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
JULHO 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 LaTeX.

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
JULHO 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 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
JULHO 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 9
Figura 5 – Exemplo de um diagrama UML para utilizando agregação entre cliente e pedido 10
Figura 6 – Exemplo da disposição dos dados no NoSQL, utilizando agregação entre cliente e pedido 11
Figura 7 – Camada de controle 17
Figura 8 – Camada de controle 19

Lista de Tabelas

Lista de Algoritmos

SQL Structure Query Language 1

Redis Remote Dictionary Server 1

SGBD Sistema de Gerenciamento do Banco de Dados 5

UML Unified Model Language iv

UoD universo de discurso 4

ACID Atomicidade, Consistência, Isolamento e Durabilidade 11

XML *eXtensible Markup Language* 12

MVC *Model View Controller* 16

Sumário

1 – Introdução **1**
1.1 Motivação 2

2 – Fundamentação Teórica **4**
2.1 Banco de Dados 4
2.2 Gêneros de Banco de dados 5
2.2.1 Banco de dados Relacional 6
2.2.2 Banco de dados não-relacional 8
2.2.3 Persistência Poliglota 13
2.3 Implementação 14
2.4 Implementação Monoglota 17
2.5 Implementação Poliglota 21

Referências **23**

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 gêneros de banco de dados, 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 gênero é necessário realizar consultas com a linguagem Structure Query Language (SQL). Alguns exemplos de banco de dados relacional são MySQL [1], Oracle [2] e PostgreSQL [3].

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, replicação simples, alta disponibilidade e novos métodos de consultas (REDMOND; WILSON, 2012). Essas opções são conhecidas como NoSQL ou banco de dados não-relacional

Existem diversos gêneros de banco de dados 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, o primeiro que utilizará apenas um banco de dados, que será do gênero orientado a documento, e o segundo que 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 *Ruby on Rails* que

[1] Sítio oficial <http://www.mysql.com>
[2] Sítio oficial <http://www.oracle.com/technetwork/oem/db-mgmt/db-mgmt-093445.html>
[3] Sítio oficial <http://www.postgresql.org/>

oferece excelentes bibliotecas para esses bancos. O intuito desse trabalho é comprovar que o uso da persistência poliglota melhora o desempenho da aplicação.

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 gênero é a maneira que os registros são armazenados. Cada registro fica armazenado em um documento que é análogo à tupla no modelo relacional. O documento é composto por um identificador único e um conjunto de valores de tipos e estruturas aninhadas. Esse gênero é bem flexível, pois não tem catálogo, ou seja, não existe tabela 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, tem soluções para tratar concorrência e foi desenhado para trabalhar em *clusters*.A linguagem utilizada para fazer consulta no MongoDB é JavaScript. Nesse trabalho iremos utilizar o banco MongoDB nos dois sistemas que serão criados. Esse banco está sendo utilizado em grandes empresas, como Cisco, eBay, Codeacademy, Microsoft, Craiglist, The Guardian e outras, conforme sítio oficial do MongoDB [4].

O segundo banco que iremos utilizar se chama Redis do gênero chave-valor. Esse tipo de armazenamento é mais simples, como próprio nome indica, é armazenado um valor para determinada chave. O valor armazenado pode ter uma estrutura variável. Essa escolha foi feita devido ao cache que esse sistema realiza antes de efetivar a operação no disco. Esse cache 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. O Redis está sendo utilizado no Twitter, Github, Craiglist e outros conforme referência do sítio oficial [5].

1.1 Motivação

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

Exitem duas variáveis opostas no ambiente de persistência de dados, consistência e disponibilidade. Quanto mais consistente um dado, menos disponível ele será e quanto mais disponível um dado menos consistente ele estará. Em aplicações é comum termos

[4] <http://www.mongodb.com/customers>
[5] <http://redis.io/topics/whos-using-redis>

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 devem ser utilizados dois banco 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, sistema monoglota, e sistemas que utilizam persistência poliglota.

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.

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 muito 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 processar 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 de 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. 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 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 transação 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 de bancos de dados sem catálogo, 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 múltipla visões e multiusuários.

