04 LTS.

### 4.1 Resultados do tempo de consulta da *feed* de *tweets*

Para fazer um teste, no qual esses tempos medidos resultassem em uma diferença significativa criamos quatro bancos de dados do MongoDB. Esses bancos foram populados com cem usuários, porém a diferença entre os bancos foi a quantidade de *tweets* de cada usuário. No primeiro banco de dados foi adicionado dez *tweets* para cada usuário, totalizando em mil *tweets*, no segundo foram adicionados cem *tweets* para cada usuário, totalizando em dez mil *tweets*, no terceiro banco foram adicionados mil *tweets* para cada usuário totalizando em cem mil *tweets* e, por último, foram adicionados, no quarto banco de dados, dez mil *tweets* para cada usuário, totalizando em um milhão de *tweets*.

Alteramos o primeiro usuário para seguir todos os outros noventa e nove usuá-

<sup>1</sup> A documentação da classe *Benchmark*. <<http://www.ruby-doc.org/stdlib-2.0/libdoc/benchmark/rdoc/Benchmark.html>>

<sup>2</sup> A especificação se encontra no sítio da Asus <[http://www.asus.com/Notebooks\\_Ultrabooks/N82Jq/](http://www.asus.com/Notebooks_Ultrabooks/N82Jq/)>

rios e executamos o método `remake_feed` da classe `User` para criar a chave no [Redis](#).

No modelo monoglota medimos o tempo de execução da seguinte função `Tweet.in(user_id: user.following_ids).desc("created_at").paginate(:page`. Essa função irá buscar os cem *tweets* mais recentes que o usuário atribuído ao objeto *user* segue. Repetimos isso cem vezes para cada banco de dados, que havíamos populado. Segue abaixo a tabela com esses resultados:

No modelo poliglota medimos o tempo de execução da seguinte função `Tweet.feed_of u` que busca no [Redis](#) a chave do usuário *user* que contém os cem *tweets* mais recentes que esse usuário segue. Também medimos cem vezes para cada banco de dados, que havíamos populado. Segue abaixo a tabela com esses resultados:

## 4.2 Resultados do tempo de inserção de um *tweet*

Na implementação poliglota a cada *tweet* inserido por um usuário, a chave de todos os seguidores é refeita no [Redis](#). Logo, para ressaltar essa desvantagem populamos o banco de dados com dez mil e um usuários, cada um com cem *tweets*, totalizando mais que um milhão de *tweets*.

Utilizamos o último usuário criado para realizar os testes. Alteramos a quantidade de seguidores desse usuário para comparar o quanto isso irá impactar no resultado. Testamos com cem, mil, e dez mil seguidores e repetimos esse teste cem vezes para cada conjunto. Para medir o tempo de inserção em ambos os sistemas utilizamos a mesma função. A diferença da implementação poliglota é que foi criado um *call back* que é executado após o *tweet* ser criado. Esse *call back* realiza uma operação de leitura e outra de escrita no [Redis](#) para cada seguidor. Os resultados seguem descrito na tabela abaixo:

## 4.3 Análise dos resultados

Para a implementação monoglota tivemos piores resultados na consulta a *feed* de *tweets*. A média aumentou significativamente quando a quantidade de *tweets* no banco foram aumentados. Isso era esperado acontecer, pois para o banco encontrar os documentos procurados, foi necessário percorrer todos os *tweets* e comparar com os parâmetros passados na consulta. Já para a implementação poliglota, o tempo de execução para as diferentes quantidades de *tweets* foram muito próximos, pois a consulta é apenas para buscar a chave, não há nenhuma outra comparação ou leitura a ser feita. Com isso, podemos observar que tivemos uma melhora significativa, pois em nenhum momento a consulta a *feed* de *tweets* de um usuário foi mais rápida no sistema monoglota.

Em relação aos resultados do tempo de inserção do *tweet*, podemos observar que em nenhum instante a implementação poliglota foi mais rápida. Isso é devido ao

tempo que foi gasto para atualizar as chaves. O aumento de seguidores foi proporcional com o aumento do tempo, ou seja, se aumentarmos em dez vezes os seguidores o tempo de atualização das chaves será dez vezes maior. Isso porque precisamos de fazer uma leitura e uma escrita para cada chave ser atualizada. Já para a implementação monoglota não houve diferença, pois a inserção de um *tweet* não depende da quantidade de seguidores.

## 5 Conclusão

A persistência poliglota pode melhorar o desempenho da aplicação, mas o modelo deve ser bem estudado, para que haja uma melhora no sistema como um todo. Os resultados demonstram que a persistência poliglota melhorou parte do sistema, mas não o sistema como um todo.

Para o funcionamento do Twitter o modelo utilizado não foi ideal, pois o tempo de inserção de um *tweet* aumentou muito. Porém, é um modelo que funcionaria muito bem em aplicações que há uma grande quantidade de leitura e uma pequena quantidade de escrita.

Este trabalho apresentado é apenas um começo do estudo de persistência poliglota. Podemos fazer diferentes análises, como a quantidade de espaço utilizado, consistência das informações e outras para afirmar qual modelo é melhor. A próxima etapa seria aprofundar os testes realizados nas duas aplicações e em seguida colocar ambas em um ambiente paralelo.

## Referências

- CHANG, F.; DEAN, J.; GHEMAWAT, S.; HSIEH, W. C.; WALLACH, D. A.; BURROWS, M.; CHANDRA, T.; FIKES, A.; GRUBER, E., R. Bigtable: A distributed storage system for structured data. In: **Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7**. Berkeley, CA, USA: USENIX Association, 2006. (OSDI '06), p. 15–15. Disponível em: <http://dl.acm.org/citation.cfm?id=1267308.1267323>. Citado na página 6.
- DATE, C. **Introdução a Sistema de Bancos de Dados**. 8º. ed. [S.l.]: Elsevier Editora LTDA, 2004. Citado 2 vezes nas páginas 1 e 4.
- DECANDIA, G.; HASTORUN, D.; JAMPANI, M.; KAKULAPATI, G.; LAKSHMAN, A.; PILCHIN, A.; SIVASUBRAMANIAN, S.; VOSSHALL, P.; VOGELS, W. Dynamo: Amazon's highly available key-value store. **ACM Symposium on Operating Systems Principles**, v. 21st, 2007. Citado na página 6.
- ELMASRI, S. B. N. R. **Sistema de Banco de Dados**. 4º. ed. [S.l.]: Pearson Education do Brasil Ltda, 2005. 7,8,9 p. Citado 3 vezes nas páginas 4, 5 e 7.
- EVANS ERIC/FOWLER, M. **Domain-Driven Design**. [S.l.]: Prentice Hall, 2003. Citado na página 8.
- REDMOND, E.; WILSON, J. R. **Seven Databases in Seven Weeks**. 1º. ed. [S.l.]: Pragmatic Programmers, LLC, 2012. Citado 3 vezes nas páginas 1, 2 e 6.
- SADALAGE, P. J.; FOWLER, M. **NoSql, A Brief Guide to the Emerging World of Polyglot Persistence**. 8º. ed. [S.l.]: Pearson Education, Inc, 2013. Citado 12 vezes nas páginas 3, 6, 7, 8, 9, 10, 11, 12, 13, 14, 22 e 23.
- WAMPLER, D.; CLARK, T. Guest editors' introduction: Multiparadigm programming. **IEEE Software**, IEEE Computer Society, Los Alamitos, CA, USA, v. 27, n. 5, p. 20–24, 2010. ISSN 0740-7459. Citado na página 14.