Como revelado acima, a utilização do banco de dados facilita o desenvolvimento das aplicações, faz 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, chama a atenção da comunidade científica, depois da publicação de dois artigos BigTable (CHANG et al., 2006) e Dynamo (DECANDIA et al., 2007)

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. O aumento

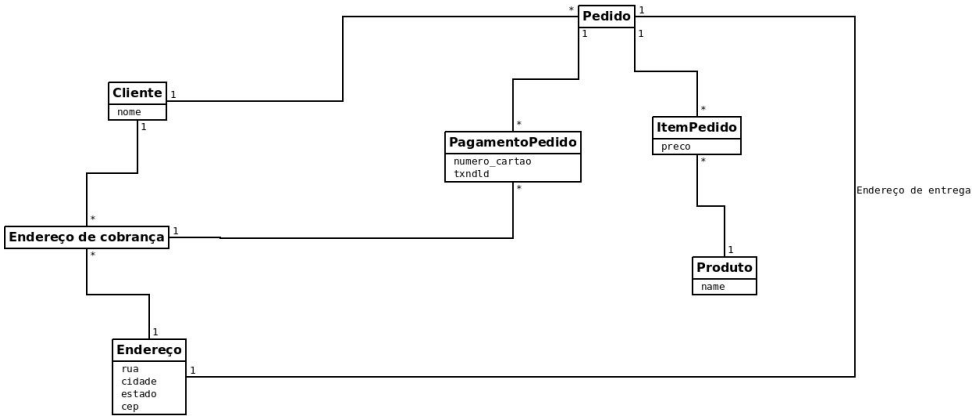
do volume de dados fez com que os especialistas buscassem novas soluções, que permitissem o armazenamento paralelo dos dados, pois o modelo relacional não foi desenhado para funcionar em *clusters*. Logo, os bancos de dados NoSQL se destacaram por funcionar bem no ambiente paralelo e ter um melhor desempenho com grande volume de dados (SADALAGE; FOWLER, 2013).

Os bancos de dados NoSQL têm as seguintes características: não usa o modelo relacional, foram desenhados para funcionar em *clusters*, são *open source* e não tem catálagos (*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. Uma conferência, realizada em São Francisco nos Estados Unidos em Junho de 2009, foi responsável por denominar esses bancos de dados de NoSQL. Johan Oskarsson que organizou essa conferência escolheu esse nome, porque era uma boa hashtag no Twitter: pequeno, memorável e tinha poucos resultados no Google. Isso facilitaria os interessados a encontrar a conferência. Apesar de o 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

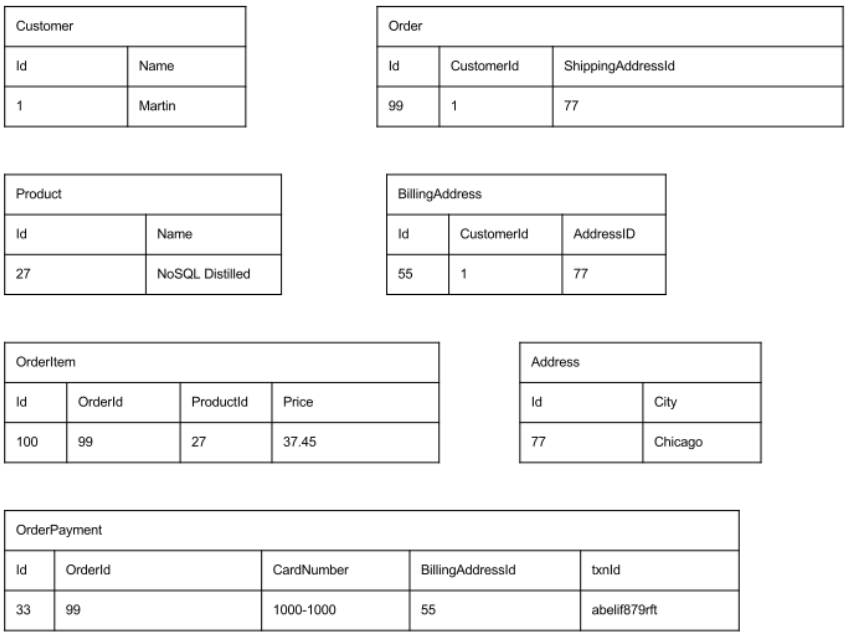
O modelo relacional armazena os dados em tabelas de duas dimensões em linhas e colunas. A interação com esse banco é feito por um SGBD que utiliza o SQL como linguagem de consulta de dados. Os dados armazenados são valores tipados e podem ser numéricos, texto, data e outros tipos, que são configurados e forçados pelo sistema. É possível fazer relações entre tabelas e cruzar 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. Um sistema de *e-commerce* poderia ser desenhado conforme o diagrama da Figura 1 e os valores ficariam dispostos 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	
Id	Name
1	Martin

Order		
Id	CustomerId	ShippingAddressId
99	1	77

Product	
Id	Name
27	NoSQL Distilled

BillingAddress		
Id	CustomerId	AddressID
55	1	77

OrderItem			
Id	OrderId	ProductId	Price
100	99	27	37.45

Address	
Id	City
77	Chicago

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

Fonte: (SADALAGE; FOWLER, 2013)

O gênero 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 frustação em muitos desenvolvedores, chamada de Impedância de Correspondência ou *Impedance Mismatch*. 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 gênero 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 é definido previamente 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 preocupar com qual valor colocar para a 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, pois 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: são 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 consequentemente, isso traria uma quantidade grande de campos vazios.

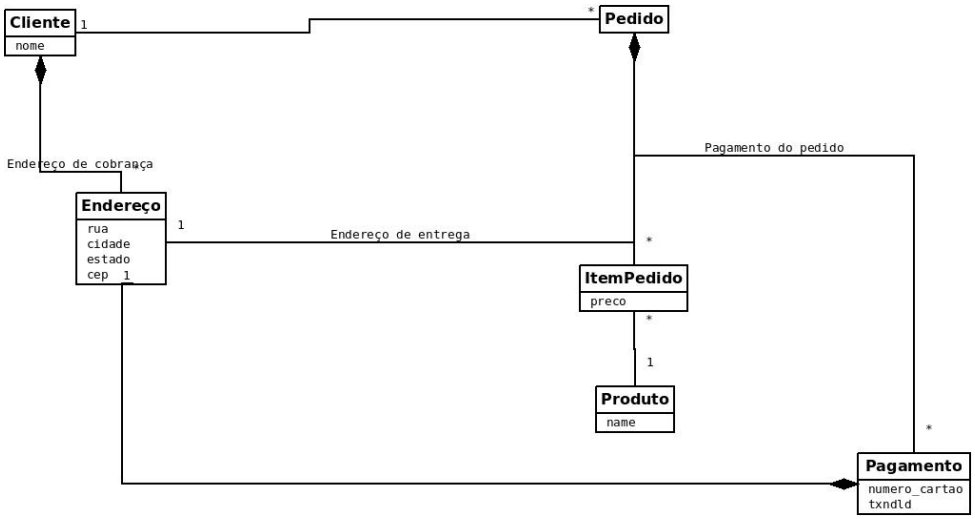
Basicamente os banco de dados NoSQL deslocam a definição do esquema para a aplicação que acessa o banco. Isso pode se tornar problemático quando há muitas aplicações acessando o mesmo banco, mas existem soluções para resolver isso, como encapsular toda a interação com o banco de dados, fazendo esse funcionar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

Além dessa flexibilidade, a maioria dos gêneros NoSQL permitem estruturar dados multivalorados [1]. Esses dados são modeladas como uma 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 com uma unidade, ou seja, é lido e escrito, sempre, todo o conjunto de dados. Como, geralmente, ao utilizar esse relacionamento buscamos operações atomicas, essa abordagem se encaixa perfeitamente com a aplicação (SADALAGE; FOWLER, 2013). A modelagem em UML do mesmo exemplo anterior utilizando agregação, ficaria como a Figura 3 e a disposição de dados ficaria como a Figura 4 [2].

[1] O banco de dados orientado a nó, é um dos gêneros que não implementam agregação.

[2] É usual a utilização de JSON para mostrar os dados em NoSQL (SADALAGE; FOWLER, 2013)

Figura 3 – Exemplo de um diagrama UML utilizando agregação



Fonte: (SADALAGE; FOWLER, 2013)

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",
        "txdId": "abelif879rft"
        "endereco_cobranca" {"cidade": "Chicago"}
        }]
}
```

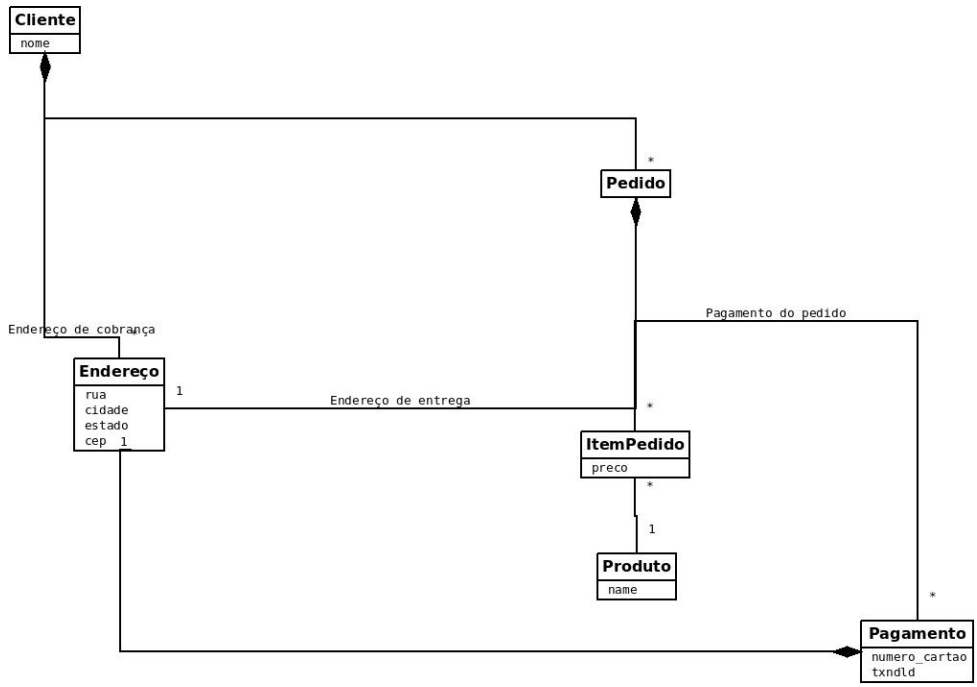
Fonte: (SADALAGE; FOWLER, 2013)

Nesse modelo, utilizamos duas agregações principais: cliente e pedido. O cliente contém uma lista de endereços, 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 adequa ao problema, pois nesse caso não queremos que o endereço em pagamento, ou em pedido seja alterado caso o cliente atualize a própria lista. Utilizando o modelo relacional temos duas maneiras de lidar com o problema. A primeira, 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.

A ligação entre pedido e cliente é uma relação que ocorre da mesma maneira que o modelo relacional. O pedido mantém um identificador do cliente, mas também tem como utilizar a agregação para essa relação. 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 modelamento não existe a melhor solução, mas sim uma que se adequa melhor para um certo tipo de problema, depende completamente da natureza da aplicação. Se for interessante para a aplicação listar o histórico dos pedidos, o segundo modelo, utilizando NoSQL, não é o ideal, pois será necessário entrar em cada usuário para ler os pedidos. Já no primeiro modelo, utilizando NoSQL, essa consulta do histótico se torna trivial (SADALAGE; FOWLER, 2013).

Outra razão para utilizar agregação é que isso facilita o uso do banco de dados em *clusters*, pois isso informa ao sistema, quais *bits* dos 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 dessas vantagens apresentadas sobre agregação, o modelo relacional tem suas vantanges 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 triviais ou até mesmo mais lentas.

Outra desvantagem do NoSQL, é não implementar transações Atomicidade, Consistência, Isolamento e Durabilidade (ACID) sobre várias agregações. 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.

O objeto desse trabalho são duas aplicações que irão utilizar 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[3] e Riak[4] A consistência utilizando 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 a 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. A consulta sobre os valores é feita apenas buscando a chave, qualquer outro filtro necessário, deverá ser feito pela aplicação, após ler todo o valor retornado. Também é possível programar o tempo de expiração da chave. Essa maneira que o banco chave-valor trabalha o 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 valores que precisam ser filtrados.

O gênero orientado a documento se 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 podem não ser idênticos. Se fizermos uma analogia, a coleção seria a tabela no modelo relacional e o documento seria a tupla no modelo relacional. Exemplo desse gênero são MongoDB [5] e CouchDB [6](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 . A outra maneira 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 um série de políticas que são configuráveis. Esse gênero não implementa transações(operações de inserir, atualizar ou deletar com a opção de fazer o *commit* ou *rollback*). O MongoDB permite que seja feito consultas, a linguagem utilizada é o JavaScript. Consultas mais complexas que utilizam os documentos embutidos em outros

[3] Sítio oficial do Redis <http://redis.io/>
[4] Sítio oficial do Riak <http://basho.com/riak/>
[5] Sítio oficial do MongoDB <http://www.mongodb.com/>
[6] Sítio oficial do CouchDB <http://couchdb.apache.org/>

documentos são realizadas com *MapReduce* (SADALAGE; FOWLER, 2013).

2.2.3 Persistência Poliglota

Diferentes bancos de dados foram desenhados para resolver diferentes problemas (SADALAGE; FOWLER, 2013). Conseguir identificar diferentes problemas 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 usado em uma mesma aplicação (WAMPLER; CLARK, 2010)

2.3 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. Para descrever essas funcionalidades segue os casos de uso abaixo:

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
3. O usuário clica em *Sign up*
4. O sistema retorna a tela com o formulário de cadastro de usuário
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*

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.

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*
2. O sistema abre a listagem dos *tweets* do usuário
3. O usuário clica no botão *New Tweet*
4. O sistema retorna com o formulário para cadastrar *tweet*
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

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*
2. O sistema abre a listagem dos *tweets* do usuário
3. O usuário clica no menu *Users*
4. O sistema retorna a listagem paginada com todos os usuários cadastrados
5. O usuário escolhe algum usuário que deseja seguir e clica no botão *Follow*
6. O sistema armazena essa informação

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*
2. O sistema abre a listagem dos *tweets* do usuário
3. O usuário clica no menu *Users*
4. O sistema retorna a listagem paginada com todos os usuários cadastrados
5. O usuário encontra o usuário que deseja parar seguir e clica no botão *Unfollow*
6. O sistema armazena essa informação

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*
2. O sistema abre a listagem dos *tweets* do usuário
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.

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*, a lista utilizada em cada aplicação criada se encontra no arquivo chamado Gemfile.lock. Podemos destacar a gem Devise [7] e a gem MongoId [8] 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.

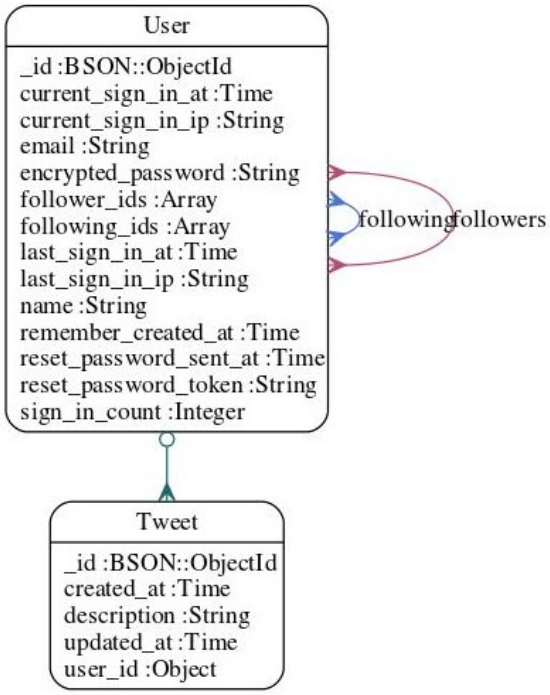
A diferença dos sistemas se descreve nas seções seguintes.

2.4 Implementação Monoglota

Essa implementação utiliza apenas o banco de dados MongoDB que é orientado a documento.

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 7 podemos visualizar o relacionamento entre as classes.

Figura 7 – Camada de controle



Fonte: (SADALAGE; FOWLER, 2013)

A classe `Tweet` armazena identificador do próprio *tweet*, data de criação, data

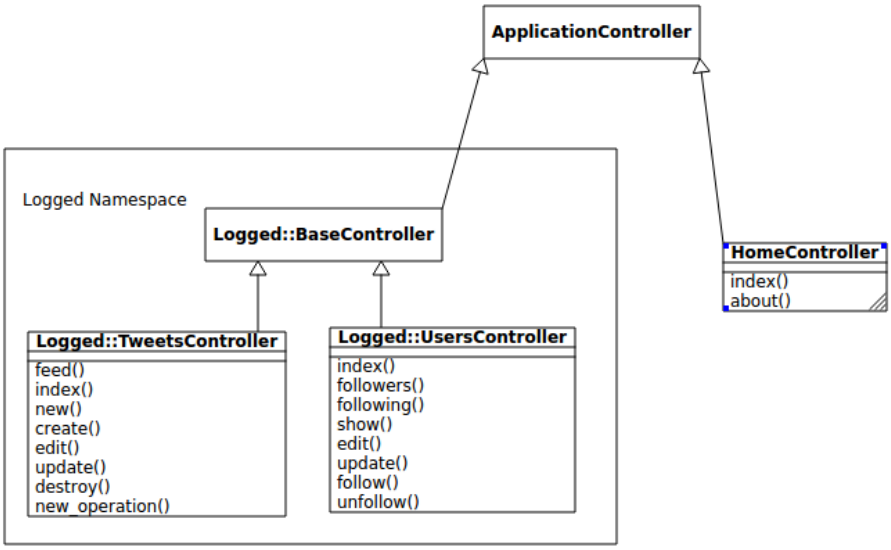
[7] <https://github.com/plataformatec/devise>
[8] <http://mongoid.org/>

da ultima atualização desse objeto, descrição do *tweet* e o identificador do usuário que criou o *tweet*. Não escolhemos usar agregação nessa relação, pois umas 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 identificadores 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 conforme mostra a Figura 8. `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 8 – Camada de controle



Fonte: (SADALAGE; FOWLER, 2013)

Toda consulta que resulte em mais de trinta registros são páginadas, 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 `limite`. 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`.

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 realizado apenas 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 realizado apenas 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 apenas 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 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`.

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 da 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 apenas uma consulta no banco de dados que busca esses *tweets*.

A ação `new` renderiza o formulário de cadastro de *tweet*, não é feito nehuma 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 desse no banco, 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.

2.5 Implementação Poliglota

Na aplicação monoglota a única maneira de consultar os *tweets* na `feed` é buscar todos os *tweets*, cujo o autor esteja 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 irá armazenar os cem *tweets* mais recentes, cujo o autor está 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 Ï_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 de alterar o sistema. Inicialmente foi necessário instalar outra *gem*, chamada redis [9], 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`.

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`: `feed_of`, `to_redis_json`,u

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.

[9] <https://github.com/antirez/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.

Algumas outras pequenas alterações foram necessárias, mas nenhuma que acrescente algum valor para o trabalho.

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 5.

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 5.

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 2 vezes nas páginas 4 e 5.

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 Programers, 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 10 vezes nas páginas 6, 7, 8, 9, 10, 11, 12, 13, 17 e 19.

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 13.